Flask-Blogging Documentation

Release 0.8.0

Gouthaman Balaraman

May 16, 2017

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
- Upload and manage static assets for the blog
- Models to store blog
- Authentication of User's choice
- Sitemap, ATOM support
- Disqus support for comments
- Google analytics for usage tracking
- Permissions enabled to control which users can create/edit blogs
- Integrated Flask-Cache based caching for optimization
- Well documented, tested, and extensible design

CHAPTER 1

Quick Start Example

```
from flask import Flask, render_template_string, redirect
from sqlalchemy import create_engine, MetaData
from flask_login import UserMixin, LoginManager, login_user, logout_user
from flask_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"
app.config["BLOGGING_SITENAME"] = "My Site"
app.config["FILEUPLOAD_IMG_FOLDER"] = "fileupload"
app.config["FILEUPLOAD_PREFIX"] = "/fileupload"
app.config["FILEUPLOAD_ALLOWED_EXTENSIONS"] = ["png", "jpg", "jpeg", "gif"]

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


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>
          <a href="/fileupload/">FileUpload</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. Please note that as of Flask-Login 0.3.0 the `is_authenticated` attribute in the `UserMixin` is a property and not a method. Please use the appropriate option based on your Flask-Login version.

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

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

```
from sqlalchemy import create_engine, MetaData

engine = create_engine("sqlite:////tmp/sqlite.db")
meta = MetaData()
storage = SQLAStorage(engine, metadata=meta)
meta.create_all(bind=engine)
```

Here we have created the storage, and created all the tables in the metadata. Once you have created the blogging engine, storage, and all the tables in the storage, you can connect with your app using the `init_app` method as shown below:

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

If you are using `Flask-Sqlalchemy`, you can do the following:

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy(app)
storage = SQLAStorage(db=db)
db.create_all()
```

One of the changes in version 0.3.1 is the ability for the user to provide the `metadata` object. This has the benefit of the table creation being passed to the user. Also, this gives the user the ability to use the common metadata object, and hence helps with the tables showing up in migrations while using Alembic.

As of version 0.5.2, support for the multi database scenario under Flask-SQLAlchemy was added. When we have a multiple database scenario, one can use the `bind` keyword in `SQLAStorage` to specify the database to bind to, as

shown below:

```
# config value
SQLALCHEMY_BINDS = {
    'blog': "sqlite:////tmp/blog.db"),
    'security': "sqlite:////tmp/security.db")
}
```

The storage can be initialised as:

```
db = SQLAlchemy(app)
storage = SQLAStorage(db=db, bind="blog")
db.create_all()
```

As of version 0.4.0, Flask-Cache integration is supported. In order to use caching in the blogging engine, you need to pass the `Cache` instance to the `BloggingEngine` as:

```
from flask_cache import Cache
from flask_blogging import BloggingEngine

blogging_engine = BloggingEngine(app, storage, cache)
```

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.

As of version 0.6.0, a plugin interface is available to add new functionality. Custom processes can be added to the `posts` by subscribing to the `post_process_before` and `post_process_after` signals, and adding new functionality to it.

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. One way to do this would be to inherit the default `PostProcessor` object and override `process` method.

In version 0.4.1 and onwards, the `BloggingEngine` object can be accessed from your `app` as follows:

```
engine = app.extensions["blogging"]
```

The engine method also exposes a `get_posts` method to get the recent posts for display of posts in other views.

In earlier versions the same can be done using the key `FLASK_BLOGGING_ENGINE` instead of `blogging`. The use of `FLASK_BLOGGING_ENGINE` key will be deprecated moving forward.

Adding Custom Markdown Extensions

One can provide additional MarkDown extensions to the blogging engine. One example usage is adding the `codehilite` MarkDown extension. Additional extensions should be passed as a list while initializing the `BlogggingEngine` as shown:

```
from markdown.extensions.codehilite import CodeHiliteExtension

extn1 = CodeHiliteExtension({})
blogging_engine = BloggingEngine(app, storage,extensions=[extn1])
```

This allows for the MarkDown to be processed using CodeHilite along with the default extensions. Please note that one would also need to include necessary static files in the `view`, such as for code highlighting to work.

Extending using Markdown Metadata

Let's say you want to include a summary for your blog post, and have it show up along with the post. You don't need to modify the database or the models to accomplish this. This is infact supported by default by way of Markdown metadata syntax. In your blog post, you can include metadata, as shown below:

```
Summary: This is a short summary of the blog post

This is the much larger blog post. There are lot of things
to discuss here.
```

In the template `page.html` this metadata can be accessed as, `post.meta.summary` and can be populated in the way it is desired. The same metadata for each post is also available in other template views like `index.html`.

Extending using the plugin framework

The plugin framework is a very powerful way to modify the behavior of the blogging engine. Lets say you want to show the top 10 most popular tag in the post. Lets show how one can do that using the plugins framework. As a first step, we create our plugin:

```
# plugins/tag_cloud/__init__.py
from flask_blogging import signals
from flask_blogging.sqlastorage import SQLAStorage
import sqlalchemy as sqla
from sqlalchemy import func


def get_tag_data(sqla_storage):
    engine = sqla_storage.engine
    with engine.begin() as conn:
        tag_posts_table = sqla_storage.tag_posts_table
        tag_table = sqla_storage.tag_table

        tag_cloud_stmt = sqla.select([
            tag_table.c.text,func.count(tag_posts_table.c.tag_id)]).group_by(
            tag_posts_table.c.tag_id
        ).where(tag_table.c.id == tag_posts_table.c.tag_id).limit(10)
        tag_cloud = conn.execute(tag_cloud_stmt).fetchall()
    return tag_cloud
```

```
def get_tag_cloud(app, engine, posts, meta, count, page):
    if isinstance(engine.storage, SQLAStorage):
        tag_cloud = get_tag_data(engine.storage)
        meta["tag_cloud"] = tag_cloud
    else:
        raise RuntimeError("Plugin only supports SQLAStorage. Given storage"
                           "not supported")
    return


def register(app):
    signals.index_posts_fetched.connect(get_tag_cloud)
    return
```

The `register` method is what is invoked in order to register the plugin. We have connected this plugin to the `index_posts_fetched` signal. So when the posts are fetched to show on the index page, this signal will be fired, and this plugin will be invoked. The plugin basically queries the table that stores the tags, and returns the tag text and the number of times it is referenced. The data about the tag cloud we are storing in the `meta["tag_cloud"]` which corresponds to the metadata variable.

Now in the *index.html* template, one can reference the `meta.tag_cloud` to access this data for display. The plugin can be registered by setting the config variable as shown:

```
app.config["BLOGGING_PLUGINS"] = ["plugins.tag_cloud"]
```

CHAPTER 3

Configuration Variables

The Flask-Blogging extension 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` (*str*): 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` (*str*): The url of the site.
- `BLOGGING_RENDER_TEXT` (*bool*): Value to specify if the raw text should be rendered or not. (default `True`)
- `BLOGGING_DISQUS_SITENAME` (*str*): Disqus sitename for comments. A `None` value will disable comments. (default `None`)
- `BLOGGING_GOOGLE_ANALYTICS` (*str*): Google analytics code for usage tracking. A `None` value will disable google analytics. (default `None`)
- `BLOGGING_URL_PREFIX` (*str*) : The prefix for the URL of blog posts. A `None` value will have no prefix (default `None`).
- `BLOGGING_FEED_LIMIT` (*int*): 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`)
- `BLOGGING_PERMISSIONS` (*bool*): if `True`, this will enable permissions for the blogging engine. With permissions enabled, the user will need to have "blogger" `Role` to edit or create blog posts. Other authenticated users will not have blog editing permissions. The concepts here derive from `Flask-Principal` (default `False`)
- `BLOGGING_PERMISSIONNAME` (*str*): The role name used for permissions. It is effective, if "BLOGGING_PERMISSIONS" is "True". (default "blogger")
- `BLOGGING_POSTS_PER_PAGE` (*int*): This sets the default number of pages to be displayed per page. (default 10)
- `BLOGGING_CACHE_TIMEOUT` (*int*): The timeout in seconds used to cache the blog pages. (default 60)
- `BLOGGING_PLUGINS` (*list*): A list of plugins to register.

CHAPTER 4

Blog Views

There are various views that are exposed through Flask-Blogging. The URL for the various views are:

- `url_for('blogging.index')` (GET): The index blog posts with the first page of articles.
- `url_for('blogging.page_by_id', post_id=<post_id>)` (GET): The blog post corresponding to the `post_id` is retrieved.
- `url_for('blogging.posts_by_tag', tag=<tag_name>)` (GET): The list of blog posts corresponding to `tag_name` is returned.
- `url_for('blogging.posts_by_author', user_id=<user_id>)` (GET): The list of blog posts written by the author `user_id` is returned.
- `url_for('blogging.editor')` (GET, POST): The blog editor is shown. This view needs authentication and permissions (if enabled).
- `url_for('blogging.delete', post_id=<post_id>)` (POST): The blog post given by `post_id` is deleted. This view needs authentication and permissions (if enabled).
- `url_for('blogging.sitemap')` (GET): The sitemap with a link to all the posts is returned.
- `url_for('blogging.feed')` (GET): Returns ATOM feed URL.

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

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

CHAPTER 5

Permissions

In version 0.3.0 Flask-Blogging, enables permissions based on Flask-Principal. This addresses the issue of controlling which of the authenticated users can have access to edit or create blog posts. Permissions are enabled by setting `BLOGGING_PERMISSIONS` to `True`. Only users that have access to `Role` "blogger" will have permissions to create or edit blog posts.

CHAPTER 6

Screenshots

Blog Page

Delete Edit New

Dirac Equation

Posted by *Paul Dirac* on 16 May, 2017

In particle physics, the Dirac equation is a relativistic wave equation derived by British physicist Paul Dirac in 1928. In its free form, or including electromagnetic interactions, it describes all spin-1/2 massive particles such as electrons and quarks for which parity is a symmetry. It is consistent with both the principles of quantum mechanics and the theory of special relativity,[1] and was the first theory to account fully for special relativity in the context of quantum mechanics. It was validated by accounting for the fine details of the hydrogen spectrum in a completely rigorous way.

The Dirac Equation is given as

$$\left(\beta mc^2 + c\left(\sum_{n=1}^{3} \alpha_n p_n\right)\right)\psi(x,t) = i\hbar\frac{\partial\psi(x,t)}{\partial t}$$

QUANTUM MECHANICS

0 Comments test Login

Recommend Share Sort by Best

Blog Editor

Title

Dirac Equation

B I H Preview

```
In particle physics, the Dirac equation is a relativistic wave equation
derived by British physicist Paul Dirac in 1928. In its free form, or
including electromagnetic interactions, it describes all spin-1/2 massive
particles such as electrons and quarks for which parity is a symmetry. It
is consistent with both the principles of quantum mechanics and the theory
of special relativity,[1] and was the first theory to account fully for
special relativity in the context of quantum mechanics. It was validated
by accounting for the fine details of the hydrogen spectrum in a
completely rigorous way.

The Dirac Equation is given as
$$
\left(\beta mc^2 +c\left(\sum_{n \mathop =1}^{3}\alpha_n p_n\right)\right)
\psi (x,t) = i \hbar \frac{\partial\psi(x,t) }{\partial t}
$$
```

Learn more about MarkDown

Upload new File

Tags

QUANTUM MECHANICS

submit

CHAPTER 7

Useful Tips

- **Migrations with Alembic**: (Applies to versions 0.3.0 and earlier) 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
```

  If you have a value set for `table_prefix` argument while creating the `SQLAStorage`, then the table names will contain that prefix in their names. In which case, you have to use appropriate names in the table names.

  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 8

Release Notes

- **Version 0.8.0** *Release May 16, 2017*
  - Added integration with Flask-FileUpload to enable static file uploads (#99)
  - Updated compatibility to latest Flask-WTF package (#96, #97)
  - Updated to latest bootstrap-markdown package (#92)
  - Added alert fade outs (#94)
- **Version 0.7.4** *Release November 17, 2016*
  - Fix Requirements.txt error
- **Version 0.7.3** *Release November 6, 2016*
  - Fix issues with slugs with special characters (#80)
- **Version 0.7.2** *Release October 30, 2016*
  - Moved default static assets to https (#78)
  - Fixed the issue where post fetched wouldn't emit when no posts exist (#76)
- **Version 0.7.1**

  Released July 5, 2016
  - Improvements to docs
  - Added extension import transition (@slippers)
- **Version 0.7.0**

  Released May 25, 2016
- **Version 0.6.0**

  Released January 14, 2016
  - The plugin framework for Flask-Blogging to allow users to add new features and capabilities.

- **Version 0.5.2**

  Released January 12, 2016

  - Added support for multiple binds for SQLAStorage

- **Version 0.5.1**

  Released December 6, 2015

  - Fixed the flexibility to add custom extensions to *BloggingEngine*.

- **Version 0.5.0**

  Released November 23, 2015

  - Fixed errors encountered while using Postgres database

- **Version 0.4.2**

  Released September 20, 2015

  - Added compatibility with Flask-Login version 0.3.0 and higher, especially to handle migration of `is_autheticated` attribute from method to property. (#43)

- **Version 0.4.1**

  Released September 16, 2015

  - Added javascript to center images in blog page
  - Added method in blogging engine to render post and fetch post.

- **Version 0.4.0**

  Released July 26, 2015

  - Integrated Flask-Cache to optimize blog page rendering
  - Fixed a bug where anonymous user was shown the new blog button

- **Version 0.3.2**:

  Released July 20, 2015

  - Fixed a bug in the edit post routines. The edited post would end up as a new one instead.

- **Version 0.3.1**:

  Released July 17, 2015

  - The `SQLAStorage` accepts metadata, and `SQLAlchemy` object as inputs. This adds the ability to keep the blogging table metadata synced up with other models. This feature adds compatibility with `Alembic` autogenerate.
  - Update docs to reflect the correct version number.

- **Version 0.3.0**:

  Released July 11, 2015

  - Permissions is a new feature introduced in this version. By setting `BLOGGING_PERMISSIONS` to `True`, one can restrict which of the users can create, edit or delete posts.
  - Added `BLOGGING_POSTS_PER_PAGE` configuration variable to control the number of posts in a page.
  - Documented the url construction procedure.

- **Version 0.2.1**:

  Released July 10, 2015

  - `BloggingEngine` `init_app` method can be called without having to pass a `storage` object.
  - Hook tests to `setup.py` script.

- **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 9

Compatibility Notes

- **Version 0.4.1**:

  The documented way to get the blogging engine from `app` is using the key `blogging` from `app.extensions`.

- **Version 0.3.1**:

  The `SQLAStorage` will accept metadata and set it internally. The database tables will not be created automatically. The user would need to invoke `create_all` in the metadata or `SQLAlchemy` object in `Flask-SQLAlchemy`.

- **Version 0.3.0**:

  - In this release, the templates folder was renamed from `blog` to `blogging`. To override the existing templates, you will need to create your templates in the `blogging` folder.
  - The blueprint name was renamed from `blog_api` to `blogging`.

- **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 10

API Documentation

Module contents

Submodules

flask_blogging.engine module

The BloggingEngine module.

class flask_blogging.engine.**BloggingEngine**(*app=None*, *storage=None*, *post_processor=None*, *extensions=None*, *cache=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_blogging import BloggingEngine, SQLAStorage
from sqlalchemy import create_engine

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

__init__(*app=None*, *storage=None*, *post_processor=None*, *extensions=None*, *cache=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*) – (optional) A list of markdown extensions to add to post processing step.
- **cache** (*Object*) – (Optional) A Flask-Cache object to enable caching

Returns

blogger_permission

get_posts(*count=10*, *offset=0*, *recent=True*, *tag=None*, *user_id=None*, *include_draft=False*, *render=False*)

classmethod get_user_name(*user*)

init_app(*app*, *storage=None*, *cache=None*)
Initialize the engine.

Parameters

- **app** (*Object*) – The app to use
- **storage** (*Object*) – The blog storage instance that implements the
- **cache** (Object `Storage` class interface.) – (Optional) A Flask-Cache object to enable caching

is_user_blogger()

process_post(*post*, *render=True*)
A high level view to create post processing. :param post: Dictionary representing the post :type post: dict :param render: Choice if the markdown text has to be converted or not :type render: bool :return:

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

flask_blogging.processor module

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

classmethod all_extensions()

classmethod construct_url(*post*)

static create_slug(*title*)

classmethod is_author(*post*, *user*)

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*)

flask_blogging.sqlastorage module

class flask_blogging.sqlastorage.**SQLAStorage**(*engine=None*, *table_prefix=''*, *metadata=None*, *db=None*, *bind=None*)

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.

__init__(*engine=None*, *table_prefix=''*, *metadata=None*, *db=None*, *bind=None*)

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 "").

Parameters

- **metadata** (*object*) – (Optional) The SQLAlchemy MetaData object
- **db** (*object*) – (Optional) The Flask-SQLAlchemy SQLAlchemy object
- **bind** – (Optional) Reference the database to bind for multiple

database scenario with binds :type bind: str

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.

engine

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.

metadata

post_table

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.

tag_posts_table

tag_table

user_posts_table

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.

flask_blogging.views module

`flask_blogging.views.`**cached_func**(*blogging_engine*, *func*)

`flask_blogging.views.`**create_blueprint**(*import_name*, *blogging_engine*)

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

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

`flask_blogging.views.`**feed**()

`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.`**sitemap**()

`flask_blogging.views.`**unless**(*blogging_engine*)

flask_blogging.forms module

class `flask_blogging.forms.`**BlogEditor**(*formdata=<object object>*, ***kwargs*)

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

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

tags = <UnboundField(StringField, ('tags',), {'validators': [<wtforms.validators.DataRequired object>]})>

text = <UnboundField(TextAreaField, ('text',), {'validators': [<wtforms.validators.DataRequired object>]})>

title = <UnboundField(StringField, ('title',), {'validators': [<wtforms.validators.DataRequired object>]})>

flask_blogging.signals module

The flask_blogging signals module

flask_blogging.**signals = <module 'flask_blogging.signals' from '/home/docs/checkouts/readthedocs.org/user_builds/flas**
The flask_blogging signals module

flask_blogging.signals.**engine_initialised = <blinker.base.NamedSignal object at 0x7fcaadc68e10; 'engine_initi**
Signal send by the BloggingEngine after the object is initialized. The arguments passed by the signal are:

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized

flask_blogging.signals.**post_processed = <blinker.base.NamedSignal object at 0x7fcaadc68e50; 'post_processed'>**
Signal sent when a post is processed (i.e., the markdown is converted to html text). The arguments passed along with this signal are:

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **post** (*dict*) – The post object which was processed
- **render** (*bool*) – Flag to denote if the post is to be rendered or not

flask_blogging.signals.**page_by_id_fetched = <blinker.base.NamedSignal object at 0x7fcaadc68e90; 'page_by_id_**
Signal sent when a blog page specified by id is fetched, and prior to the post being processed.

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **post** (*dict*) – The post object which was fetched
- **meta** (*dict*) – The metadata associated with that page
- **post_id** (*int*) – The identifier of the post
- **slug** (*str*) – The slug associated with the page

flask_blogging.signals.**page_by_id_processed = <blinker.base.NamedSignal object at 0x7fcaadc68ed0; 'page_by_**
Signal sent when a blog page specified by id is fetched, and prior to the post being processed.

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **post** (*dict*) – The post object which was processed
- **meta** (*dict*) – The metadata associated with that page
- **post_id** (*int*) – The identifier of the post
- **slug** (*str*) – The slug associated with the page

flask_blogging.signals.**posts_by_tag_fetched = <blinker.base.NamedSignal object at 0x7fcaadc68f10; 'posts_by_**
Signal sent when posts are fetched for a given tag but before processing

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized

- **posts** (*list*) – Lists of post fetched with a given tag
- **meta** (*dict*) – The metadata associated with that page
- **tag** (*str*) – The tag that is requested
- **count** (*int*) – The number of posts per page
- **page** (*int*) – The page offset

flask_blogging.signals.**posts_by_tag_processed = <blinker.base.NamedSignal object at 0x7fcaadc68f50; 'posts_**
Signal sent after posts for a given tag were fetched and processed

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched and processed with a given tag
- **meta** (*dict*) – The metadata associated with that page
- **tag** (*str*) – The tag that is requested
- **count** (*int*) – The number of posts per page
- **page** (*int*) – The page offset

flask_blogging.signals.**posts_by_author_fetched = <blinker.base.NamedSignal object at 0x7fcaadc68f90; 'posts**
Signal sent after posts by an author were fetched but before processing

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched with a given author
- **meta** (*dict*) – The metadata associated with that page
- **user_id** (*str*) – The `user_id` for the author
- **count** (*int*) – The number of posts per page
- **page** (*int*) – The page offset

flask_blogging.signals.**posts_by_author_processed = <blinker.base.NamedSignal object at 0x7fcaadc68fd0; 'po**
Signal sent after posts by an author were fetched and processed

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched and processed with a given author
- **meta** (*dict*) – The metadata associated with that page
- **user_id** (*str*) – The `user_id` for the author
- **count** (*int*) – The number of posts per page
- **page** (*int*) – The page offset

flask_blogging.signals.**index_posts_fetched = <blinker.base.NamedSignal object at 0x7fcaadc14050; 'index_post**
Signal sent after the posts for the index page are fetched

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched for the index page
- **meta** (*dict*) – The metadata associated with that page
- **count** (*int*) – The number of posts per page
- **page** (*int*) – The page offset

flask_blogging.signals.**index_posts_processed = <blinker.base.NamedSignal object at 0x7fcaadc14090; 'index_p**
Signal sent after the posts for the index page are fetched and processed

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched and processed with a given author
- **meta** (*dict*) – The metadata associated with that page
- **count** (*int*) – The number of posts per page
- **page** (*int*) – The page offset

flask_blogging.signals.**feed_posts_fetched = <blinker.base.NamedSignal object at 0x7fcaadc140d0; 'feed_posts_**
Signal send after feed posts are fetched

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched and processed with a given author

flask_blogging.signals.**feed_posts_processed = <blinker.base.NamedSignal object at 0x7fcaadc14110; 'feed_pos**
Signal send after feed posts are processed

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **feed** (*list*) – Feed of post fetched and processed

flask_blogging.signals.**sitemap_posts_fetched = <blinker.base.NamedSignal object at 0x7fcaadc14150; 'sitemap**
Signal send after posts are fetched

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched and processed with a given author

flask_blogging.signals.**sitemap_posts_processed = <blinker.base.NamedSignal object at 0x7fcaadc14190; 'siten**
Signal send after posts are fetched and processed

Parameters

- **app** (*object*) – The Flask app which is the sender

- **engine** (*object*) – The blogging engine that was initialized
- **posts** (*list*) – Lists of post fetched and processed with a given author

flask_blogging.signals.**editor_post_saved = <blinker.base.NamedSignal object at 0x7fcaadc141d0; 'editor_post_s**
Signal sent after a post was saved during the POST request

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **post_id** (*int*) – The id of the post that was deleted
- **user** (*object*) – The user object
- **post** (*object*) – The post that was deleted

flask_blogging.signals.**editor_get_fetched = <blinker.base.NamedSignal object at 0x7fcaadc14210; 'editor_get_f**
Signal sent after fetching the post during the GET request

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **post_id** (*int*) – The id of the post that was deleted
- **form** (*object*) – The form prepared for the editor display

flask_blogging.signals.**post_deleted = <blinker.base.NamedSignal object at 0x7fcaadc14250; 'post_deleted'>**
The signal sent after the post is deleted.

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **post_id** (*int*) – The id of the post that was deleted
- **post** (*object*) – The post that was deleted

flask_blogging.signals.**blueprint_created = <blinker.base.NamedSignal object at 0x7fcaadc14290; 'blueprint_cre**
The signal sent after the blueprint is created. A good time to add other views to the blueprint.

Parameters

- **app** (*object*) – The Flask app which is the sender
- **engine** (*object*) – The blogging engine that was initialized
- **blueprint** (*object*) – The blog app blueprint

flask_blogging.signals.**sqla_initialized = <blinker.base.NamedSignal object at 0x7fcaadc142d0; 'sqla_initialized**
Signal sent after the SQLAStorage object is initialized

Parameters

- **sqlastorage** (*object*) – The SQLAStorage object
- **engine** (*object*) – The blogging engine that was initialized
- **table_prefix** (*str*) – The prefix to use for tables
- **meta** (*object*) – The metadata for the database
- **bind** (*object*) – The bind value in the multiple db scenario.

CHAPTER 11

Contributors

- Gouthaman Balaraman
- adilosa
- slippers
- Sundar Raman G
- Arthur Holzner
- Sean Pianka

Python Module Index

f

Index

Symbols

A

B

C

D

E

F

G

I

M

N

P

R

S

T

U