Flask-Storm Documentation

Release 0.1.2

Andreas Runfalk

Sep 18, 2017

Contents

Flask-Storm is an extension for Flask that adds support for Canonical's ORM Storm to your application. Flask-Storm automatically opens and closes database connections on demand when requests need them.

CHAPTER 1

Installation

```
$ pip install flask_storm
```

CHAPTER 2

Example

Access to the database is done using the *store* application context local. Within an application context this variable holds a reference to a Storm Store instance. If no connection is opened it will automatically open one. When the application context is torn down, normally after the request has returned, the store is closed.

```
from flask_storm import store
from storm.locals import Int, Unicode

class User(object):
    __storm_table__ = "users"

    id = Int(primary=True)
    name = Unicode()


@app.route("/")
def index():
    # Get name of user with ID 1
    return store.get(User, 1).name
```

CHAPTER 3

Index

Documentation

asdfasdfasfdf

Quickstart

This will be a quick example of how to use Flask-Storm to create a REST API endpoint for a list of messages. For the full example code, see *Full example.py*.

Imports

```
# example.py
from flask import Flask, jsonify
from flask_storm import FlaskStorm, store
from random import choice
from storm.locals import Int, Unicode
```

To get a minimal application running there are a few needed imports. The only noteworthy thing here is the `store` context local. For a primer on context locals see the flask documentation. This variable is a bit magic, and works just like the built-in g.

Application setup

```
app = Flask("example")
app.config["STORM_DATABASE_URI"] = "sqlite:///test.db"

flask_storm = FlaskStorm()
flask_storm.init_app(app)
```

Flask-Storm needs the `STORM_DATABASE_URI` to know which database to connect to. The format is described in the official documentation for Storm. It does not matter when the configuration variable is set, as long as it is done before using the *store* context local. Flask-Storm is bound to a Flask application by using *init_app()*. One instane of *FlaskStorm* can be bound to multiple applications at once. One application can however only be bound to a single *FlaskStorm* instance.

Declaring a model

```
class Post(object):
    __storm_table__ = "posts"

    id = Int(primary=True)
    name = Unicode()
    text = Unicode()

    def __init__(self, name=None, text=None):
        if name is not None:
            self.name = name

        if text is not None:
            self.text = text
```

This is how a model is declared in Storm. In this case a Post has an integer `id` column as primary key, and two Unicode text columns; `name` and `text`. The table `posts` is declared using a special dunderscore `__storm_table__`.

Initializing the database

```
@app.cli.command()
def initdb():
    """Create schema and fill database with 15 sample posts by random authors"""

    store.execute("""DROP TABLE IF EXISTS posts""")
    store.execute("""
        CREATE TABLE posts(
            id    INTEGER PRIMARY KEY,
            name  VARCHAR,
            text  VARCHAR
        )
    """)

    names = [
        u"Alice",
        u"Bob",
        u"Eve",
    ]

    for i in range(1, 16):
        store.add(Post(choice(names), u"Post #{}".format(i)))

    store.commit()
```

Starting with Flask version 0.11 there is a default command line interface that one can hook into. To create all tables and fill them with sample data, it is just a matter of running:

```
FLASK_APP=example.py flask initdb
```

The command will create `test.db` with the schema and data. This is the first use of an actual connection to the database. Note that there is no need to connect or close the store. This is handled automatically by Flask-Storm. A connection is never opened until the *`store`* context local is accessed, and remains open until the application context is torn down.

Serving requests

```
@app.route("/")
def index():
    """Return the 10 latest posts in JSON format"""

    return jsonify([{
        "id": post.id,
        "name": post.name,
        "text": post.text,
    } for post in store.find(Post).order_by(Desc(Post.id)).config(limit=10)])
```

The route serves a JSON array response containing the last 10 posts. In this case it is post 6 through 15.

Now it is just a matter of running the application. This is easily done using the Flask command line interface:

```
FLASK_APP=example.py flask run
```

This will, if there are no errors, serve the application on http://localhost:5000/. Open this page in a web-browser to see the JSON data.

Setup and tear down procedure

To prevent unnecessary overhead, database connections are created on demand when used within the application context. The same connection gets reused, and remains open, until the application context is torn down.

This means the only thing required to use the *`store`* context local is a configured application context.

Configuration options

This is the full list of configuration options for Flask Storm.

`STORM_DATABASE_URI` URI for the default database to connect to. This has the same format as the argument to Storm's `create_database` as defined in the official documentation.

`STORM_BINDS` A dictionary of Storm URIs that Flask Storm can connect to. A bind is defined as an arbitrary key, used to identify the bind, and a URI for the database. See *Using with multiple Stores* for an in-depth explaination.

Using with Flask CLI

When using `flask shell`, Flask Storm will automatically provide a refence to the *`store`* context local. Flask Storm also sets up debug output of the SQL statements created by Storm. This makes `flask shell` a good testing environment for building complex queries with Storm.

To make things more convenient it is recommended to provide model objects directly to the shell context. This is done easily by adding them using a shell context processor.

```
from flask import Flask
from storm.locals import Int, Unicode

class User(object):
    __storm_table__ = "users"

    id = Int(primary=True)
    name = Unicode()

app = Flask("example")

@app.shell_context_processor
def shell_context():
    return {"User": User}
```

This example makes automatically makes `User` available in the shell environment.

Note: It is possible to disable SQL statement printing by calling stop on the tracer.

```
>>> _storm_tracer.stop()
```

To disable color printing, reset the `fancy` flag:

```
>>> _storm_tracer.fancy = False
```

Using with multiple Stores

To interface with multiple Stores simultaneously binds exist. Apart from the default database, declared in `STORM_DATABASE_URI`, an arbitrary number of extra databases can be declared in `STORM_BINDS`. A bind declaration may look something like this:

```
STORM_BINDS = {
    "extra": "sqlite://:memory:",
}
```

The key `extra` is used to reference the bind, and the URI is used when connecting to the database. To make binds as easy to use as the normal *`store`* context local it is possible to create context locals for every bind, using *`create_context_local()`*.

```
# Use the same key as when declaring the bind. In this case "extra"
extra_store = create_context_local("extra")
```

`extra_store` can now be used just like *`store`* as long as an application context is available. Just like the default store, all binds are automatically closed on application context teardown.

Tip: Declare extra bind context locals in a separate Python file that can be imported.

Full example.py

```python
# example.py
from flask import Flask, jsonify
from flask_storm import FlaskStorm, store
from random import choice
from storm.locals import Int, Unicode


app = Flask("example")
app.config["STORM_DATABASE_URI"] = "sqlite:///test.db"

flask_storm = FlaskStorm()
flask_storm.init_app(app)


class Post(object):
    __storm_table__ = "posts"

    id = Int(primary=True)
    name = Unicode()
    text = Unicode()

    def __init__(self, name=None, text=None):
        if name is not None:
            self.name = name

        if text is not None:
            self.text = text


@app.cli.command()
def initdb():
    """Create schema and fill database with 15 sample posts by random authors"""

    store.execute("""DROP TABLE IF EXISTS posts""")
    store.execute("""
        CREATE TABLE posts(
            id    INTEGER PRIMARY KEY,
            name  VARCHAR,
            text  VARCHAR
        )
    """)

    names = [
        u"Alice",
        u"Bob",
        u"Eve",
    ]

    for i in range(1, 16):
        store.add(Post(choice(names), u"Post #{}".format(i)))

    store.commit()


@app.route("/")
def index():
    """Return the 10 latest posts in JSON format"""

    return jsonify([{
        "id": post.id,
        "name": post.name,
        "text": post.text,
    } for post in store.find(Post).order_by(Desc(Post.id)).config(limit=10)])
```

API documentation

The following functions and classes are part of Flask Storm's public API. All are available directly from the top level namespace `flask_storm`. Imports from namespaces below `flask_storm`, such as `flask_storm.debug` is not supported and may change across versions.

Extension

class `FlaskStorm`(*app=None*)

Create a FlaskStorm instance.

Parameters **`app`** – Application to enable Flask-Storm for. This is the same as calling `init_app()` after initialization.

`connect`(*bind=None*)

Return a new Store instance with a connection to the database specified in the STORM_DATABASE_URI configuration variable of the bound application. This method normally not be called externally as it does not close the store once the application context tears down.

Parameters **`bind`** – Database URI to use, defaults to `STORM_DATABASE_URI`. See *`get_binds()`* for details.

Returns Store instance

Raises **`RuntimeError`** – if no connection URI is found.

`get_binds`()

Return dict of database URIs for the application as defined by the `STORM_BINDS` configuration variable. If `STORM_DATABASE_URI` is defined it will be available using the key `None`.

Returns Dict from bind names to database URIs.

`get_store`(*bind=None*)

Return a Store instance for the current application context. If there is no instance a new one will be created. Instances created using this method will close on application context tear down.

Parameters **`bind`** – Bind name of database URI. Defaults to the one specified by `STORM_DATABASE_URI`.

Returns Store for the current application context.

Raises **`RuntimeError`** – if accessed outside the scope of an application context.

`init_app`(*app*)

Binds this extension to the given application. This is important since the database connection will not close unless the application has been initialized.

Note: One instance of FlaskStorm may be registered to multiple applications. One application should however only be registered to one FlaskStorm instance, or tear down functionality will trigger multiple times.

Parameters **`app`** – Application to enable Flask-Storm for.

Raises **`RuntimeError`** – if an application is already bound to this instance by being passed to the constructor.

`store`

Return a Store instance for the current application context. If there is no instance a new one will be created. Instances created using this method will close on application context tear down.

Parameters **`bind`** – Bind name of database URI. Defaults to the one specified by `STORM_DATABASE_URI`.

Returns Store for the current application context.

Raises **`RuntimeError`** – if accessed outside the scope of an application context.

Context locals

`store` **= <LocalProxy unbound>**

Shorthand for *`FlaskStorm.store`* which does not depend on knowing the FlaskStorm instance bound to the current request context. This is the prefered method of accessing the Store, since using the *`FlaskStorm.store`* property directly makes it easy to accidentally create circular imports.

`create_context_local`(*bind*)

Create a context local for the given bind. None means the default store.

```
# Context local for the bind report_store
report_store = create_context_local("report_store")
```

This is what is used to implement the store context local which analogue to

```
store = create_context_local(None)
```

Parameters **`bind`** – Bind name of database URI to use when creating store.

Raises **`RuntimeError`** – if working outside application context or if FlaskStorm is not bound to the current application.

Tracers

Tracers provide facilities to intercept and read SQL statements generated by Storm. The tracers provided in Flask-Storm are tailored for use in multithreaded environments, and work in conjunction with Flask's contexts.

DebugTracer

class **`DebugTracer`**

A tracer which stores all queries with parameters onto the current request context. Queries are accessible using *`get_debug_queries()`*. It is used as a context manager. Since all queries are stored on the request context, they can only be accessed until the current request tears down.

```
with DebugTracer():
    # Perform queries
    queries = get_debug_queries()
```

Note: *get_debug_queries()* do not need to be called within the context manager, as long as the request context is still alive, since all queries are stored on the request context.

get_debug_queries()

Return an array of queries executed within the context of a *DebugTracer* under the current application context and thread.

RequestTracer

class **RequestTracer**(*file=None*, *fancy=None*)

A tracer which prints all SQL queries generated by Storm directly to STDOUT. This is useful when debugging views in flask.

```
with RequestTracer():
    # Queries executed here will be printed into STDOUT
    ...
```

Parameters

- **file** – File like object (has write method) where queries will be logged to.
- **fancy** – When `True` (default) colored output is used, if support is detected, when `False` plain text is used.

start()

Install and activate this tracer for all statements executed in this thread (or greenlet).

stop()

Stop using this tracer.

Utility

find_flask_storm(*app*)

Find and return *FlaskStorm* instance for the given application.

Parameters **app** – Application to look for FlaskStorm instance in.

Returns FlaskStorm instance if found, else None.

Changelog

Version are structured like the following: `<major>.<minor>.<bugfix>`. Unless explicitly stated, changes are made by Andreas Runfalk.

Version 0.1.2

Released on 14th June 2017

- Fixed an issue with query logging in `flask shell` and PostgreSQL

Version 0.1.1

Released on 9th June 2017

- Fixed issue with new versions of sqlparse by bumping its version requirement

Version 0.1.0

Released on 19 July 2016

- Initial release

Python Module Index

Index