Python Business Logic Documentation

Release 0.2.0

Jakub Skałecki

Sep 16, 2018

Contents

Contents:

CHAPTER 1

Python Business Logic

Traditionally, most web applications are written using MVC pattern. Python Business Logic helps you to add *Business Layer*, also called *Application Layer*, that is dependent only on models and composed of simple functions. Code written this way is extremely easy to read, test, and use in different scenarios. Package has no dependencies and can be used in any web framework, like Django, Flask, Bottle and others.

1.1 Documentation

The full documentation is at https://python-business-logic.readthedocs.io. Still under development.

1.2 Installation

Install Python Business Logic:

```
pip install python-business-logic
```

1.3 Getting Started

Core elements of library are validators, functions created to ensure that business logic is correct:

```
>>> from business_logic.core import validator

>>> @validator
... def can_watch_movie(user, movie):
...     # some example business logic, it can be as complex as you want
...     return user.is_parent or user.age >= movie.age_restriction
```

With validators you can decorate actions performed that will be checked against that validator:

```
>>> from business_logic.core import validated_by

>>> @validated_by(can_watch_movie)
... def watch_movie(user, movie):
...     print("'{}' is watching movie '{}'".format(user.name, movie.name))
```

As you can see, arguments to validator must match those passed to function. Now every call to `watch_movie` will require that validator `can_watch_movie` passes:

```
>>> import collections
>>> User = collections.namedtuple('User', ['name', 'age', 'is_parent'])
>>> Movie = collections.namedtuple('Movie', ['name', 'age_restriction'])
>>> alice = User('Alice', 32, True)
>>> bob = User('Bob', 6, False)
>>> cartoon = Movie('Tom&Jerry', 0)
>>> horror = Movie('Scream', 18)

>>> watch_movie(bob, cartoon)
'Bob' is watching movie 'Tom&Jerry'

>>> watch_movie(alice, horror)
'Alice' is watching movie 'Scream'

>>> watch_movie(bob, horror)
Traceback (most recent call last):
  File "business_logic/core.py", line 48, in wrapper
    raise ServiceException("Validation failed!")
business_logic.exceptions.LogicException: Validation failed!
```

You can skip validation using `validate=False`:

```
>>> watch_movie(user=bob, movie=horror, validate=False)
'Bob' is watching movie 'Scream'
```

Also, if we just want to know if action is permitted, just let's run:

```
>>> validation = can_watch_movie(bob, horror, raise_exception=False)
>>> validation
<PermissionResult success=False error=Validation failed!>

>>> bool(validation)
False

>>> validation.error  # it's an actual exception
LogicException('Validation failed!', error_code=None)
```

Chaining validators is really easy and readable:

```
>>> @validator
... def is_old_enough(user, movie):
...     return user.age >= movie.age_restriction

>>> @validator
... def can_watch_movie(user, movie):
...     is_old_enough(user, movie)
...     # we don't have to return anything, @validator makes use of exceptions
```

(continues on next page)

(continued from previous page)

```
>>> can_watch_movie(bob, horror)
Traceback (most recent call last):
   File "business_logic/core.py", line 48, in wrapper
     raise LogicException("Validation failed!")
business_logic.exceptions.LogicException: Validation failed!
```

Ok, but we're still missing something. We don't know why exactly validation failed, all we have is a generic "Validation failed!" message. How to fix that? It's easy, let's make our own errors!

```
>>> from business_logic import LogicErrors, LogicException
>>> class AgeRestrictionErrors(LogicErrors):
...     CANT_WATCH_MOVIE_TOO_YOUNG = LogicException("User is too young to watch this")

>>> @validator
... def is_old_enough(user, movie):
...     if user.age < movie.age_restriction:
...         raise AgeRestrictionErrors.CANT_WATCH_MOVIE_TOO_YOUNG

>>> is_old_enough(bob, horror)
Traceback (most recent call last):
business_logic.exceptions.LogicException: User is too young to watch this

>>> # we can also obtain exception details like this
>>> result = is_old_enough(bob, horror, raise_exception=False)
>>> bool(result)
False

>>> result.error
LogicException('User is too young to watch this', error_code='CANT_WATCH_MOVIE_TOO_
↪YOUNG')

>>> result.error_code == 'CANT_WATCH_MOVIE_TOO_YOUNG'
True

>>> # result.errors is shortcut to registry with all errors
>>> result.error == result.errors['CANT_WATCH_MOVIE_TOO_YOUNG']
True
```

Testing is really easy:

```
>>> def test_user_cant_watch_movie_if_under_age_restriction():
...     bob = User('Bob', 6, False)
...     horror = Movie('Scream', 18)
...     result = is_old_enough(bob, horror, raise_exception=False)
...     # There are two ways to check if expected exceptions was raised
...     assert result.error_code == 'CANT_WATCH_MOVIE_TOO_YOUNG'
...     assert result.error == AgeRestrictionErrors.CANT_WATCH_MOVIE_TOO_YOUNG

>>> test_user_cant_watch_movie_if_under_age_restriction()
```

Also, if you need to display parametrizable error messages, just use *.format* method

1.4 Usage

When using this package, you should write all your business logic as simple functions, using only inputs and Database Layer (eg. *Django ORM or SQLAlchemy*). This way, you can easily test your logic and use it in any way you like. Convention that I follow is to put all functions inside *logic.py* file or *logic* submodule.

In **views** and **API** calls: Your role is to prepare all required data for business function (from forms, user session etc), call function and present results to user. Middleware catching LogicException and, for example, displaying message to user in a generic way can improve readability a lot, because no exception handling need to be done in view.

As **management commands**: In Django you can create custom *management command*, that allows you to use cli to perform custom logic. Python Business Logic functions works very well for that use case!

From **external code**: Just import your function and use it. Since there shouldn't be any framework-related inputs other than Database Models, usage is really simple. In reality, your business functions form *business API* of your application.

1.5 Examples

For examples how to use this library, look into directory *examples*. Currently there is only one called *Football match*. Most important file there is `logic.py`.

1.6 Running Tests

Does the code actually work?

```
$ pip install -r requirements_test.txt
$ tox
```

CHAPTER 2

Usage

CHAPTER 3

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.1 Types of Contributions

3.1.1 Report Bugs

Report bugs at https://github.com/Valian/python-business-logic/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

3.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

3.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

3.1.4 Write Documentation

Python Business Logic could always use more documentation, whether as part of the official Python Business Logic docs, in docstrings, or even on the web in blog posts, articles, and such.

3.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/Valian/python-business-logic/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.2 Get Started!

Ready to contribute? Here's how to set up *python-business-logic* for local development.

1. Fork the *python-business-logic* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/python-business-logic.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv python-business-logic
$ cd python-business-logic/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 business_logic tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/Valian/python-business-logic/pull_requests and make sure that the tests pass for all supported Python versions.

3.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_business_logic
```

CHAPTER 4

Credits

4.1 Development Lead

- Jakub Skałecki <jakub.skalecki@gmail.com>

4.2 Contributors

- Marek Siarkowicz
- Filip Figiel

CHAPTER 5

History

5.1 0.2.0 (2017-10-22)

- Support for Python 3.7
- Parametrizable errors

5.2 0.1.1 (2017-10-22)

- Fixed encoding in python 2.7.

5.3 0.1.0 (2017-07-16)

- First release on PyPI.