lifter Documentation

Release 0.4.1

Eliot Berriot

November 13, 2016

Contents

Lifter is a lightweight query engine for Python iterables, inspired by Django ORM.

If you find if painful to gather data from collections of Python objects or dictionaries, such as API results, lifter is for you!

> **Warning:** This package is still in alpha state and a lot of work is still needed to make queries faster and efficient. Contributions are welcome :)

Contents:

CHAPTER 1

Overview

Lifter is a package that heavily inspires from Django's ORM in the way it works. Some parts of the API are also quite similar to TinyDB and SQLAlchemy.

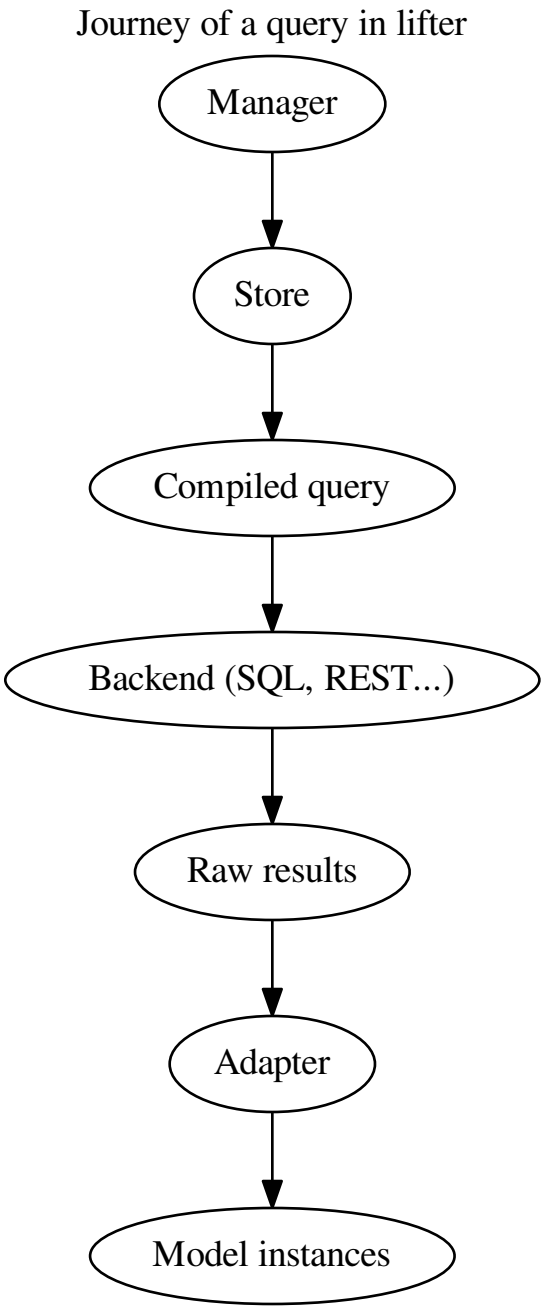
However, lifter intends to work with any data provider, meaning you can query data from various sources, using the same high-level API.

If implemented correctly, you can for exemple work on a project with static JSON data from a file during development, and easily switch to a real REST API when it's time.

To achieve this, lifter tries to be as agnostic and flexible as possible regarding data sources, while adressing most common use cases.

1.1 The big picture

Lifter is made of several layers, each one fulfilling a specific task.



1.1.1 Models

Just like Django, lifter models are Python classes representing your data. You could have a `User` model and a `Group` model.

1.1.2 Stores

Stores are responsible for parsing queries about a model, send them to an underlying backend and return proper results to lifter.

1.1.3 Adapters

Because we don't want to deal with raw data such as SQL or JSON Responses, adapters are reponsible for converting data returned by our refined stores to actual model instances.

1.1.4 Managers

Managers are the main entrypoint in lifter to issue queries on our data stores.

1.1.5 Querysets

Just like in Django, querysets provide a high-level API to query, exclude, filter results from our stores. Internally, querysets build query objects, that are then interpreted by the refined store.

1.1.6 Backends

We try to keep lifter's code as agnostic and generic as possible.

Because of that, logic that is specific to a data source should be stored in a dedicated module, such modules being named `backends`.

Lifter comes build-in with a few backends, the most advanced being the Python backend. However, work is in progress to implement file and HTTP backends.

1.2 The query language

At the moment, lifter support two ways two make queries:

1. the explicit engine, which looks like SQLAlchemy. Example: *manager.filter(User.age == 42)*. **This is the recommended engine for any new project**
2. the keyword engine, which looks like Django. Example: *manager.filter(age=42)*. This is the engine that was built-in in the first release of the package.

Internally, lifter converts all queries to the explicit engine. This mean if you do *manager.filter(age=42)*, lifter will convert it to *manager.filter(User.age == 42)*.

> **Warning:** We recommend using the explicit engine for any new projects involving lifter. The keyword engine will be kept for backward compatibility until and if a deprecation plan is adopted.

CHAPTER 2

Installation

At the command line:

```
$ easy_install lifter
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv lifter
$ pip install lifter
```

That's it, you can continue to the next section!

CHAPTER 3

Quickstart

Using lifter consists only of a few steps.

3.1 Creating a model

This model will represent the data you are querying:

```
import lifter.models

class User(lifter.models.Model):
    pass
```

3.2 Creating a data store

In this example, we'll assume the data you want to query is available in the python process, as an iterable. Therefore, we use the Python store:

```
from lifter.backends.python import IterableStore

data = [
    {
        'age': 27,
        'is_active': False,
        'email': 'kurt@cobain.music',
    },
    {
        'age': 687,
        'is_active': True,
        'email': 'legolas@deepforest.org',
    },
    {
        'age': 34,
        'is_active': False,
        'email': 'golgoth@lahorde.org',
    }
]

store = IterableStore(data)
```

In this step, we made our data available in lifter, so it can be queried later.

3.3 Creating a manager

Creating a manager is the next step:

```
manager = store.query(User)
```

With the previous line, we told lifter that our store will return `User` instances.

3.4 Your first query

Enough set up, let's run your first query:

```
young_users = manager.filter(User.age < 30)
```

The previous query will return a queryset containing all users matching `age < 30`.

You could then iterate over those results and use them in your application:

```
for user in young_users:
    print('Hello {0}!'.format(user['email']))
```

3.5 More complex queries

Of course, we just scratched the surface here. More complex and useful methods are available in lifter:

```
# get the total number of inactive users
total_inactive = manager.filter(User.is_active == False).count()

# get only legolas' data
legolas = manager.get(User.email == 'legolas@deepforest.org')

# get a list containing all users emails
emails_list = manager.all().values_list('email', flat=True)

# get total and average age of our users
from lifter import aggregates

manager.all().aggregate(
    aggregates.Sum('age'),
    aggregates.Avg('age'),
)
>>> {'age__avg': 249.33, 'age__sum': 748}
```

If you're interested, head over Query API!

CHAPTER 4

Query API

At the moment, lifter is read-only, meaning no write query can be issued to a store.

`QuerySet` is the class used for read-related queries.

4.1 Paths

`Paths` refers to the fields of a model involved in a query. For example, take the following queryset:

```
import lifter.models

class User(lifter.models.Model):
    pass

manager.filter(User.age > 13, User.is_active == True)
```

In the previous example, two paths are used in the query: `User.age` and `User.is_active`.

Thanks to a little magic, these paths are actually represented as plain python objects:

```
User.age
>>> <Path: age>
```

Paths can actually be nested. If you have the following data source:

```
users = [
    {
        'id': 1,
        'name': 'Jeff Winger',
        'company': {
            'id': 3,
            'name': 'Greendale',
        }
    }
]
```

You can totally create paths such as `User.company.id` or `User.company.name`.

4.2 Query nodes

Paths are used to create query nodes. A `QueryNode` is a Python object, used to represent a condition:

```
path = User.age
qn = path < 30
```

Of course, in real life, you'll use the shorter (and also more readable notation):

```
User.age < 30
>>> <QueryNode age, <built-in function lt>, [30], {}>
```

In the previous example, we have created a query node representing the condition `age < 30`. We can now use this node to fetch data:

```
qn = User.age < 30
manager.all().filter(qn)
>>> Returns all user matching the condition
```

4.2.1 Combining nodes

It is possible to combine nodes together using python bitwise operators to build more complex queries:

```
qn = (User.age < 30) & (User.is_active == True)
>>> a node matching User.age < 30 AND User.is_active == True

qn = (User.age < 30) | (User.is_active == True)
>>> a node matching User.age < 30 OR User.is_active == True

# use ~ to invert a query node
qn = ~(User.age < 30)
```

4.3 Queries

Queries are higher-level objects that describe an action to run on the data store:

```
import lifter.query

qn = (User.age < 30) & (User.is_active == False)
query = lifter.query.Query(action='select', filters=qn)
```

4.4 QuerySets

Don't worry, you won't have to instanciate all of these objects by hand to use lifter.

QuerySets are here to provide the high-level API for interacting with data stores.

Once you have a manager instance, issuing query is done easily with querysets:

```
import lifter.models
from lifter.backends.python import IterableStore

class User(lifter.models.Model):
    pass

data = [
    {
```

```
        'age': 27,
        'is_active': False,
        'email': 'kurt@cobain.music',
    },
    {
        'age': 687,
        'is_active': True,
        'email': 'legolas@deepforest.org',
    },
    {
        'age': 34,
        'is_active': False,
        'email': 'golgoth@lahorde.org',
    }
]

store = IterableStore(data)
manager = store.query(User)

# Here you pass query nodes directly to the queryset to obtain results from the store
manager.filter(User.age < 30)
```

4.4.1 QuerySet methods

filter(**explicit_queries*, ***keyword_queries*)

Return a set of objects that match one or multiple queries.

Simple example using one query:

```
# return all 42 years-old users
manager.filter(User.age == 42)
```

Providing multiple queries to this method will merge all of them using AND operator:

```
manager.filter(User.age >= 42, User.age <= 56)
```

The previous example will return only objects that match *both* queries.

This is equivalent of writing:

```
manager.filter((User.age >= 42) & (User.age <= 56))
```

exclude(**explicit_queries*, ***keyword_queries*)

This method is the exact opposite of `filter()`. It will return objects that do *not* match the provided queries:

```
# Exclude inactive users
manager.exclude(User.is_active == False)

# Exclude only inactive users that are 42 years-old
manager.exclude(User.is_active == False, User.age == 42)
```

Providing multiple queries to this method will merge all of them using AND operator:

This is equivalent of writing:

```
manager.exclude((User.age >= 42) & (User.age <= 56))
```

get(**explicit_queries*, ***keyword_queries*)

This method retrieve a single object that match all of the given queries:

```
kurt = manager.get(User.id == 447)
```

Get will raise `lifter.exceptions.DoesNotExist` if no object is found, and `lifter.exceptions.MultipleObjectsReturned` if multiple objects are found:

```
import lifter.exceptions

try:
    kurt = manager.get(User.first_name == 'Kurt')
except lifter.exceptions.DoesNotExist:
    print('Sorry, no user found, try something else')
except lifter.exceptions.MultipleObjectsReturned:
    print('Multiple users are named Kurt, please precise your query')
```

This method will retrieve the final object among the queryset values:

```
>>> manager.get(User.first_name == 'Kurt')
# Retrieve among all manager loaded objects
>>> manager.filter(User.age == 42).get(User.first_name == 'Kurt')
# Retrieve among 42 years-old users
```

order_by(**paths*)

Order the queryset results using the provided attribute(s):

```
>>> manager.order_by(User.age)
# Returns a queryset of users, from younger to older
```

You can reverse the ordering using python invert operator:

```
>>> manager.order_by(~User.age)
# Returns a queryset of users, from older to younger, this time
```

It's possible to sort using multiple paths:

```
>>> manager.order_by(User.is_active, User.age)
# Sort by is_active then by age
```

Finally, you can also use random sorting, by passing a question mark instead of a path:

```
>>> manager.order_by('?')
# Random order
>>> manager.order_by(User.age, '?')
# Sort by age then randomly
```

values(**paths*)

Use this method if you only want to retrieve specific values from your object list, instead of the objects themselves. It will return a list of dictionaries, with the requested values as keys:

```
>>> manager.values(User.age, User.email)
[{'age': 36, 'email': 'benard@blackbooks.com'}, {'age': 33, 'email': 'manny@blackbooks.com'}]
```

values_list(**paths*, *flat=False*)

This method works as *values()*, but instead of of list of dictionaries, it returns a list of tuples.

```
>>> manager.values_list(User.age, User.email)
[(36, 'benard@blackbooks.com'), (33, 'manny@blackbooks.com')]
```

If you're only requesting a single value and want a flat list (no tuples in it), you can set the *flat* parameter to True:

```
>>> manager.values_list(User.email, flat=True)
['benard@blackbooks.com', 'manny@blackbooks.com']
```

count()
A helper method that return the number of objects inside the queryset:

```
>>> manager.filter(User.age == 42).count()
56
```

You can achieve the same result using *len*:

```
qs = manager.filter(User.age == 42)
print(len(qs))
```

first()
A helper method that return the first object of the queryset or *None* if it's empty:

```
>>> manager.filter(User.age == 42).first()
<User object>
>>> manager.filter(User.age == 666).first()
None
```

last()
Works as *first()* but returns the last object of the queryset.

exists()
A helper method that return *True* if the queryset has at least one result, *False* otherwise:

```
>>> manager.filter(User.age == 666).exists()
False
```

distinct()
A method that remove duplicates from a queryset:

```
>>> manager.values_list(User.eye_color, flat=True)
['green', 'brown', 'green', 'red', 'brown', 'red']
>>> manager.values_list(User.eye_color, flat=True).distinct()
['green', 'brown', 'red']
```

aggregate(**aggregates*, ***named_aggregates*, *flat=False*)
Extract data from the queryset objects and return it as a dictionary.

A simple example to retrieve the average age of all users:

```
>>> import statistics
>>> manager.aggregate((User.age, statistics.mean))
{'age__mean': 44.2}
```

Under the hood, the previous example will loop on all loaded users, grab the *age* attribute, append the age to a list, then pass this list to the *mean* function and return the final result.

The method expect *(path, callable)* tuples as parameters. The path is the object attribute you want to gather, and the callable is the function that will return a value from the gathered data.

You can request multiple aggregates at once:

```
>>> manager.aggregate((User.age, statistics.mean), (User.age, min))
{'age__mean': 44.2, 'age__min': 12}
```

Bind them to specific keys:

```
>>> manager.aggregate(average_age=(User.age, statistics.mean))
{'average_age': 44.2}
```

And return aggregates as a list instead of a dictionary using the *flat* parameter:

```
>>> manager.aggregate((User.age, statistics.mean), (User.age, min), flat=True)
[44.2, 12]
```

4.4.2 Chaining querysets

Some of the previously described methods allow chaining You can chain querysets at will using *filter()* and/or *exclude()*:

```
manager.exclude(User.age == 34).filter(User.is_active == True).filter(User.has_beard == False)
```

The previous example tranlates to:

1. In all users, exclude then one where *age* equals 34
2. Then, from the previous queryset, keep only active users
3. Then, from the previous queryset, leave only users with no beard

4.4.3 Querysets are lazy

No matter how much time you chain *filter()* and/or *exclude()* calls, the final query will only be actually applied when you try to access the queryset data:

```
# This will be instant, even if your user list has 1,000,000,000 entries in it
queryset = manager.exclude(User.age == 16)

# however, calling one of the following will apply the filter
queryset.count()
for user in queryset:
    print(user.age)
```

Once a queryset is evaluated (when queries have been applied), results are stored internally, and the queryset can be looped has many times as you want at no cost.

CHAPTER 5

Backends

Lifter is bundled with a few backends, the main one being the Python backend, that operates on Python iterables.

However, you are not limited to querying Python iterables. Lifter is designed to be backend-agnostic. Therefore, it's simple enough to make lifter talk to other data sources, such as a SQL database, a REST API, a LDAP store, a file...

We we use the term backend, we mainly designate a dedicated `Store` implementation, because the store is responsible for compiling a generic query, sending it to the underlying data source and return results. A backend, though, could also include dedicated models, parsers or adapters.

Contents:

5.1 Python backend

5.1.1 Supported data structures

Lifter operates on collections of objects. A collection can be any Python iterable, like a list, a tuple, or even a generator. As long as lifter can iterate on it, everythings fine.

Such collections must contain similarly structured objects in order for lifter to work:

```
# okay
tags = [
    {'name': 'python', 'articles_count': 134},
    {'name': 'ruby', 'articles_count': 42},
    {'name': 'php', 'articles_count': 23},
]

# okay
class User(object):
    def __init__(self, age, first_name, last_name):
        self.age = age
        self.first_name = first_name
        self.last_name = last_name

users = [
    User(42, 'Douglas', 'Adams'),
    User(867, 'Legolas', 'The Elf'),
    User(98, 'Benjamin', 'Button'),
]
```

```
# not okay
users_and_tags = [tags[0], users[0], tags[1], users[1]]
```

If your iterable contains mappings (such as dictionaries), lifter will treat them as regular objects, and transparently access keys instead of attributes when filtering, retrieving and aggregating values.

CHAPTER 6

Caching

Lifter offers a built-in caching mechanism that can be used store query results and retrieve them later.

This is an efficient way to reduce the I/O of your application, avoid reaching rate limites, suffer from network latency, etc.

Caching is configured on store creation, via the following API:

```
from lifter import caches
from lifter import models
from lifter.backend import http

class MyModel(models.Model):
    class Meta:
        app_name = 'my_app'
        name = 'my_model'

cache = caches.DummyCache()
store = http.RESTStore(identifier='my_store', cache=cache)
manager = store.query(MyModel)
```

You can use the same *Cache* instance accross multiple store if you want, this won't lead to cache collisions.

6.1 How does it work?

When a cache is configured for a given store and the store execute a query, the following happens:

1. The store identifier, the model app, the model name and the query are hashed together to form a cache key
2. The cache is then queried using that key
3. If a result is found with that key, it's returned directly without sending the query to the underlying backend
4. If no result is found, the query is processed normally, but result will be stored in the cache for further use

Once a cache is configured for a store, it is automatically used:

```
# This will execute the query and store results in the cache
manager.all().count()

# For this one, the query won't execute, since the value is present in the cache
manager.all().count()
```

For the previous example, the cache key will look like:

```
my_store:my_app:my_model:ee26b0dd4af7e749aa1a
```

6.2 Cache options

The following arguments are available to all cache instances, all are optional.

6.2.1 default_timeout

Cache.**default_timeout**

The default timeout in seconds that will be used for cached values. Defaults to None, meaning the value will never expire.

6.2.2 enabled

Cache.**enabled**

Wether the cache is enabled by default or not.

Note: You can override this behaviour by using *Cache.disable()* and *Cache.enable()* context managers

6.3 Cache methods

class lifter.caches.**Cache**(*default_timeout=None*, *enabled=True*)

disable()
Returns a context manager to bypass the cache:

```
with cache.disable():
    # Will ignore the cache
    manager.count()
```

enable()
Returns a context manager to force enabling the cache if it is disabled:

```
with cache.enable():
    manager.count()
```

get(*key*, *default=None*, *reraise=False*)
Get the given key from the cache, if present. A default value can be provided in case the requested key is not present, otherwise, None will be returned.

Parameters

- **key** (*bool*) – the key to query
- **default** – the value to return if the key does not exist in cache
- **reraise** – wether an exception should be thrown if now value is found, defaults to False.

Example usage:

```
        cache.set('my_key', 'my_value')

        cache.get('my_key')
        >>> 'my_value'

        cache.get('not_present', 'default_value')
        >>> 'default_value'

        cache.get('not_present', reraise=True)
        >>> raise lifter.exceptions.NotInCache
```

set(*key*, *value*, *timeout=<class 'lifter.caches.NotSet'>*)

Set the given key to the given value in the cache. A timeout may be provided, otherwise, the *Cache.default_timeout* will be used.

Parameters

- **key** (*str*) – the key to which the value will be bound
- **value** – the value to store in the cache
- **timeout** (*integer or None*) – the expiration delay for the value. None means it will never expire.

Example usage:

```
        # this cached value will expire after half an hour
        cache.set('my_key', 'value', 1800)
```

6.4 Available cache backends

At the moment, the only cache backend available is the `DummyCache`, that store values in a Python dictionary.

You can use it's code as a starting point to implement your own backends, using Redis or Memcached, for example.

CHAPTER 7

Contrib modules

A few contrib modules are available to integrate lifter with existing frameworks and/or libraries.

Contents:

7.1 Django

The `lifter.contrib.django` module provides integration of lifter's Python backend on django's queryset. Once configured, you can apply additional filters and orderings to a django queryset without extra calls to database.

It is compatible with django versions >= 1.7.

7.1.1 Installation

You just need to add `lifter.contrib.django` to your installed apps:

```
INSTALLED_APPS = [
    # your other apps
    'lifter.contrib.django',
]
```

Internally, the application will monkey patch django's `QuerySet`, by adding a single method.

7.1.2 Usage

Usage is really simple:

```
from django.contrib.auth import User

# this is your regular django queryset
all_users = User.objects.all()

# we convert it to lifter Python's backend
local_qs = all_users.locally()

# the first query will evaluate the django's queryset (if needed),
# triggering a database query
local_qs.count()
>>> 157
```

```
# other queries are executed locally, on already loaded models,
# so no database query will be thrown
for user in local_qs.exclude(username='bob').order_by('-username'):
    print(user)
```

Of course, you can any method or lookup that is supported by lifter's Python backend.

CHAPTER 8

Performance considerations

Under the hood, lifter uses plain Python code and, therefore, is just a higher level API to interact with iterables.

It means lifter queries will always be slower than vanilla Python filtering/querying using comprehension lists and for loops. If performance is critical in your application, and lifter is not fast enough, you'll have to use another tool.

At the moment, lifter is much, much more slower than vanilla Python equivalent (from 10x to 100x, depending on your dataset and the kind of filtering you apply). You can get an up to date comparison by running the benchmark suite:

```
# Get the code and install the package
git clone git@github.com:EliotBerriot/lifter.git
cd lifter
python setup.py install

# run the whole benchmark suite
python -m tests.benchmarks

# Run a subset of benchmarks
python -m tests.benchmarks single_filter combined_filter
```

Lifter performance is a huge concern and as it's API become more stable, focus will progressively move on from implementing new features to optimizing current ones.

If you want to help with that, by reporting a performance pitfall, fixing one, or even writing a benchmark, your contribution is welcome!

CHAPTER 9

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:

9.1 Types of Contributions

9.1.1 Report Bugs

Report bugs at https://github.com/eliotberriot/lifter/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.

9.1.2 Fix Bugs

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

9.1.3 Implement Features

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

9.1.4 Write Documentation

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

9.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/eliotberriot/lifter/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 :)

9.2 Get Started!

Ready to contribute? Here's how to set up *lifter* for local development.

1. Fork the *lifter* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/lifter.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 lifter
$ cd lifter/
$ 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 lifter 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.

9.3 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 2.7, 3.3, 3.4 and 3.5. Check https://travis-ci.org/eliotberriot/lifter/pull_requests and make sure that the tests pass for all supported Python versions.
4. The pull request must target the *develop* branch, since the project relies on git-flow branching model

9.4 Tips

To run a subset of tests:

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

CHAPTER 10

Credits

10.1 Development Lead

- Eliot Berriot <contact@eliotberriot.com>

10.2 Contributors

- Ogreman
- Pentusha
- Mec-iS
- johnfraney
- youtux
- angru
- radarhere

Join us ;)

CHAPTER 11

History

RefinedStore removal

RefinedStore where entirely removed from lifter, resulting in a cleaner API / logic. You can now implement your own backend simply by creating you own *Store* class.

11.1 0.4.1 (2016-8-2)

This release fix issue #42: some files were not included in lifter distribution, mainly the `backends` and `contrib` directories, causing various imports to fail.

11.2 0.4 (2016-7-20)

This release introduces the django contrib module to enable filtering with lifter's python backend on a django queryset, effectively reducing number of queries sent to the database.

Work is also in progress regarding caching (see #39) but this is not over yet.

11.3 0.3 (2016-7-12)

This is a big release, that breaks backward-compatibility with previous ones.

This release implements a new flow to help implementing #33. The general idea is to make lifter generic and be able to query any data source with it.

The 0.3 release sets the foundation for that by moving all python-iterable related code to a dedicated backend, and by implementing the Store -> Adapter > Model layout to deal with queries and result parsing.

An additional, very simple, `filesystem` backend is provided to demonstrate how you can implement your own datasource in lifter.

The work, though, is still incomplete, because the *filesystem* store internally uses the *IterableStore* from the python backend.

A real store (such as REST or SQL) would be able to understand queries and pass them to a real backend (PostgreSQL).

Anyway, we're in the good direction here :)

11.4 0.2.1 (2016-3-4)

This is a small release, with a few improvements on ordering API and on the overall documentation:

- Can now order using multiple fields, fix #30
- [Backward incompatible] Can now invert ordering in explicit engine using path and ~ operator. Passing a *reverse* argument to *order_by* is not possible anymore
- Can now query for field existences, fix #26

11.5 0.2 (2016-2-23)

This is quite an important release:

- A whole new API is now available to make queries, see #15 for more information [angru, eliotberriot]
- Querysets are now lazy
- A brand new documentation is now available at http://lifter.readthedocs.org
- Splitted some huge files into submodules for more clarity

Considering the new query API, we basically switched from django's ORM-style (keyword engine) to SQLAlchemy style (explicit engine).

The keyword engine is still available and will continue to work as before. It is neither under depreciation at the moment, but using the new engine is recommended.

11.6 0.1.1 (2016-2-21)

- Can now pass arguments to underlying manager via lifter.load
- Random order_by for queryset [Pentusha]
- Improve code examples readability in readme
- Removed duplicate method on queryset [Mec-iS]
- Can now run some lookups within iterables (WIP) [Ogreman].
- Lots of improvements and corrections (typos, examples, etc.) in README [johnfraney, youtux]
- Can now return flat lists as results for aggregates [Ogreman]

11.7 0.1.0 (2016-2-17)

- First release on PyPI.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

l

A

C

D

E

F

G

L

O

S

V