SQLAlchemy filters plus

Release 1.0

El Mehdi Karami

Dec 03, 2022

USER GUIDE:

sqlalchemy-filters-plus is a light-weight extendable library for filtering queries with sqlalchemy.

CHAPTER

ONE

INSTALLATION

To install sqlalchemy-filters-plus use the following command using `pip`:

```
$ pip install sqlalchemy-filters-plus
```

1.1 Requirements

sqlalchemy-filters-plus is tested against all supported versions of Python from 3.6 to 3.9 as well as all versions of SQAlchemy from 1.0 to 1.4.

Since this is library enhances the way you filter queries with SQLAlchemy you will obviously need to have it in your requirements file.

CHAPTER

TWO

USAGE

This library provides an easy way to filter your SQLAlchemy queries, which can for example be used by your users as a filtering mechanism for your exposed models via an API.

Let's define an example of models that will be used as a base query.

```
from sqlalchemy import Column, Date, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class User(Base):
    id = Column(Integer, primary_key=True)
    email = Column(String)
    age = Column(Integer)
    birth_date = Column(Date, nullable=False)


class Article(Base):
    id = Column(Integer, primary_key=True)
    title = Column(String)
    user_id = Column(Integer, ForeignKey(User.id), nullable=False)
    user = relationship(
        User,
        uselist=False,
        lazy="select",
        backref=backref("articles", uselist=True, lazy="select"),
    )
```

2.1 Define your first filter

Let's then define our first Filter class for the User model

```
from sqlalchemy_filters import Filter, Field
from sqlalchemy_filters.operators import EqualsOperator


class EmailFilter(Filter):
    email = Field(lookup_operator=EqualsOperator)
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = User
        session = my_sqlalchemy_session
```

The example above defines a new filter class attached to the User model, we declared one field to filter with, which is the `email` field and and we defined the `lookup_operator` (default value is `EqualsOperator`) that will be used to filter with on the database level. We will see other operators that can also be used.

To apply the filter class, we instantiate it and pass it the data(as a dictionary) to filter with.

```
my_filter = EmailFilter(data={"email": "some@email.com"})
query = my_filter.apply()  # query is a SQLAlchemy Query object
```

Note: Meta.session is optional, but if it's not provided at declaration time, then it needs to be passed at the instantiation level or replaced by a sqlalchemy `Query`.

Example:

```
my_filter = EmailFilter(data={"email": "some@email.com"}, session=my_session)
# or
my_filter = EmailFilter(data={"email": "some@email.com"}, query=some_query)
```

2.1.1 Related models

Fields can also refer to columns in related models (Foreign keys), let's extend our models to include a new one: Category.

Category will have a one-to-many relationship with the Article

```
class Category(Base):
    __tablename__ = "categories"
    id = Column(Integer, primary_key=True)
    name = Column(String)


class Article(Base):
    ...
    category_id = Column(Integer, ForeignKey(Category.id), nullable=False)
    category = relationship(
        Category,
        uselist=False,
        lazy="select",
        foreign_keys=[category_id],
        backref=backref("articles", uselist=True, lazy="select"),
    )
```

we can now create a new filter that makes use of these relationships in a very simple way(especially when dealing with joins).

Let's take this example: Ability to filter authors by category name and by article title

```
from sqlalchemy_filters import Filter, StringField
from sqlalchemy_filters.operators import EqualsOperator, IStartsWithOperator,␣
↪IContainsOperator


class AuthorFilter(Filter):
    title = StringField(
        field_name="articles.title", lookup_operator=IContainsOperator
    )
    category = StringField(
        field_name="articles.category.name", lookup_operator=IContainsOperator
    )

    class Meta:
        model = User
        session = my_sqlalchemy_session
```

Warning: Trying to inherit from a filter that has a different model class will raise a *OrderByException* FilterNotCompatible.

2.2 Declaring fields

Declaring fields is generally used to specify the attributes will be used to query the database, but it can get far more complex that just that. With SQLAlchemy filters plus you can define fields by using either one of these two methods or combining them:

1. Define each attribute using the Field class as we described in the example above.
2. Set the `fields` attributes on the metadata to indicate the fields that you can filter with

The first method gives you most flexibility using pre-defined or custom operators while the other one only works with the `EqualOperator`

These two block defines exactly the same filter

```
class EmailFilter(Filter):
    email = Field(lookup_operator=EqualsOperator)

    class Meta:
        model = User
        session = my_sqlalchemy_session

# EmailFilter behaves exactly the same as EmailFilter2

class EmailFilter2(Filter):

    class Meta:
        model = User
        session = my_sqlalchemy_session
        fields = ["email"]
```

So if you're trying to use only the `EqualsOperator` you can just define them using the `fields` attributes on the meta class.

> **Warning:** Once the fields attribute is set and not empty, it has to include the fields that were declared explicitly inside the filter class, otherwise they will be ignored.

```
from sqlalchemy_filters.operators import StartsWithOperator

class MyFilter(Filter):
    email = Field(lookup_operator=StartsWithOperator)

    class Meta:
        model = User
        session = my_sqlalchemy_session
        fields = ["age", "email"]
```

For fields that were not explicitly declared, SQLAlchemy filters plus will try to match the appropriate Field type for it, in this example `age` will be of type `sqlalchemy_filters.IntegerField`.

2.3 Field options

- `field_name`: The attribute name of the fields must not necessarily be the name of the Model attribute, as long as we override the Field's `field_name`. Example:

```
class MyFilter(Filter):
    # Note that the filter class will look for `email_address` inside the provided data
    email_address = Field(field_name="email")
```

> **Warning:** If none of the attribute name/field name is found on the Model, an `AttributeError` will be thrown.

- `lookup_operator`: (default: *`EqualsOperator`*) Accepts an operator class used to specify how to perform the lookup operation on the database level.
- `custom_column`: Used to filter explicitly against a custom column, it can accept a `str`, `column` object or a model attribute as shown below:

```
class MyFilter(Filter):
    email_address = Field(custom_column="email")
    user_age = Field(custom_column=column("age"))
    user_birth_date = Field(custom_column=User.birth_date)
```

- `data_source_name` defines the key used to look for the field's value inside the data dictionary.

```
class MyFilter(Filter):
    email = Field(data_source_name="email_address")

...

f = MyFilter(data={"email_address": "some@email.com"})
```

- `allow_none` (default to `False`): allow filtering with None values. Only if the data contains the value *None*:

```
class MyFilter(Filter):
    email = Field(allow_none=True)

...
# Will filter by "email is None" in the database level
MyFilter(data={"email": None}).apply()
# No filtering will be applied to the database
MyFilter(data={}).apply()
```

Note: When *allow_none* is switched off, sending None values will be ignored.

2.4 Method fields

`MethodField` is a field that delegates the filtering part of a specific field to a Filter method or a custom function.

```
from sqlalchemy import func
from sqlalchemy_filters.fields import MethodField

def filter_first_name(value):
    # sanitize value and filter with first_name column
    return func.lower(User.first_name) == value.lower()

class MyFilter(Filter):
    email = MethodField("get_email")
    my_field = MethodField(filter_first_name, data_source_name="custom_key")

    class Meta:
        model = User

    def get_email(self, value):
        domain = value.split("@")[1]
        return User.first_name.endswith(domain)


MyFilter(data={"email": "some@email.com", "custom_key": "John"}).apply()
```

The methods/functions that were used for filtering should return a sql expression that SQLAlchemy can accept as a parameter for the `filter` function of a Query.

The benefit of using a object method is that you can access other values which can be useful to filter based on multiple inputs using `self.data`.

Note: MethodField can also be referenced inside *Meta.fields*.

Warning: MethodFields do not validated input values. It is strongly recommended to validate the value before filtering.

2.5 Paginating results

Giving users the ability to paginate through results matching some filters is mandatory in every modern application.

To paginate result, you should add a *page_size* attribute to the class *Meta* of the filter or pass it as part of the data at the instantiation level. Calling the `paginate` on a filter object will return a `Paginator` object, this object should do all the heavy lifting of slicing and paginating through objects from the database.

Here is an example of how can the paginator be generated:

```
class MyFilter(Filter):
    first_name = StringField()

    class Meta:
        model = User
        page_size = 10
# Or
>>> data = {
    #...
    "page_size": 20
}
# Note that we did not specify which page to get, by default it will return the first␣
↪page
>>> paginator = MyFilter(data=data).paginate()
>>> paginator.page
1
# We can specify the exact page we want by passing it as part of the data
>>> data["page"] = 2
>>> paginator = MyFilter(data=data).paginate()
>>> paginator.page
2
# The paginator object has plenty of methods to make your life easier
>>> paginator.has_next_page()
True
>>> paginator.has_previous_page()
True
# how many pages should we expect given that the total object matching query and the␣
↪page_size parameter
>>> paginator.num_pages
5
# How many objects match the query
>>> paginator.count
95
>>> next_paginator = paginator.next_page()
>>> next_paginator.page
3
>>> previous_paginator = next_paginator.previous_page()
>>> previous_paginator.to_json()
{
    "count": 95,
    "page_size": 20,
    "page": 2,
    "num_pages": 5,
    "has_next_page": True,
```

(continues on next page)

(continued from previous page)

```
    "has_prev_page": True,
}
# Will return the objects matching the page of the paginator
>>> users = paginator.get_objects()
# Will return the sliced query using `limit` and `offset` accordingly
>>> query = paginator.get_sliced_query()
```

2.6 Ordering results

sqlalchemy-filters-plus gives you the possibility to filter the queries by one or multiple fields.

You can either specify a fixed number of fields to order by or override this behavior at instantiation level.

To tell *sqlalchemy-filters-plus* how to order you results, add a *order_by* attribute in the *Meta* class, this attribute accepts multiple formats:

1. Specify directly the field you want to order by (using the *SQLAlchemy way*)

```
class MyFilter(Filter):
    first_name = StringField()

    class Meta:
        model = User
        order_by = User.first_name.asc()

# Or as a list

class MyFilter(Filter):
    first_name = StringField()

    class Meta:
        model = User
        order_by = [User.first_name.asc(), User.last_name.desc()]
```

2. Specify the field(s) as a string or as a list of strings, *sqlalchemy-filters-plus* will evaluate the string to decide which ordering should be applied. Prefix the field name with a `-` (minus) to apply descending order or omit it for ascending.

```
class MyFilter(Filter):
    first_name = StringField()

    class Meta:
        model = User
        order_by = "first_name" # ascending
        # Or as a list
        # First name ascending, while last_name descending
        order_by =  ["first_name", "-last_name"]
        # or Multiple fields as a single string
        # The space between fields will be ignored, but recommended for readability
        order_by =  "first_name, -last_name"
```

Notice that the last option enables us to use it as an ordering mechanism for an API, giving users the ability to order by any field

```
>>> MyFilter(data={"order_by": "first_name, -last_name"})
>>> MyFilter(data={"order_by": ["first_name", "-last_name"]})
>>> MyFilter(data={"order_by": "first_name"})
>>> MyFilter(data={"order_by": User.first_name.asc()})
>>> MyFilter(data={"order_by": [User.first_name.asc(), User.last_name.desc()]})
```

Warning: Specifying a field that does not belong to the model class will raise an *OrderByException* exception.

CHAPTER

THREE

OPERATORS

SQLAlchemy filters plus provides a handful operators that makes easy to filter objects in a flexible manner. These operators define how to filter columns in the database. Basically an operator reflects an sql operation that could be performed on a column, a value or both.

3.1 API Usage

The Operator API is pretty straightforward, we can think of them as wrappers of the builtin sqlalchemy operators.

Here's an example on how we can use an Operator:

```
from sqlalchemy import column

from sqlalchemy_filters.operators import IsOperator, BetweenOperator

is_operator = IsOperator(sql_expression=column("my_column"), params=["value"])
is_operator.to_sql()  # equivalent to column("my_column").is_("value")

is_operator = BetweenOperator(sql_expression=column("age"), params=[20, 30])
is_operator.to_sql()  # equivalent to column("age").between(20, 30)
```

3.2 Define custom operators

Sometimes the provided operators are not enough, hence the need of creating custom operators. Fortunately, this is a simple process as shown bellow.

Let's say we want to have a custom operator that tries to match the end of a string in lower case

```
from sqlalchemy import func
from sqlalchemy.sql.operators import endswith_op

from sqlalchemy_filters.operators import BaseOperator, register_operator


@register_operator(endswith_op)
class MyCustomOperator(BaseOperator):

    def to_sql(self):
```

(continues on next page)

(continued from previous page)

```
        return self.operator(func.lower(self.sql_expression), *map(func.lower, self.
↪params))
```

Sometime there is no builtin SQLALchemy operator that can be used to make life easier for what you want to do, the good part about *sqlalchemy_filters.operators.register_operator*, is that you don't have to register anything. Example:

```
@register_operator
class MyCustomOperator(BaseOperator):

    def to_sql(self):
        return self.sql_expression == "ABC"
```

CHAPTER

FOUR

VALIDATION

Validating inputs at the filters/fields level is crucial to be in accordance of what the database expects and prevent unexpected errors.

SQLAlchemy filters plus provides multiple level of validations.

4.1 How it works

Field validation is ensuring that a specific field value is what we expect it to be. There are multiple field types that are predefined and can be used to validate the desired fields.

Let's see an example of how we can apply that into our example:

```
from sqlalchemy_filters import Filter, DateField


class MyFilter(Filter):
    birth_date = DateField()  # can take a format parameter, default is "%Y-%m-%d"

    class Meta:
        model = User
        session = my_session
```

The above defines a filter with a single `DateField` field. This will ensure that the passed value is a *datetime* value or can be parsed as a *datetime*. Otherwise a `FilterValidationError` exception will be thrown.

```
>>> from sqlalchemy_filters.exceptions import FilterValidationError
>>> try:
>>>     MyFilter(data={"birth_date": "abc"}).apply()
>>> except FilterValidationError as exc:
>>>     print(exc.json())
[
    {"birth_date": "time data 'abc' does not match format '%Y-%m-%d'"}
]
```

This exception encapsulates all the field errors that were encountered, it also provides a `json()` method to make it human readable which gives the possibility of returning it as a response in a REST API.

It's also a wrapper around the `FieldValidationError` exception, you can get the full list of wrapped exceptions by accessing to `fields_errors` attribute

```
>>> exc.field_errors
```

4.2 Custom Schema Validation

SQLAlchemy filters plus support custom validation with Marhsmallow.

The Marshmallow schema will provide a validation for the whole *Filter* class.

Let's define our fist Marshmallow schema

```
from marshmallow import Schema, fields, validate


class FirstNameSchema(Schema):
    first_name = fields.String(validate=validate.OneOf(["john", "james"]), required=True)
```

First define a Marshmallow schema, then we can inject it into the Filter class using 2 approaches:

1. The first one is using *Meta.marshmallow_schema* attribute:

```
from sqlalchemy_filters import Filter, StringField


class MyFilter(Filter):

    class Meta:
        model = User
        fields = ["first_name"]
        session = my_session
        marshmallow_schema = FirstNameSchema

>>> MyFilter(data={"first_name": "test"}).apply()
marshmallow.exceptions.ValidationError: {'first_name': ['Must be one of: john,␣
↪james.']}
```

2. Or pass it as an argument at the instantiation level of the filter class

```
>>> MyFilter(data={"first_name": "test"}, marshmallow_schema=FirstNameSchema).
↪apply()
marshmallow.exceptions.ValidationError: {'first_name': ['Must be one of: john,␣
↪james.']}
```

4.3 Define custom field and validation

Field validation is performed by the *validate* method. The Filter class calls the validate method for each defined field.

To create a custom field validation we can inherit from the *Field* class or any other class that inherits from the Field class (example: StringField, DateField...) and redefine the validate method, the return value will be used to filter the column with, or an *FieldValidationError* exception can be raised

Example:

```
from sqlalchemy_filters.fields import StringField
from sqlalchemy_filters.exceptions import FieldValidationError


class EmailField(StringField):
    def validate(self, value):
        value = super().validate(value)
        if "@mydomain.com" not in value:
            raise FieldValidationError("Only emails from mydomain.com are␣
↪allowed.")
        return value
```

CHAPTER

FIVE

NESTED FILTERS

SQLAlchemy filters plus provides a way to build very complex queries by using *NestedFilter* which make use of existing filter classes to act as Field with the ability to specify how the inner fields of that NestedFilter should be grouped and specifying how to combine it with the other declared fields using *AndOperator* and *OrOperator*.

Let's create an complete example:

```
from sqlalchemy.fields import StringField, IntegerField, MethodField
from sqlalchemy.filters import DateField, Filter, NestedFilter
from sqlalchemy.operators import GTOperator, LTEOperator, ContainsOperator, AndOperator,␣
↪OrOperator

class MyFilter(Filter):
    min_age = IntegerField(field_name="age", lookup_operator=GTOperator)
    max_age = IntegerField(field_name="age", lookup_operator=LTEOperator)
    created_at = MethodField(method="filter_created_at")

    def filter_created_at(self, value):
        return User.created_at == value

    class Meta:
        model = User
        fields = ["first_name", "min_age", "max_age", "created_at"]

class SecondFilter(Filter):
    email = StringField(lookup_operator=ContainsOperator)
    max_birth_date = DateField(field_name="birth_date", lookup_operator=LTEOperator)
    nested_data = NestedFilter(
        MyFilter,
        operator=AndOperator,  # How to join the inner fields of MyFilter (first_name,␣
↪min_age)
        outer_operator=OrOperator,  # How to join the result of the NestedFilter with the
                                    # rest of SecondFilter's fields
        data_source_name="nested_data_key",  # Used to specify from where the data␣
↪should be extracted
                                             # ignored if flat is True
        flat=False  # True if MyFilter fields should be extracted at the root level,
                    # If False, then the fields data will be extracted from the key data_
↪source_name if specified
                    # otherwise from the NestedFilter field name, in our example it's␣
↪`nested_data`
    )
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        model = User
        session = my_session
```

Let's filter some objects

```
my_filter = SecondFilter(data={
    "email": "@example",
    "max_birth_date": "1980-01-01",
    "nested_data": {
        "first_name": "John",
        "min_age": 25,
        "max_age": 45,
        "created_at": "2020-01-01",
    }
}, operator=OrOperator)
users = my_filter.apply_all()
```

The result of the sql query would be similar to

```
SELECT users.id,
       users.first_name,
       users.last_name,
       users.email,
       users.age,
       users.is_approved,
       users.birth_date,
       users.created_at,
       users.last_login_time
FROM   users
WHERE  ( users.created_at = '2020-01-01'
          OR users.age > 25
          OR users.age <= 45
          OR users.first_name = 'John' )
       AND ( (users.email LIKE '%' || '@example' || '%') OR users.birth_date <= '1980-01-
↪01' )
```

Note: NestedFilter can use other NestedFilters as fields.

CHAPTER

SIX

APIS

6.1 operators

This module contains the defined operators used to construct simple or more complex sql queries.

`sqlalchemy_filters.operators.register_operator(cls: Optional[Type] = None, *, sql_operator: Optional[Callable] = None)`

Register a class as an operator class.

Parameters

- **cls** – Registering an operator without providing a builtin SQLAlchemy builtin operator.
- **sql_operator** – A sqlalchemy operator or a custom callable that acts as an sqlalchemy operator.

`sqlalchemy_filters.operators.sa_1_4_compatible(f)`

Decorator for the method `BaseOperator.to_sql`

Since *TextClause* does not support *BinaryExpression* as a left operand in SqlAlchemy 1.4, we revert the left/right sides of the operation

Ex:

```
>>> # raises: unsupported operand type(s) for | TextClause and BinaryExpression
>>> text("1 = 1") | (column("x") == 1)
```

would change to:

```
>>> (column("x") == 1) | text("1 = 1")
```

`class sqlalchemy_filters.operators.BaseOperator(sql_expression: V, params: Optional[List[T]] = None)`

Bases: `object`

Base operator class.

Inherit from this class to create custom operators.

operator: Callable

sqlalchemy operator, but can also be any callable that accepts `sql_expression` handled by sqlalchemy operators

classmethod __init_subclass__(***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__init__(*sql_expression: V*, *params: Optional[List[T]] = None*)

sql_expression = None

Anything that can be used an operand for the sqlalchemy operators.

params: list

A list of parameters or operands for the operator

get_sql_expression() → ClauseElement

Returns a *ClauseElement* depends on the `sql_expression` is

Returns

to_sql() → BinaryExpression

Execute the operator against the database.

classmethod check_params(*params: list*) → None

Validates the params.

Can be refined by subclasses to define a custom validation for `params`

Parameters
params – operands for the operator.

Raises
`InvalidParamError` if checking failed.

__weakref__

list of weak references to the object (if defined)

class sqlalchemy_filters.operators.**IsOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: `BaseOperator`

is sql operator.

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**IsNotOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: `BaseOperator`

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**IsEmptyOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

params: list = [None]

A list of parameters or operands for the operator

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

class sqlalchemy_filters.operators.**IsNotEmptyOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

params: list = [None]

A list of parameters or operands for the operator

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

class sqlalchemy_filters.operators.**INOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

to_sql() → BinaryExpression

Execute the operator against the database.

property operator

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**EqualsOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**RangeOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

classmethod check_params(*params: list*) → None

Validates the params.

Can be refined by subclasses to define a custom validation for *params*

Parameters

params – operands for the operator.

Raises
InvalidParamError if checking failed.

property operator

to_sql() → BinaryExpression
Execute the operator against the database.

params: list
A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**LTEOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)
Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression
Execute the operator against the database.

params: list
A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**LTOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)
Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression
Execute the operator against the database.

params: list
A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**GTEOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)
Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression
Execute the operator against the database.

params: list
A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**GTOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)
Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression
Execute the operator against the database.

params: list
A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**AndOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)
Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**OrOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**ContainsOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**IContainsOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

to_sql()

Execute the operator against the database.

property operator

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**StartsWithOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**IStartsWithOperator**(*sql_expression: V*, *params: Optional[List[T]] = None*)

Bases: *StartsWithOperator*

to_sql()

Execute the operator against the database.

property operator

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**EndsWithOperator**(*sql_expression: V, params: Optional[List[T]] = None*)

Bases: *BaseOperator*

property operator

to_sql() → BinaryExpression

Execute the operator against the database.

params: list

A list of parameters or operands for the operator

class sqlalchemy_filters.operators.**IEndsWithOperator**(*sql_expression: V, params: Optional[List[T]] = None*)

Bases: *BaseOperator*

property operator

params: list

A list of parameters or operands for the operator

to_sql() → BinaryExpression

Execute the operator against the database.

6.2 Filters

Modules defines the Filter and NestedFilter classes

class sqlalchemy_filters.filters.**NestedFilter**(*filter_class: ~typing.Type[~sqlalchemy_filters.filters.Filter], operator: ~typing.Type[~sqlalchemy_filters.operators.BaseOperator] = <class 'sqlalchemy_filters.operators.AndOperator'>, outer_operator: ~typing.Optional[~typing.Type[~sqlalchemy_filters.operators.BaseOperator]] = None, flat: bool = True, data_source_name: ~typing.Optional[str] = None, marshmallow_schema=None*)

Bases: object

NestedFilters are a way to use already defined filter classes as fields and build complex queries.

__init__(*filter_class: ~typing.Type[~sqlalchemy_filters.filters.Filter], operator: ~typing.Type[~sqlalchemy_filters.operators.BaseOperator] = <class 'sqlalchemy_filters.operators.AndOperator'>, outer_operator: ~typing.Optional[~typing.Type[~sqlalchemy_filters.operators.BaseOperator]] = None, flat: bool = True, data_source_name: ~typing.Optional[str] = None, marshmallow_schema=None*)

filter_class: Type[*Filter*]

The filter class that will be used as a *NestedFilter* field

operator

How to join the inner fields of the *filter_class*

alias of *AndOperator*

outer_operator: Optional[Type[*BaseOperator*]]

Operator describes how to join the inner fields of the NestedFilter and the field of the parent filter If not defined the operator of the parent filter will be used

flat: bool = False

If True, means that the nested filter should get the value of its fields from root level of the data If False, the values will be extracted using either *data_source_name* if defined or the name of field in the parent filter:

```
>>> # data for the nested filter will be extracted from the key `custom_field`
>>> custom_field = NestedFilter(MyFilter, flat=False)
```

data_source_name: Optional[str] = None

key to look for the nested filter fields from the data, this is ignored if *flat* is True.

marshmallow_schema = None

Custom Marshmallow for validation

__eq__(*other*)

Return self==value.

get_data_source_name() → str

Returns
key used to extract the data that will be used to validate and filter with.

get_data(*parent_filter_obj:* BaseFilter) → dict

How to extract the data from the parent filter object.

Parameters
parent_filter_obj – The parent filter object instance.

Returns
data that will be passed to the *filter_class*

apply(*parent_filter_obj:* BaseFilter)

Gets the data from the parent filter then apply the filter for the *filter_class*

Parameters
parent_filter_obj – The parent filter object instance.

Returns
the sql expression that will be combined with `parent_filter_obj`'s inner fields.

__hash__ = None

__weakref__

list of weak references to the object (if defined)

class sqlalchemy_filters.filters.**BaseFilter**(*, *data: dict*, *operator: ~typing.Type[~sqlalchemy_filters.operators.BaseOperator] = <class 'sqlalchemy_filters.operators.AndOperator'>*, *query=None*, *session=None*, *marshmallow_schema=None*)

Bases: `object`

Base filter that any other filter class should inherit from.

fields: Dict[str, *Field*] = {}

Fields declared using any class that inherits from *Filter*

nested: Dict[str, *NestedFilter*] = {}

NestedFilter

method_fields: Dict[str, *MethodField*] = {}

Method fields

validated: bool

Flag to whether the data was validated or not

__init__(*, *data: dict*, *operator: ~typing.Type[~sqlalchemy_filters.operators.BaseOperator] = <class 'sqlalchemy_filters.operators.AndOperator'>*, *query=None*, *session=None*, *marshmallow_schema=None*)

data: dict

Contains the original data passed to the constructor

validated_data: Dict

Contains the validated data extracted from *data*

operator

Operator describing how to join the fields

alias of `Type[`*BaseOperator*`]`

session: Any = None

sqlalchemy session object

marshmallow_schema: Any = None

marshmallow schema class

set_query()

Sets the query to the current filter object (called at the *__init__()* method).

Returns
None

validate_nested()

Validate all *NestedFilters* fields.

Raise
FilterValidationError

Returns
None

validate_fields()

Validates the data by calling *validate* of each field.

Each validated value is put back inside *validated_data* using the return of field method *get_data_source_name* as a key. That value will be used as as input to the operator of the corresponding field.

Raise
`FilterValidationError`

Returns
None

validate()

Calls `validate_fields` and `validate_nested`. If no error is raised, the `validated` attribute is set to *True*

Returns
None

__weakref__

list of weak references to the object (if defined)

apply_fields()

Calls `apply_filter` of each field and join them using the defined `operator`

Returns
SQLAlchemy *BinaryExpression*

apply_nested(*filters*)

Calls apply filter of each field and join them using the defined `operator`

Parameters
filters – The return value of `apply_fields()`

Returns
SQLAlchemy *BinaryExpression*

apply_methods(*filters*)

Calls `apply_filter` of each field and join them

Parameters
filters – The return value of `apply_fields()`

Returns
SQLAlchemy *BinaryExpression*

apply_all()

Validates the `data` and applies all the fields.

This method can be used to get the sqlalchemy *BinaryExpression* without having to construct a SQLAlchemy *Query* object.

Returns
SQLAlchemy *BinaryExpression*

order_by(*query*)

Order the query by the specified *order_by* in the Meta class :param query: SQLAlchemy *Query* object. :return: Ordered SQLAlchemy *Query* object.

paginate() → *Paginator*

Creates a paginator that does all the work to slice the queries into a `Paginator` object.

Returns
Return a `Paginator`

apply()

Applies all fields, then using that result to filter the query then apply the joining of any potentiel foreign keys.

Returns
SQLAlchemy *Query* object.

class sqlalchemy_filters.filters.**Filter**(*, *marshmallow_schema: Optional[Type] = None*, ***kwargs*)

Bases: `MarshmallowValidatorFilterMixin`, *`BaseFilter`*

Filter class.

Makes use of MarshmallowValidatorFilterMixin to add the marshmallow validation capability.

fields: Dict[str, *Field*] = {}

Fields declared using any class that inherits from *`Filter`*

marshmallow_schema: Any = None

marshmallow schema class

method_fields: Dict[str, *MethodField*] = {}

Method fields

nested: Dict[str, *NestedFilter*] = {}

NestedFilter

session: Any = None

sqlalchemy session object

validated: bool

Flag to whether the data was validated or not

data: dict

Contains the original data passed to the constructor

validated_data: Dict

Contains the validated data extracted from *`data`*

6.3 Fields

This module defines all types of fields used by the filter classes.

class sqlalchemy_filters.fields.**BaseField**(*, *field_name: ~typing.Optional[str] = None*, *lookup_operator: ~typing.Optional[~typing.Type[~sqlalchemy_filters.operators.BaseOperator]] = <class 'sqlalchemy_filters.operators.EqualsOperator'>*, *join: ~typing.Optional[~typing.Union[~typing.Any, ~typing.Tuple[~typing.Any, ~typing.Any]]] = None*, *custom_column: ~typing.Optional[~sqlalchemy.sql.elements.ColumnClause] = None*, *data_source_name: ~typing.Optional[str] = None*, *allow_none: ~typing.Optional[bool] = False*)

Bases: `object`

Base field class

__init__(*, *field_name: ~typing.Optional[str] = None*, *lookup_operator: ~typing.Optional[~typing.Type[~sqlalchemy_filters.operators.BaseOperator]] = <class 'sqlalchemy_filters.operators.EqualsOperator'>*, *join: ~typing.Optional[~typing.Union[~typing.Any, ~typing.Tuple[~typing.Any, ~typing.Any]]] = None*, *custom_column: ~typing.Optional[~sqlalchemy.sql.elements.ColumnClause] = None*, *data_source_name: ~typing.Optional[str] = None*, *allow_none: ~typing.Optional[bool] = False*) → None

Parameters

- **field_name** –

  Field name of the model, can also refer to a field in a foreign key.
  We don't not have to specify it if that field name that's defined with is the same as the model attribute

```
>>> class MyFilter(Filter):
>>>     # field1 is not a attribute/field of the model
>>>     # hence we specified it explicitly
>>>     field1 = Field(field_name="column")
>>>     # field2 is an attribute/field of the model
>>>     # we don't have to explicitly declare it
>>>     field2 = Field()
>>>     field3 = Field(field_name="foreign_model.attribute")
>>>     ...
```

- **lookup_operator** – The operator class used to join the fields filtering together. Can only be AndOperator or OrOperator.
- **join** – A Model to join the query with, can also be a tuple. This will be passed to the *join* method of the SQLAlchemy Query object.
- **custom_column** –

  You can use a custom column to filter with. It can accept a string, a column or a Model
  field

```
>>> from sqlalchemy import column
>>>
>>> class MyFilter(Filter):
>>>     field1 = Field(custom_column="my_column")
>>>     field2 = Field(custom_column=column("some_column"))
>>>     field3 = Field(custom_column=MyModel.field)
>>>     ...
```

- **data_source_name** – The key used to extract the value of the field from the data provided.
- **allow_none** – (default to `False`): If set to *True* it allows filtering with None values. But Only if the data contains the value *None*

__eq__(*other*)

Return self==value.

validate(*value*) → Any

Validates the value

Parameters

value – value extracted from the original data

Returns
Sanitized or original value

Raises
`FieldValidationError` if validation fails. This is used for custom validation.

get_data_source_name() → str

Returns
Return the key to be used to look for the value of the field.

get_field_value(*data*)

Parameters
data – Data provided while instantiating the filter class (pre-validation: `data`)

Returns
The field value from the data, if not found it returns *Empty* class.

get_field_value_for_filter(*filter_obj*)

Extracts the value of the field from the `validated_data`.

Parameters
filter_obj – The filter instance

Returns
The field value from the data, if not found it returns *Empty* class.

apply_filter(*filter_obj*)

Applies the filtering part using the operator class and the value extracted using `get_field_value_for_filter()`.

Parameters
filter_obj – The Filter instance

Returns
SQLAlchemy *BinaryExpression*

__hash__ = None

__weakref__

list of weak references to the object (if defined)

class sqlalchemy_filters.fields.**MethodField**(*, *method: Union[Callable, str]*, *data_source_name=None*)

Bases: `BaseField`

Field used to delegate the filtering logic to a Filter method or a standalone function.

Warning
The `MethodField` does not provide any validation and consumes any values extracted from the `data` field.

__init__(*, *method: Union[Callable, str]*, *data_source_name=None*)

Parameters

- **method** – A callable that accepts a single value which is the field value.
- **data_source_name** – The key used to extract the value of the field from the data provided.

method: Callable

extract_method(*filter_obj*) → Callable

Extracts the method from the filter instance if found, otherwise checks if *method* is a callable

Parameters
filter_obj – the Filter instance

Returns
Callable used to apply the filtering part of the current field.

get_field_value_for_filter(*filter_obj*)

Extracts the value of the field from the *data*.

Parameters
filter_obj – The filter instance

Returns
The field value from the data, if not found it returns *Empty* class.

class sqlalchemy_filters.fields.**Field**(*, *field_name=None*, ***kwargs*)

Bases: ForeignKeyFieldMixin, *BaseField*

This is the Default field instance that can be instantiated as used as a filter field.

class sqlalchemy_filters.fields.**TypedField**(*, *field_name=None*, ***kwargs*)

Bases: *Field*

validate(*value: Any*) → Any

Validates the value

Parameters
value – value extracted from the original data

Returns
Sanitized or original value

Raises
FieldValidationError if validation fails. This is used for custom validation.

class sqlalchemy_filters.fields.**IntegerField**(*, *field_name=None*, ***kwargs*)

Bases: *TypedField*

type_

alias of int

class sqlalchemy_filters.fields.**DecimalField**(*, *field_name=None*, ***kwargs*)

Bases: *TypedField*

type_

alias of Decimal

class sqlalchemy_filters.fields.**FloatField**(*, *field_name=None*, ***kwargs*)

Bases: *TypedField*

type_

alias of float

class sqlalchemy_filters.fields.**StringField**(*, *field_name=None*, ***kwargs*)

Bases: *TypedField*

type_

alias of `str`

class sqlalchemy_filters.fields.**BooleanField**(*, *field_name=None*, ***kwargs*)

Bases: *TypedField*

type_

alias of `bool`

class sqlalchemy_filters.fields.**TimestampField**(*, *timezone=datetime.timezone.utc*, ***kwargs*)

Bases: *FloatField*

__init__(*, *timezone=datetime.timezone.utc*, ***kwargs*)

Parameters

- **field_name** –

  Field name of the model, can also refer to a field in a foreign key.
  We don't not have to specify it if that field name that's defined with is the same as the model attribute

```
>>> class MyFilter(Filter):
>>>     # field1 is not a attribute/field of the model
>>>     # hence we specified it explicitly
>>>     field1 = Field(field_name="column")
>>>     # field2 is an attribute/field of the model
>>>     # we don't have to explicitly declare it
>>>     field2 = Field()
>>>     field3 = Field(field_name="foreign_model.attribute")
>>>     ...
```

- **lookup_operator** – The operator class used to join the fields filtering together. Can only be AndOperator or OrOperator.
- **join** – A Model to join the query with, can also be a tuple. This will be passed to the *join* method of the SQLAlchemy Query object.
- **custom_column** –

  You can use a custom column to filter with. It can accept a string, a column or a Model
  field

```
>>> from sqlalchemy import column
>>>
>>> class MyFilter(Filter):
>>>     field1 = Field(custom_column="my_column")
>>>     field2 = Field(custom_column=column("some_column"))
>>>     field3 = Field(custom_column=MyModel.field)
>>>     ...
```

- **data_source_name** – The key used to extract the value of the field from the data provided.
- **allow_none** – (default to `False`): If set to *True* it allows filtering with None values. But Only if the data contains the value *None*

validate(*value: Union[int, float]*) → datetime

Validates the value

Parameters
value – value extracted from the original data

Returns
Sanitized or original value

Raises
FieldValidationError if validation fails. This is used for custom validation.

class sqlalchemy_filters.fields.**BaseDateField**(*, *date_format: str = '%Y-%m-%d'*, *is_timestamp=False*, ***kwargs*)

Bases: *TimestampField*

__init__(*, *date_format: str = '%Y-%m-%d'*, *is_timestamp=False*, ***kwargs*)

Parameters

- **date_format** – date_format that can be accepted by the *datetime.strptime* method.
- **is_timestamp** – True if it's intented to be used as a timestamp
- **kwargs** –

validate(*value: Union[str, datetime, date, int, float]*) → datetime

Validates the value

Parameters
value – value extracted from the original data

Returns
Sanitized or original value

Raises
FieldValidationError if validation fails. This is used for custom validation.

class sqlalchemy_filters.fields.**DateTimeField**(*, *datetime_format: str = '%Y-%m-%d'*, ***kwargs*)

Bases: *BaseDateField*

__init__(*, *datetime_format: str = '%Y-%m-%d'*, ***kwargs*)

Parameters

- **date_format** – date_format that can be accepted by the *datetime.strptime* method.
- **is_timestamp** – True if it's intented to be used as a timestamp
- **kwargs** –

validate(*value: Union[str, datetime, date, int, float]*) → datetime

Validates the value

Parameters
value – value extracted from the original data

Returns
Sanitized or original value

Raises
FieldValidationError if validation fails. This is used for custom validation.

class sqlalchemy_filters.fields.**DateField**(*, *date_format: str = '%Y-%m-%d'*, *is_timestamp=False*, ***kwargs*)

Bases: *BaseDateField*

validate(*value: Union[float, int, str, datetime, date]*) → date

Validates the value

Parameters
value – value extracted from the original data

Returns
Sanitized or original value

Raises
FieldValidationError if validation fails. This is used for custom validation.

6.4 Paginator

class sqlalchemy_filters.paginator.**Paginator**(*query*, *page: int*, *page_size: int*)

Bases: `object`

Utility class to help paginate through results of a SQLAlchemy query.

This is a 1-based index, meaning page=1 is the first page.

__init__(*query*, *page: int*, *page_size: int*)

has_next_page()

Returns
True if the current has is not the last page .

has_previous_page()

Returns
True if the current has is not the first page .

next_page()

If this current paginator is the last page, then this method will return the current one.

Returns
Paginator object

previous_page()

If this current paginator is the first page, then this method will return the current one.

Returns
Paginator object

get_objects()

Returns
Evaluates the query and returns the objects from the database.

get_sliced_query()

Returns
Can be used to get the sliced version of the query without evaluating it

to_json()

Returns
dictionary containing useful data in case of paginating through an API.

Example:

```
>>> paginator.to_json()
{
    "count": 111,
    "page_size": 10,
    "page": 2,
    "num_pages": 12,
    "has_next_page": True,
    "has_prev_page": True,
}
```

__weakref__

list of weak references to the object (if defined)

6.5 Exceptions

This module defined all exceptions used by the library.

exception sqlalchemy_filters.exceptions.**BaseError**

Bases: `Exception`

Base Exception for all exceptions.

exception sqlalchemy_filters.exceptions.**InvalidParamError**(*message: str*)

Bases: *BaseError*

Raised during the call of the *check_params* method of the Operator class.

exception sqlalchemy_filters.exceptions.**FieldMethodNotFound**(*parent_filter: Type*, *field_name: str*, *method_name: str*)

Bases: *BaseError*

Raised when a method specified using string is not found in the filter class.

exception sqlalchemy_filters.exceptions.**FieldValidationError**(*message: Optional[str] = None*)

Bases: *BaseError*

default_error: str = 'error validating this field.'

default error message

set_field_name(*field_name: str*) → None

sets `field_name`

Parameters

field_name – The field name of the errored field name.

Returns

None

json() → Dict[Optional[str], str]

Converts the error into a dictionary {field_name: error_message} :return: dict

exception sqlalchemy_filters.exceptions.**FilterValidationError**(*field_errors: List[FieldValidationError]*)

Bases: *BaseError*

field_errors: List[*FieldValidationError*]

List of *FieldValidationError*

json() → List[Dict[Optional[str], str]]

Jsonify all the *sqlalchemy_filters.exceptions.FieldValidationError* exceptions

Returns

List of dictionary representing the errors for each field

Example:

```
>>> exc.json()
[
    {"age": "Expected to be of type int"},
    {"last_name": "Expected to be of type str"}
]
```

exception sqlalchemy_filters.exceptions.**OrderByException**

Bases: *BaseError*

Raised when trying to order by a field that does not belong to the filter's model.

exception sqlalchemy_filters.exceptions.**FilterNotCompatible**(*class_name*, *model*, *base_filter*)

Bases: *BaseError*

Raised when trying to inherit from a filter(or used as a NestedFilter) with different model.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

N

O

P