Flask-Orator Documentation

Release 0.2.0

Sébastien Eustace

October 06, 2016

Contents

Flask-Orator adds Orator ORM support to Flask applications.

Since it is merely a wrapper for Orator, it has all its benefits:

- A simple but powerful ORM
- A database agnostic Schema Builder
- A low level Query Builder to avoid the overhead of the ORM
- Migrations
- Support for **PostgreSQL**, **MySQL** and **SQLite** out of the box

Flask-Orator supports python versions **2.7+** and **3.2+**

CHAPTER 1

Installation

You can install Flask-Orator in 2 different ways:

- The easier and more straightforward is to use pip

```
pip install flask-orator
```

- Install from source using the official repository (https://github.com/sdispater/flask-orator)

Note: The different dbapi packages are not part of the package dependencies, so you must install them in order to connect to corresponding databases:

- PostgreSQL: `psycopg2`
- MySQL: `PyMySQL` or `mysqlclient`
- SQLite: The `sqlite3` module is bundled with Python by default

CHAPTER 2

Basic Usage

2.1 A Minimal Application

Note: This example application will not go into details as to how the ORM works. You can refer to the Orator documentation for more information.

Setting up Flask-Orator for a single Flask application is quite simple. Create your application, load its configuration and then create an `Orator` object.

The `Orator` object behaves like a `DatabaseManager` instance set up to work flawlessly with Flask.

```
from flask import Flask
from flask_orator import Orator

app = Flask(__name__)
app.config['ORATOR_DATABASES'] = {
    'development': {
        'driver': 'sqlite',
        'database': '/tmp/test.db'
    }
}

db = Orator(app)


class User(db.Model):

    __fillable__ = ['name', 'email']

    def __repr__(self):
        return '<User %r>' % self.name
```

Now, you need to create the database and the `users` table using the embedded CLI application. Let's create a file named `db.py` which has the following content:

```
from your_application import db

if __name__ == '__main__':
    db.cli.run()
```

This file, when executed, gives you access to useful commands to manage you databases.

Note: For the exhaustive list of commands see the *CLI* section.

You first need to make a migration file to create the table:

```
python db.py make:migration create_users_table --table users --create
```

This will add a file in the `migrations` folder named `create_users_table` and prefixed by a timestamp:

```
from orator.migrations import Migration


class CreateTableUsers(Migration):

    def up(self):
        """
        Run the migrations.
        """
        with self.schema.create('users') as table:
            table.increments('id')
            table.timestamps()

    def down(self):
        """
        Revert the migrations.
        """
        self.schema.drop('users')
```

You need to modify this file to add the `name` and `email` columns:

```
with self.schema.create('users') as table:
    table.increments('id')
    table.string('name').unique()
    table.string('email').unique()
    table.timestamps()
```

Then, you can run the migration:

```
python db.py migrate
```

Confirm and you database and the table will be created.

Once your database set up, you can create some users:

```
from your_application import User

admin = User.create(name='admin', email='admin@example.com')
guest = Guest.create(name='guest', email='guest@example.com')
```

The `create()` method will create the users instantly. But you can also initiate them and save them later:

```
admin = User(name='admin', email='admin@example.com')
# Do something else...
admin.save()
```

Note: Optionally you can use a transaction.

```
from your_application import db, User
```

```
with db.transaction():
    admin = User.create(name='admin', email='admin@example.com')
    guest = Guest.create(name='guest', email='guest@example.com')
```

You can now retrieve them easily from the database:

```
users = User.all()

admin = User.where('name', 'admin').first()
```

2.2 Relationships

Setting up relationships between tables is a breeze. Let's create a `Post` model with the `User` model as a parent:

```
from orator.orm import belongs_to


class Post(db.Model):

    __fillable__ = ['title', 'content']

    @belongs_to
    def user(self):
        return User
```

And we add the `posts` relationship to the `User` model:

```
from orator.orm import has_many


class User(db.Model):

    @has_many
    def posts(self):
        return Post
```

Before we can play with these models we need to create the `posts` table and set up the relationship at database level:

```
python db.py make:migration create_posts_table --table posts --create
```

And we modify the generated file to look like this:

```
from orator.migrations import Migration


class CreatePostsTable(Migration):

    def up(self):
        """
        Run the migrations.
        """
        with self.schema.create('posts') as table:
            table.increments('id')
            table.string('title')
            table.text('content')
            table.integer('user_id', unsigned=True)
```

```
            table.timestamps()

            table.foreign('user_id').references('id').on('users')

    def down(self):
        """
        Revert the migrations.
        """
        self.schema.drop('posts')
```

Finally we run it:

```
python db.py migrate
```

We can now instantiate some posts:

```
admin_post = Post(title='Admin Post',
                  description='This is a restricted post')

guest_post = Post(title='Guest Post',
                  description='This is a guest post')
```

and associate them with users:

```
# Associate from user.posts relation
admin.posts().save(admin_post)

# Associate from post.user relation
guest_post.user().associate(guest)
```

Note: You can also create the posts directly.

```
admin.posts().create(
    title='Admin Post',
    description='This is a restricted post'
)
```

Relationships properties are dynamic properties meaning that `user.posts` is the underlying collection of posts so we can do things like:

```
user.posts.first()
user.posts[2:7]
user.posts.is_empty()
```

But, if we need to retrieve a more fine-grained portion of posts we can actually to so:

```
user.posts().where('title', 'like', '%admin%').get()
user.posts().first()
```

2.3 Pagination

Flask-Orator supports pagination:

```
users = User.paginate(15)
```

This will retrieve `15` users. The current page is determined by default by the `?page` query string parameter of the request.

This behavior can be modified if needed, either by explicitely specifying the current page:

```
users = User.paginate(15, request.args['index'])
```

or by changing the default `Paginator` current page resolver:

```
from flask import request
from orator import Paginator

def current_page_resolver():
    return request.args.get('index', 1)

Paginator.current_page_resolver(current_page_resolver)
```

2.4 What's more?

Like said in the introduction Flask-Orator is a wrapper around Orator to integrate it more easily with Flask applications. So, basically, everything you can do with Orator is also possible with Flask-Orator.

Referer to the Orator documentation to see the features available.

CHAPTER 3

CLI

The following examples assume that a file named `db.py` has been created with the following content:

```
from your_application import db

if __name__ == '__main__':
    db.cli.run()
```

3.1 Migrations

3.1.1 Creating Migrations

To create a migration, you can use the `make:migration` command on the CLI:

```
python db.py make:migration create_users_table
```

This will create a migration file that looks like this:

```
from orator.migrations import Migration


class CreateTableUsers(Migration):

    def up(self):
        """
        Run the migrations.
        """
        pass

    def down(self):
        """
        Revert the migrations.
        """
        pass
```

By default, the migration will be placed in a `migrations` folder relative to where the command has been executed, and will contain a timestamp which allows the framework to determine the order of the migrations.

If you want the migrations to be stored in another folder, use the `--path/-p` option:

```
python db.py make:migration create_users_table -p my/path/to/migrations
```

The `--table` and `--create` options can also be used to indicate the name of the table, and whether the migration will be creating a new table:

```
python db.py make:migration add_votes_to_users_table --table=users

python db.py make:migration create_users_table --table=users --create
```

These commands would respectively create the following migrations:

```
from orator.migrations import Migration


class AddVotesToUsersTable(Migration):

    def up(self):
        """
        Run the migrations.
        """
        with self.schema.table('users') as table:
            pass

    def down(self):
        """
        Revert the migrations.
        """
        with self.schema.table('users') as table:
            pass
```

```
from orator.migrations import Migration


class CreateTableUsers(Migration):

    def up(self):
        """
        Run the migrations.
        """
        with self.schema.create('users') as table:
            table.increments('id')
            table.timestamps()

    def down(self):
        """
        Revert the migrations.
        """
        self.schema.drop('users')
```

3.1.2 Running Migrations

To run all outstanding migrations, just use the `migrate` command:

```
python db.py migrate
```

3.1.3 Rolling back migrations

Rollback the last migration operation

```
python db.py migrate:rollback
```

Rollback all migrations

```
python db.py migrate:reset
```

3.1.4 Getting migrations status

To see the status of the migrations, just use the `migrations:status` command:

```
python db.py migrate:status
```

This would output something like this:

```
+--------------------------------------------------+------+
| Migration                                        | Ran? |
+--------------------------------------------------+------+
| 2015_05_02_04371430559457_create_users_table     | Yes  |
| 2015_05_04_02361430725012_add_votes_to_users_table | No   |
+--------------------------------------------------+------+
```