Flask-WaffleConf Documentation

Release 0.3.1

Rafael Medina García

June 18, 2016

Contents

1 Quickstart **3**
1.1 Installation . . . 3
1.2 Configuration . . . 3
1.3 Example Application using SQLAlchemy as ORM . . . 3
1.4 Example Application using peewee as ORM . . . 4

2 Configuration **7**
2.1 WAFFLE_CONFS . . . 7
2.2 WAFFLE_MULTIPROC . . . 7
2.3 WAFFLE_WATCHTYPE . . . 8
2.4 WAFFLE_WATCHER_FILE . . . 8
2.5 WAFFLE_REDIS_HOST . . . 8
2.6 WAFFLE_REDIS_PORT . . . 8
2.7 WAFFLE_REDIS_CHANNEL . . . 8
2.8 Deprecated . . . 8

3 Multiprocess deployments **11**
3.1 Problem . . . 11
3.2 Solution (kind of) . . . 11
3.3 Setup for multiprocess deployments . . . 11

4 Usage in views **13**
4.1 Initialization . . . 13
4.2 Obtaining stored values . . . 13
4.3 Updating stored values . . . 14

5 API reference **15**
5.1 flask_waffleconf.core . . . 15
5.2 flask_waffleconf.models . . . 16
5.3 flask_waffleconf.store . . . 16
5.4 flask_waffleconf.util . . . 18
5.5 flask_waffleconf.watcher . . . 18

6 Indices and tables **19**

Python Module Index **21**

WaffleConf is a Flask extension that enables storage of configuration variables in the database as well as runtime modification of said variables.

Released under GPLv2+ license.

Latest version: **0.3.0**

Contents:

CHAPTER 1

Quickstart

1.1 Installation

WaffleConf only has *Flask* as a *hard* requirement. You can install the extension by running:

```
pip install Flask-WaffleConf
```

1.2 Configuration

Simple usage of the extension requires the following configuration variables (e.g., in your application's `config.py`):

- **`WAFFLE_CONFS`: used for specifying the configuration variables that are** going to be stored in the database. It has the following structure:

```
WAFFLE_CONFS = {
     'MAX_FILESIZE': {
         'desc': 'Max upload filesize (in bytes)',
         'default': 1000
     },

     'SITENAME': {
         'desc': 'Name of the site appearing in the header',
         'default': 'Waffle'
     }
 }
```

For more detailed information, check Configuration.

1.3 Example Application using SQLAlchemy as ORM

```
    from flask import Flask, current_app
    from flask_waffleconf import WaffleConf, AlchemyWaffleStore, \
            WaffleMixin
    from flask_sqlalchemy import SQLAlchemy

    app = Flask(__name__)
    app.config['WAFFLE_CONFS'] = {
            'MAX_FILESIZE': {
                    'desc': 'Max upload filesize (in bytes)',
```

```
                        'default': 1000
                },

                'SITENAME': {
                        'desc': 'Name of the site appearing in the header',
                        'default': 'Waffle'
                }
        }

        # Define your database
        # db = ...

        # Define model
        class ConfModel(db.Model, WaffleMixin):
        __tablename__ = 'confs'

        id = db.Column(db.Integer, primary_key=True)
                key = db.Column(db.String(255), unique=True)
                value = db.Column(db.Text)

        # Create database tables
        # ...

        # Initialize WaffleConf
        configstore = AlchemyWaffleStore(db=db, model=ConfModel)
        waffle = WaffleConf(app, configstore)

@app.before_first_request
def do_before():
        """Update stored configurations."""
        waffle.state.update_conf()

        @app.route('/')
        def index():
                """Display content of configured variable 'SITENAME'."""
                state = current_app.extensions['waffleconf']

                parsed = state.parse_conf()
                # {'MAX_FILESIZE': 1000, 'SITENAME': 'Waffle'}

                return parsed['SITENAME']
```

1.4 Example Application using peewee as ORM

```
        from flask import Flask, current_app
        from flask_waffleconf import WaffleConf, PeeweeWaffleStore, \
                WaffleMixin
        import peewee

        app = Flask(__name__)
        app.config['WAFFLE_CONFS'] = {
                'MAX_FILESIZE': {
                        'desc': 'Max upload filesize (in bytes)',
                        'default': 1000
                },
```

```
            'SITENAME': {
                    'desc': 'Name of the site appearing in the header',
                    'default': 'Waffle'
            }
    }

    # Define your database
    # db = ...

    # Define model
    class ConfModel(peewee.Model, WaffleMixin):
            class Meta:
                    database = db

            key = peewee.CharField(unique=True)
            value = peewee.TextField()

    # Create database tables
    # ...

    # Initialize WaffleConf
    configstore = PeeweeWaffleStore(model=ConfModel)
    waffle = WaffleConf(app, configstore)

@app.before_first_request
def do_before():
    """Update stored configurations."""
    waffle.state.update_conf()


    @app.route('/')
    def index():
            """Display content of configured variable 'SITENAME'."""
            state = current_app.extensions['waffleconf']

            parsed = state.parse_conf()
            # {'MAX_FILESIZE': 1000, 'SITENAME': 'Waffle'}

            return parsed['SITENAME']
```

CHAPTER 2

Configuration

The extension relies on several configuration variables that should be present in the `config` dict from the Flask application it is attached to.

2.1 WAFFLE_CONFS

The `WAFFLE_CONFS` variable is used to specify information on each of the variables that are going to be stored in the database:

```
WAFFLE_CONFS = {
              'MAX_FILESIZE': {
                      'desc': 'Max upload filesize (in bytes)',
                      'default': 1000
              },

              'SITENAME': {
                      'desc': 'Name of the site appearing in the header',
                      'default': 'Waffle'
              }
      }
```

It is a simple `dict` that uses the name of the configuration variable to store as key and stores child `dict` objects with the following attributes:

- `desc`: human-readable name or short description of the variable
- **`default`: default value when the variable does not exist in database (Must be picklable)**

Note: Only variables that appear in this dict can be updated during runtime.

Changed in 0.3.0: the `type` field is deprecated as values are serialized when stored in the database and deserialized when obtained with *`parse_conf()`*.

2.2 WAFFLE_MULTIPROC

When set to `True`, the extension will take into account multiprocess deployments. See Multiprocess deployments for more information.

Defaults to `False`.

2.3 WAFFLE_WATCHTYPE

Specifies the medium that will be used to notify other application instances when there is a change in the variables stored in database. Supported values are:

- `'file'`: use timestamps of a plain file in the filesystem (default)
- `'redis'`: use a Redis channel with pub/sub

Added in version 0.3.0.

2.4 WAFFLE_WATCHER_FILE

Path to the file to use when using a file watcher to check for updates. This file is polled every *10 seconds* in a separate thread and its timestamp checked against the one stored in the `_WaffleState` object instance.

Defaults to `'/tmp/waffleconf.txt'`.

> **Warning:** Make sure that the user running the application has the necessary permissions to check and update the timestamp of the file.

Added in version 0.3.0.

2.5 WAFFLE_REDIS_HOST

When using Redis for update notifications, this variable is used to determine the host to be used in the connection.

Defaults to `'localhost'`.

2.6 WAFFLE_REDIS_PORT

When using Redis for update notifications, this variable is used to determine the port to be used in the connection.

Defaults to `6379`.

2.7 WAFFLE_REDIS_CHANNEL

When using Redis for update notifications, this variable is used to determine the channel to be used for the pub/sub messages.

Defaults to `'waffleconf'`.

2.8 Deprecated

The following variables are deprecated in the latest version of the extension.

2.8.1 WAFFLE_TEMPLATE

Deprecated in version 0.3.0: the extension no longer uses any views or templates.

The extension only uses a single template that contains a form for displaying and updating the values.

You are highly encouraged to extend this template.

Defaults to `'waffleconf/waffle_form'`.

CHAPTER 3

Multiprocess deployments

3.1 Problem

The simple usage shown in the Quickstart works fine when your deployed application uses a single process (i.e., a single app instance). However, chances are you are using something like uWSGI, and your application is served using several processes/workers.

If this is the case, when one of the workers (`app` instances) updates its configuration values, the rest of the workers will still have their old configuration values. If these old values are then updated, your application workers will have inconsistent configuration variables all the way.

3.2 Solution (kind of)

In *version 0.2.0*, `Flask-WaffleConf` introduced support for multprocess deployments by allowing these processes to listen for and send updates through a *Redis channel*.

Version 0.3.0 introduced a simpler approach by using *stat* information from a plain file in the filesystem.

The mode to use can be configured setting the value of the `WAFFLE_WATCHTYPE` variable (see Configuration).

Note: As both approaches use threads, the GIL should be taken into account, although it should not be much of an issue performance-wise.

Note: Application servers (such as uWSGI) may require additional configuration to enable threads in hosted applications. For uWSGI, for instance, the `--enable-threads` and `--lazy-apps` flags are needed.

3.3 Setup for multiprocess deployments

Changed in version 0.3.0: `gevent` support was removed as the implementation was not correct.

3.3.1 File watching

Using a file is the simplest approach for notifications and does not require any additional dependencies. Simply set `WAFFLE_WATCHTYPE` to `'file'` and change the `WAFFLE_WATCHER_FILE` to a valid filesystem path. If it does

not exist, it will be created automatically.

Added in version 0.3.0.

3.3.2 Redis pub/sub

Using Redis for notifications requires a valid connection to a Redis server (usually in localhost), as well as the `redis-py` module. Local installation in Debian systes would be done like this:

```
apt-get install redis-server

pip install redis
```

Next, it is necessary to set additional configuration variables in the configuration of the application:

```
# Enable multiprocess use
WAFFLE_MULTIPROC = True

# Redis host (defaults to 'localhost')
WAFFLE_REDIS_HOST = 'MY_HOST'

# Redis port (defaults to 6379)
WAFFLE_REDIS_PORT = 6379

# The channel to listen and send signals to (defaults to 'waffleconf')
WAFFLE_REDIS_CHANNEL = 'MY_CHANNEL'
```

Once the extension is initialized, the listener will be automatically created.

Note: Configuring the extension to use Redis without the `redis-py` module installed will fallback to the default *file watcher* configuration.

CHAPTER 4

Usage in views

Since *version 0.3.0* the extension does not impose a specific view or template to use. Instead, you can implement your own views and work with the `_WaffleState()` instance in the application.

4.1 Initialization

To initialize the extension, two different things are required: a model implementing the `WaffleMixin` interface, for instance using SQLAlchemy or peewee; and a configured `WaffleStore`.

As of *version 0.3.0*, there are two stores available (although it is very easy to create a new one using the `WaffleStore` class as a base):

- **`AlchemyWaffleStore`: uses `SQLAlchemy`** for the database backend
- **`PeeweeWaffleStore`: uses `peewee`** for the database backend

Changed in 0.3.1: stored configurations are not updated when the extension is initialized and require manually calling the update method.

Note: Model and store should use the same ORM/library as backend.

4.2 Obtaining stored values

The following simple views are an example of how you can use the extension to parse stored values of configuration variables:

```
from flask import current_app

@app.route('/all')
def get_all():
    """Returns the whole list of stored configuration variables."""
    state = current_app.extensions['waffleconf']

    # Get all the variables
    parsed = state.parse_conf() # Returns a dict

    return parsed

@app.route('/<key>')
```

```
def get_key(key):
    """Return the value of a single key."""
    state = current_app.extensions['waffleconf']

    # Get variable
    parsed = state.parse_conf([key,]) # Returns a dict

    return parsed
```

As the `parse_conf()` method returns a Python `dict`, creating a form for showing or updating the values is very easy.

4.3 Updating stored values

Similarly, it is also possible to update values at runtime using a custom view:

```
from flask import current_app, form

 @app.route('/update', methods=['POST'])
 def update_vars():
     """Update the vars with the values of a hypothetical form."""
     # Suppose WTForms with fields `SITENAME` and `DESCRIPTION`
     form = Form(request.form)

     if form.validate():
         vals = {
             'SITENAME': form.sitename.data,
             'DESCRIPTION': form.desc.data
         }

         state = current_app.extensions['waffleconf']
         state.update_db(vals)
```

CHAPTER 5

API reference

5.1 flask_waffleconf.core

class `flask_waffleconf.core.`**`WaffleConf`**(*app=None*, *configstore=None*)

Bases: `object`

Initialize the Flask-WaffleConf extension

Parameters

- **app** – Flask application instance
- **configstore** (`WaffleStore`) – database store.

`state`

_WaffleState

Direct access to the initialized state object.

`init_app`(*app*, *configstore*)

Initialize the extension for the given application and store.

Parse the configuration values stored in the database obtained from the `WAFFLE_CONFS` value of the configuration.

Parameters

- **app** – Flask application instance
- **configstore** (`WaffleStore`) – database store.

class `flask_waffleconf.core.`**`_WaffleState`**(*app*, *configstore*)

Bases: `object`

Store configstore for the app state.

This object will update its application's configuration and if the `WAFFLE_MULTIPROC` setting is set to True, it will also notify other processes using Redis channel or file timestamp.

Parameters

- **app** – Flask application instance.
- **configstore** (`WaffleStore`) – database store.

`parse_conf`(*keys=[]*)

Parse configuration values from the database.

The extension must have been previously initialized.

If a key is not found in the database, it will be created with the default value specified.

Parameters **keys** (*list[str]*) – list of keys to parse. If the list is empty, then all the keys known to the application will be used.

Returns dict of the parsed config values.

update_conf()

Update configuration values from database.

This method should be called when there is an update notification.

update_db(*new_values*)

Update database values and application configuration.

The provided keys must be defined in the `WAFFLE_CONFS` setting.

Parameters **new_values** (*dict*) – dict of configuration variables and their values The dict has the following structure:

{ 'MY_CONFIG_VAR' : <CONFIG_VAL>, 'MY_CONFIG_VAR1' : <CONFIG_VAL1>

}

5.2 flask_waffleconf.models

class flask_waffleconf.models.**WaffleMixin**

Bases: `object`

Mixin used in the creation of the database model.

WaffleConf expects a model that has (at least) the following fields:

- key (str): Unique identifier for configuration variable.
- value (str): Value for the configuration variable.

Values may be a large string, so make sure to define a field capable of storing big strings. These values are later parsed according to the `type` specified in the application configuration.

get_key()

Obtain the key for the configuration variable.

Mixin expects a `self.key` attribute (str) in the model. If this is not the case, you should override this method.

Returns Configuration key string.

get_value()

Obtain the value for the configuration variable.

Mixin expects a `self.value` attribute (str) in the model. If this is not the case, you should override this method.

Returns Configuration value string.

5.3 flask_waffleconf.store

class flask_waffleconf.store.**AlchemyWaffleStore**(*db=None*, *model=None*)

Bases: *flask_waffleconf.store.WaffleStore*

Config store for SQLAlchemy.

commit()

delete(*key*)

get(*key*)

put(*key*, *value*)

class flask_waffleconf.store.**PeeweeWaffleStore**(*db=None*, *model=None*)
Bases: *flask_waffleconf.store.WaffleStore*

Config store for peewee.

commit()

delete(*key*)

get(*key*)

put(*key*, *value*)

class flask_waffleconf.store.**WaffleStore**(*db=None*, *model=None*)
Bases: object

Object for connecting to the application database.

Offers common methods that have to be overriden depending on the database type / ORM used.

Parameters

- **db** – Database instance.
- **model** – Model to work with.

commit()
Commit to database where needed.

delete(*key*)
Remove a configuration variable from the database.

Parameters **key** (*str*) – Name of the configuration variable to delete.

Returns Deleted record or None if it could not be deleted.

get(*key*)
Obtain a configuration variable from the database.

Parameters **key** (*str*) – Name of the configuration variable to obtain.

Returns Record or None if record could not be obtained.

put(*key*, *value*)
Insert / Update a configuration variable in the database.

Parameters

- **key** (*str*) – Name of the configuration variable that is being updated.
- **value** – Value to store in the database (serialized).

Returns Updated record or None on error.

5.4 flask_waffleconf.util

`flask_waffleconf.util.`**`deserialize`**(*data*)

Deserialize data using base64 encoding and pickle.

Parameters **data** (*str*) – Data to deserialize (must be in base64 and pickled)

Returns Unpickled object.

`flask_waffleconf.util.`**`serialize`**(*data*)

Serialize data using pickle and converting it to a base64 string.

Parameters **data** – data to serialize (must be picklable)

Returns Serialized object.

5.5 flask_waffleconf.watcher

`flask_waffleconf.watcher.`**`_dummy`**(*state*)

Does nothing.

`flask_waffleconf.watcher.`**`_file_notifier`**(*state*)

Notify of configuration update through file.

Parameters **state** (`_WaffleState`) – Object that contains reference to app and its configstore.

`flask_waffleconf.watcher.`**`_file_watcher`**(*state*)

Watch for file changes and reload config when needed.

Parameters **state** (`_WaffleState`) – Object that contains reference to app and its configstore.

`flask_waffleconf.watcher.`**`_redis_notifier`**(*state*)

Notify of configuration update through redis.

Parameters **state** (`_WaffleState`) – Object that contains reference to app and its configstore.

`flask_waffleconf.watcher.`**`_redis_watcher`**(*state*)

Listen to redis channel for a configuration update notifications.

Parameters **state** (`_WaffleState`) – Object that contains reference to app and its configstore.

`flask_waffleconf.watcher.`**`get_notifier`**(*notifier_type*)

Obtain a notifier function.

Parameters **notifier_type** (*str*) – Either 'file' or 'redis'. If redis is not available, it will default to file watcher.

Returns Notifier function.

`flask_waffleconf.watcher.`**`get_watcher`**(*watcher_type*)

Obtain a watcher function.

These functions should be executed in a separate thread.

Parameters **watcher_type** (*str*) – Either 'file' or 'redis'. If redis is not available, it will default to file watcher.

Returns Watcher function.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

flask_waffleconf.core, 15
flask_waffleconf.models, 16
flask_waffleconf.store, 16
flask_waffleconf.util, 18
flask_waffleconf.watcher, 18

Symbols

_WaffleState (class in flask_waffleconf.core), 15
_dummy() (in module flask_waffleconf.watcher), 18
_file_notifier() (in module flask_waffleconf.watcher), 18
_file_watcher() (in module flask_waffleconf.watcher), 18
_redis_notifier() (in module flask_waffleconf.watcher), 18
_redis_watcher() (in module flask_waffleconf.watcher), 18

A

AlchemyWaffleStore (class in flask_waffleconf.store), 16

C

commit() (flask_waffleconf.store.AlchemyWaffleStore method), 17
commit() (flask_waffleconf.store.PeeweeWaffleStore method), 17
commit() (flask_waffleconf.store.WaffleStore method), 17

D

delete() (flask_waffleconf.store.AlchemyWaffleStore method), 17
delete() (flask_waffleconf.store.PeeweeWaffleStore method), 17
delete() (flask_waffleconf.store.WaffleStore method), 17
deserialize() (in module flask_waffleconf.util), 18

F

flask_waffleconf.core (module), 15
flask_waffleconf.models (module), 16
flask_waffleconf.store (module), 16
flask_waffleconf.util (module), 18
flask_waffleconf.watcher (module), 18

G

get() (flask_waffleconf.store.AlchemyWaffleStore method), 17
get() (flask_waffleconf.store.PeeweeWaffleStore method), 17
get() (flask_waffleconf.store.WaffleStore method), 17
get_key() (flask_waffleconf.models.WaffleMixin method), 16
get_notifier() (in module flask_waffleconf.watcher), 18
get_value() (flask_waffleconf.models.WaffleMixin method), 16
get_watcher() (in module flask_waffleconf.watcher), 18

I

init_app() (flask_waffleconf.core.WaffleConf method), 15

P

parse_conf() (flask_waffleconf.core._WaffleState method), 15
PeeweeWaffleStore (class in flask_waffleconf.store), 17
put() (flask_waffleconf.store.AlchemyWaffleStore method), 17
put() (flask_waffleconf.store.PeeweeWaffleStore method), 17
put() (flask_waffleconf.store.WaffleStore method), 17

S

serialize() (in module flask_waffleconf.util), 18
state (flask_waffleconf.core.WaffleConf attribute), 15

U

update_conf() (flask_waffleconf.core._WaffleState method), 16
update_db() (flask_waffleconf.core._WaffleState method), 16

W

WaffleConf (class in flask_waffleconf.core), 15
WaffleMixin (class in flask_waffleconf.models), 16
WaffleStore (class in flask_waffleconf.store), 17