Flask-Constance

Release 0.1.2

Ivan Fedorov

Sep 13, 2023

CONTENTS:

Dynamic settings for your Flask application.

CHAPTER

ONE

INSTALLATION

1.1 Python Version

Flask-Constance supports Python 3.6 and newer. The choice of the minimum version is due to the fact that the author needs to support several applications that work only on this version. Until this changes, the minimum version will not be increased.

1.2 Dependencies

The only required dependency of this extension is Flask. However, to use the various backends, one way or another, you will need to install additional packages. They are defined through optional dependencies.

The following optional dependencies are currently available:

- **[fsqla]** - for Flask-SQLAlchemy backend.

1.3 Installation from PyPI

This is the most common way to install Flask-Constance package.

```
python3 -m pip install flask-constance
```

And this is command to install package with optional dependencies related to Flask-SQLAlchemy backend.

```
python3 -m pip install flask-constance[fsqla]
```

1.4 Building from source

To install a package from source, you first need to clone the repository.To install package from source.

```
git clone https://github.com/TitaniumHocker/Flask-Constance.git Flask-Constance
```

Then you can install it with pip.

```
pytho3 -m pip install ./Flask-Constance
```

1.5 Signalling support

If you wish to use Flask-Constance signals, ensure that *blinker* package is installed.

```
python3 -m pip install blinker
```

CHAPTER

TWO

QUICKSTART

Here some simple tutorial how to get started with Flask-Constance after the installation.

2.1 Import and initialize

First of all you need import and initialize extension with your application and selected backend. In this example Flask-SQLAlchemy will be used as backend.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_constance import Constance
from flask_constance.backends import \
    FlaskSQLAlchemyBackend, SettingMixin

# Initialize application and Flask-SQLAlchemy.
app = Flask(__name__)
app.config["SERCET_KEY"] = "super-secret"
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///:memory:"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
db = SQLAlchemy(app)

# Define model for the backend.
class Setting(db.Model, SettingMixin):
    pass

# Finally, initialize Flask-Constance.
constance = Constance(app, FlaskSQLAlchemyBackend(Setting, db.session))

# Also you can use init_app method if you want.
# constance = Constance(backend=FlaskSQLAlchemyBackend(Setting, db.session))
# constance.init_app(app)
```

2.2 Describe settings

To set up some settings and their default values they need do be defined via config value *CONSTANCE_PAYLOAD*. This must be a dictionary where key is a setting name and value - default value for this setting.

```
app.config["CONSTANCE_PAYLOAD"] = {
    "foo": "bar",
    "hello": "world",
}
```

2.3 Use it

After connecting Flask-Constance with your application you finally can use global *settings* object to read or modify them.

Note: Please note that the *settings* object is only available in the application context. In views for example.

```
from flask import jsonify, request
from flask_constance import settings

@app.route("/")
def index():
    """This view will return current settings as a JSON."""
    if settings.foo == "bar":
        settings.foo == "not bar"
    elif settings.foo == "not bar":
        settings.foo == "bar"
    return jsonify({key: getattr(settings, key) for key in dir(settings)})
```

2.4 Full example

Here is a full example from *examples* directory of the project repo.

```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy

from flask_constance import Constance, settings
from flask_constance.backends.fsqla import FlaskSQLAlchemyBackend, SettingMixin

app = Flask(__name__)
app.config["SERCET_KEY"] = "super-secret"
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///:memory:"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
app.config["CONSTANCE_PAYLOAD"] = {"foo": "bar"}

db = SQLAlchemy(app)
```

(continues on next page)

(continued from previous page)

```
class ConstanceSettings(db.Model, SettingMixin):  # type: ignore
    pass


constance = Constance(app, FlaskSQLAlchemyBackend(ConstanceSettings, db.session))


@app.route("/")
def index():
    """This view will return current settings as a JSON."""
    if settings.foo == "bar":
        settings.foo = "not bar"
    elif settings.foo == "not bar":
        settings.foo = "bar"
    return jsonify({key: getattr(settings, key) for key in dir(settings)})


@app.route("/<name>", methods=["POST"])
def update(name: str):
    if request.json is None:
        return {}, 400
    setattr(settings, name, request.json)
    return {key: getattr(settings, key) for key in dir(settings)}


def main():
    db.create_all()
    app.run("0.0.0.0", 5000, True)


if __name__ == "__main__":
    main()
```

CHAPTER

THREE

GUIDE

The following is a more advanced guide to using the extension.

3.1 Initializing methods

As most extensions, Flask-Constance can be initialized in two ways:

1. Pass application object to extension `__init__` method:

```
from flask import Flask
from flask_constance import Constance

app = Flask(__name__)
constance = Constance(app)
```

2. Use `init_app` method:

```
from flask import Flask
from flask_constance import Constance

app = Flask(__name__)
constance = Constance()
constance.init_app(app)
```

These two methods are equal. Extension constructor just calls *init_app* method if application object was provided.

3.2 Naming restrictions

There are few restrictions on settings naming:

- Names can't contain - character.
- Names can't starts with underscore _ character.
- Names can't starts with number.

Given that the settings will be accessed through the class attributes of the global object *settings*, it is desirable that the settings names be valid for use as class attribute names.

3.3 Managing settings

For accessing dynamic settings provided by Flask-Constance extension there is global object *settings*. Actually this in werkzeug's LocalProxy object pointing to *Storage* instance. `Storage` object defines `__getattr__`, `__setattr__` and `__delattr__` methods to access settings. So you can access your settings like normal attributes of `Storage` object.

For example you defined some settings via `CONSTANCE_PAYLOAD` application config value:

```
app.config["CONSTANCE_PAYLOAD"] = {
    "foo": "bar",
    "hello": "world",
}
```

Then they becomes accessable with global `settings` object:

```
from flask_constance import settings

assert settings.foo == "bar"
assert settings.hello == "world"
```

They can be updated like normal class attributes:

```
from flask_constance import settings

settings.foo = "not bar"  # Will be updated in backend.
assert settings.foo == "not bar"
```

Also they can be deleted to reset them to default value:

```
from flask_constance import settings

assert settings.foo == "bar"     # default value.
settings.foo = "not bar"         # updating.
assert settings.foo == "not bar" # updated value.
del settings.foo                 # resetting to default value.
assert settings.foo == "bar"     # default value
```

3.4 Supported backends

Backend in Flask-Constance terminology is actual storage of settings values. When settin first accessed it will be stored in connected backend.

3.4.1 Flask-SQLAlchemy

Most common backend is *Flask-SQLAlchemy* backend provided via `FlaskSQLAlchemyBackend` class. This backend stores settings in database configured with *SQLAlchemy*. To initialize this backend *SQLAlchemy* session and corresponding model must be provided.

Database model must have some required fields:

- `name` field with required unique string type.
- `value` field with JSON type.

Here is an example of using *Flask-SQLAlchemy* backend:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_constance import Constance
from flask_constance.backends import FlaskSQLAlchemyBackend
import sqlalchemy as sa

app = Flask(__name__)
db = SQLAlchemy(app)

class Setting(db.Model):
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String, unique=True, nullable=False, index=True)
    value = sa.Column(sa.JSON, nullable=True)

constance = Constance(app, FlaskSQLAlchemyBackend())
```

There is predefined model mixin with all required fields. Here is same example with using this mixin:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_constance import Constance
from flask_constance.backends import \
    FlaskSQLAlchemyBackend, SettingMixin
import sqlalchemy as sa

app = Flask(__name__)
db = SQLAlchemy(app)

class Setting(db.Model, SettingMixin):
    pass

constance = Constance(app, FlaskSQLAlchemyBackend())
```

3.5 Caching

By default Flask-Constance will cache settings that are accessed in Flask's `g` object. This object recreates on every request, so this caching mechanism is very limited. It can reduce number of access operations to the backend during request, but nothing more.

In the future there are plans to implement some external backend cache support. For now it can be implemented by hand with *BackendCache* base class and passed to *Constance*.

See *Implementing your own backend or cache* section for additional information.

3.6 RESTlike view

If you need to manage settings via HTTP API - there is simple implementation of RESTlike view *ConstanceView*. To enable it just set `CONSTANCE_VIEW_BASE_URL` config value.

For example if config value set to `/api/constance` then this operations can be done with HTTP API:

- `GET` on `/api/constance` to get all settings values.
- `GET` on `/api/constance/<name>` to get specific config value by it's name.
- `PUT` on `/api/constance/<name>` to update specific config value by it's name.
- `DELETE` on `/api/constance/<name>` to reset specific config value by it's name.

`PUT` request accepts JSON as a payload. `GET` requests returns JSON as response payload. If setting not found by it's name - 404 status will be returned.

Here is an example how you can connect this API:

```
from flask import Flask
from flask_constance import Constance

app = Flask(__name__)
app.config["CONSTANCE_PAYLOAD"] = {"foo": "bar"}
app.config["CONSTANCE_VIEW_BASE_URL"] = "/api/constance"

if __name__ == "__main__":
    app.run(debug=True)
```

And then use it:

```
$ curl -X GET http://localhost:5000/api/constance
{
"foo": "bar"
}
$ curl -X PUT http://localhost:5000/api/constance/foo \
    -H "Content-Type: application/json" \
    -d '"new-data"'
{}
$ curl -X GET http://localhost:5000/api/constance
{
"foo": "new-data"
}
$ curl -X DELETE http://localhost:5000/api/constance/foo
```

(continues on next page)

(continued from previous page)

```
{}
$ curl -X GET http://localhost:5000/api/constance
{
"foo": "bar"
}
```

3.7 CLI

Settings can be accessed from simple CLI interface.

- `flask constance get` command for reading values.
- `flask constance set` for updating.
- and `flask constance del` for deleting(resetting) values.

3.8 Signals

Flask-Constance supports Flask's signalling feature via *blinker* package. Extension sends signalls in these cases:

- When extension was initialized: *constance_setup*.
- When setting value was accessed: *constance_get*.
- When setting value was updated: *constance_set*.

3.9 Implementing your own backend or cache

If you want to implement your own backend or backend cache there is two base classes - *Backend* and *BackendCache*.

In general backend must implement to methods:

- `set` to set setting value, that takes name and value as arguments.
- `get` to get setting value by it's name.

For backend cache signature is the same, except that in addition `invalidate` method must be implemented. This method deletes value from the cache by it's name.

Here is an example of a backend cache that uses memcached(pymemcache):

```
import typing as t
import os
import json
from pymemcache.client.base import Client
from flask_constance.backends.base import BackendCache

class MemcachedBackendCache(BackendCache):
    def __init__(self, addr: str):
        self.client = Client(addr)

    def get(self, name: str) -> t.Any:
```

(continues on next page)

(continued from previous page)

```
        return json.loads(self.client.get(name))

    def set(self, name: str, value: t.Any):
        self.client.set(name, json.dumps(value))

    def invalidate(name: str):
        self.client.delete(name)
```

3.10 Configuration

As usual for Flask extensions, Flask-Constance configuration variables stored in Flask.config object with `CONSTANCE_` prefix. Here is configuration variables of Flask-Constance extension:

Name	Description	Type	Default
CONSTANCE_PAYLOAD	Dictionay with settings.	Dict	Empty dict
CONSTANCE_VIEW_BASE_URL	Base url for RESTlike view	str or None	None

3.11 Flask-Admin integration

If you use `Flask-Admin`, then there is an integration for this extension.

```
from flask import Flask
from flask_admin import Admin
from flask_constance import Constance, settings
from flask_constance.admin import ConstanceAdminView
from flask_constance.backends.fsqla import FlaskSQLAlchemyBackend, SettingMixin

app = Flask(__name__)
app.config["SECRET_KEY"] = "super-secret"
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///:memory:"
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
app.config["CONSTANCE_PAYLOAD"] = {"foo": "bar", "hello": "world"}
db = SQLAlchemy(app)

class ConstanceSettings(db.Model, SettingMixin):  # type: ignore
    pass

constance = Constance(app, FlaskSQLAlchemyBackend(ConstanceSettings, db.session))

admin = Admin(app, template_mode="bootstrap4")
admin.add_view(ConstanceAdminView(name="Settings", endpoint="settings"))
```

CHAPTER

FOUR

API

Here you can find part of documentation that covers all public interfaces of Flask-Constance.

4.1 Extension object

class flask_constance.**Constance**(*app=None*, *backend=None*, *cache=None*, *view_base_url=None*, *view_class=<class 'flask_constance.view.ConstanceView'>*)

Bases: `object`

Constance extension

Parameters

- **app** (`Optional[Flask]`) – Flask application.
- **backend** (`Optional[Backend]`) – Backend instance to use.
- **backend_cache** – Cache for the backend.
- **view_base_url** (`Optional[str]`) – Base URL to register REST-like view for settings.
- **view_class** (`Type[ConstanceView]`) – Class to register as view.

init_app(*app*)

Initialize extension with Flask application.

Parameters
app (`Flask`) – Flask application.

Raises

- **RuntimeError** – If Constance extension was already initialized.
- **ValueError** – If invalid setting name was provided in CONSTANCE_PAYLOAD application config value.

Return type
`None`

4.2 Global settings object

flask_constance.**settings**

With this global you can access settings from any point of your application. This object actually is werkzeug's LocalProxy pointing to *Storage* object.

4.3 Storage aka settings object

class flask_constance.storage.**Storage**(*backend*, *cache=None*)

Bases: object

Flask-Constance settings storage.

This is access point for getting and setting constance settings.

Parameters

- **backend** (*Backend*) – Backend instance to use. By default is Memory backend.
- **cache** (Optional[*BackendCache*]) – Caching backend to use. This is optional.

4.4 Backends

Flask-Constance supports various types of backends. All of them implements *Backend* or *flask_constance.backends.base.BackendCache* protocols.

4.4.1 Backend Protocol

class flask_constance.backends.base.**Backend**(**args*, ***kwargs*)

Bases: Protocol

Base backend class.

get(*name*)

Return type
Any

set(*name*, *value*)

Return type
None

4.4.2 Backend Cache Protocol

class flask_constance.backends.base.**BackendCache**(**args*, ***kwargs*)

Bases: Protocol

Base backend cache class.

get(*name*)

Return type
Any

invalidate(*name*)

Return type
None

set(*name*, *value*)

Return type
None

4.4.3 Memory Backend object

Memory backend was implemented for testing purposes. It can be used only in single-process mode, so it really doesn't fit production environment requirenments.

class flask_constance.backends.memory.**MemoryBackend**

Bases: *Backend*

In-memory backend for testing purposes.

get(*name*)

Get setting value.

Parameters
key – Name of the setting.

Return type
Any

set(*name*, *value*)

Set setting value

Parameters

- **key** – Name of the setting.
- **value** (Any) – Value of the setting.

Return type
None

4.4.4 Flask-SQLAlchemy backend object

This backend implements intergration with Flask-SQLAlchemy extension as main settings storage.

class flask_constance.backends.fsqla.**FlaskSQLAlchemyBackend**(*model*, *session*)

Bases: *Backend*

Flask-SQLAlchemy backend

Parameters

- **model** (DeclarativeMeta) – Model which describes settings.
- **session** (scoped_session) – Database session.

get(*name*)

Get setting value.

Parameters
key – Name of the setting.

Return type
Any

set(*name*, *value*)

Set setting value

Parameters

- **key** – Name of the setting.
- **value** (Any) – Value of the setting.

Return type
None

4.4.5 Flask-SQLAlchemy model mixin

Mixin that will define all needed sqlalchemy fiels for your model.

class flask_constance.backends.fsqla.**SettingMixin**

Bases: object

Model mixin for Flask-SQLAlchemy backend

id = Column(None, Integer(), table=None, primary_key=True, nullable=False)

name = Column(None, String(length=256), table=None, nullable=False)

value = Column(None, JSON(), table=None)

4.5 Signals

flask_constance.signals.**constance_setup**

Signal that called after extension was initialized. Called with 2 agruments:

- Instance of *Constance*.
- Instance of Flask application.

flask_constance.signals.**constance_get**

Signal that called after setting value was accessed. Called with 2 arguments:

- Instance of *Constance*.
- Name of the setting that was accessed.

flask_constance.signals.**constance_set**

Signal that called after setting value was updated. Called with 3 arguments:

- Instance of *Constance*.
- Name of the setting that was updated.
- New value of the setting.

4.6 RESTlike view

View implementing RESTlike interface for managing Flask-Constance dynamic settings.

class flask_constance.view.**ConstanceView**

Bases: MethodView

Method view for managing dynamic settings.

delete(*name*)

Delete(actually reset) setting value.

get(*name=None*)

Get specific setting or all settings.

methods: ClassVar[Optional[Collection[str]]] = {'DELETE', 'GET', 'PUT'}

The methods this view is registered for. Uses the same default (["GET", "HEAD", "OPTIONS"]) as route and add_url_rule by default.

put(*name*)

Update value for the setting.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

V