django-rql

Release 3.7.1.dev1+gd44cc69.d20200814

CloudBlue

Aug 14, 2020

CONTENTS:

django-rql is an Django application, that implements RQL filter backend for your web application.

CHAPTER

ONE

RQL

RQL (Resource query language) is designed for modern application development. It is built for the web, ready for NoSQL, and highly extensible with simple syntax. This is a query language fast and convenient database interaction. RQL was designed for use in URLs to request object-style data structures.

RQL Reference

RQL for Web

1.1 Getting started

1.1.1 Requirements

django-rql works with Python 3.6 or newer and has the following dependencies:

- Django >= 1.11.20 and <= 3.0
- lark-parser 0.8.2

And the following optional dependency:

- djangorestframework >= 3.9

1.1.2 Install

django-rql can be installed from pypi.org with pip:

```
$ pip install django-rql
```

If you want to use *django-rql* with Django Rest Framework you have to install the optional dependency:

```
$ pip install django-rql[drf]
```

1.1.3 Write your first RQL Filter Class

For writing your first RQL Filter Class you need some models to be ready. Let's imagine you have simple Domain Model in your project, that can be represented as several models like below:

```
from django.db import models


class Product(models.Model):
    name = models.CharField()
```

Let's create an RQL Filter Class for `Product` model. All you need is to inherit from `dj_rql.filter_cls.RQLFilterClass`, define `MODEL` property and add supported `FILTERS` for class:

```
from dj_rql.filter_cls import RQLFilterClass


class ProductFilters(RQLFilterClass):
    MODEL = Product
    FILTERS = (
        'id',
        'name',
    )
```

Using simple strings in `FILTERS` property you can define what fields are available for filtering. In example above you allow filtering only by `id` and `name` filter.

1.1.4 Use your RQL filter class in your views

```
from urllib.parse import unquote

from products.filters import ProductFilters
from products.models import Product


def search_products_by_name(request):
    query = unquote(request.meta['QUERY_STRING'])

    base_queryset = Product.objects.all()

    my_filter = ProductFilters(base_queryset)

    _, filtered_qs = my_filter.apply_filters(query)

    return render(request, 'products/search.html', {'products': filtered_qs})
```

```
$ curl http://127.0.0.1:8080/api/v1/products?like(name,Unicorn*)|eq(name,LLC)
```

1.1.5 Use django-rql with Django Rest Framework

Configuring Django settings

Setup default *filter_backends* in your Django settings file:

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': ['dj_rql.drf.RQLFilterBackend']
}
```

Now your APIs are supporting RQL syntax for query strings.

Add RQL Filter Class to DRF View

In your latest step you need to add `ProductFilters` class as a `rql_filter_class` property inside your View:

```
class ProductsViewSet(mixins.ListModelMixin, GenericViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    rql_filter_class = ProductFilters
```

And that's it! Now you are able to start your local server and try to filter using RQL syntax

```
$ curl http://127.0.0.1:8080/api/v1/products?like(name,Unicorn*)|eq(name,LLC)
```

1.2 User guide

1.2.1 Supported operators

The following operators are currently supported by *django-rql*:

1. Comparison (eq, ne, gt, ge, lt, le, like, ilike, search)
2. List (in, out)
3. Logical (and, or, not)
4. Constants (null(), empty())
5. Ordering (ordering)
6. Select (select)

Note: This guide assumes that you have already read the RQL Reference.

1.2.2 Write your filter classes

A simple filter class looks like:

```
class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = ('a_field', 'another_field',)
```

Filter fields must be specified using the `FILTERS` attribute of the RQLFilterClass subclass.

For each field listed through the `FILTERS` attribute, *django-rql* determines defaults (lookup operators, null values, etc). For example if your field is a models.CharField by default you can use the operators `eq`, `ne`, `in`, `out`, `like`, `ilike` as long as the `null` constant.

Please refers to *Default lookups by field type* for a complete list of defaults.

If you want a fine grained control of your filters (allowed lookups, null values, aliases, etc) you can do that using a dictionary instead of a string with the name of the field.

Overriding default lookups

If you want for a certain filter to specify which lookups it supports you can do that using the `lookups` property:

```
from dj_rql.constants import FilterLookups

class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        {
            'filter': 'title',
            'lookups': {FilterLookups.EQ, FilterLookups.LIKE, FilterLookups.I_LIKE}
        },
    )
```

ordering

You can allow users to sort by a specific filter using the `ordering` property:

```
class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'published_at',
            'ordering': True,
        },
    )
```

On such filter you can sort in ascending order giving:

```
GET /books?ordering(published_at)
```

To sort in descending order you can use the − symbol:

```
GET /books?ordering(-published_at)
```

Note: Ordering can only be specified for database fields.

distinct

If you want to apply a SELECT DISTINCT to the resulting queryset you can use the `distinct` property:

```
class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'published_at',
            'distinct': True,
        },
    )
```

This way, if the *published_at* fielter is present in the query, a SELECT DISTINCT will be applied.

Note: If you want to perform a *SELECT DISTINCT* regardless of which filter is involved in the query, you can do that by adding the `DISTINCT` attribute to your filter class set to True. See *`dj_rql.filter_cls.RQLFilterClass`*.

search

Search allows filtering by all properties supporting such lookups that match a given pattern.

If you want to use the `search` operator you must set the `search` property to True:

```
class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'synopsis',
            'search': True,
        },
    )
```

This way you can issue the following query:

```
GET /books?search(synopsis,murder)
```

this is equivalent to:

```
GET /books?ilike(synopsis,*murder*)
```

Note: The `search` property can be applied only to text database fields, which have the `ilike` lookup.

use_repr

For fields with choices, you may want to allow users to filter for the choice label instead of its database value, so in this case you can set the `use_repr` property to True:

```
STATUSES = (
    ('1', 'Available'),
    ('2', 'Reprint'),
    ...
)

class Book(models.Model):
    ...

    status = models.CharField(max_length=2, choices=STATUSES)


class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'status',
            'use_repr': True,
        },
    )
```

So you can filter for status like:

```
GET /books?eq(status,Available)
```

Note: `use_repr` can be used neither with `ordering` nor `search`.

source and sources

Sometimes it is better to use a name other than the field name for the filter. In this case you can use the `source` property to specify the name of the field:

```
class MyFilterClass(RQLFilterClass):

    MODEL = MyModel
    FILTERS = (
        'a_field',
        {
            'filter': 'filter_name',
            'source': 'field_name',
        },
    )
```

A typical use case is to define filters for fields on related models:

```
class BookFilters(RQLFilterClass):
```

(continues on next page)

(continued from previous page)

```
    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'author',
            'source': 'author__name',
        },
    )
```

If you want to use a filter to search in two or more fields you can use the property `sources`:

```
class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'author',
            'sources': ('author__name', 'author__surname'),
        },
    )
```

dynamic and field

`django-rql` allows to filter for dynamic fields (aggregations and annotations).

Suppose you have an initial queryset like:

```
queryset = Book.objects.annotate(num_authors=Count('authors'))
```

And you want to allows to filter by the number of authors that contribute to the book, you can do that by setting the `dynamic` property to True and specify the data type for the *num_authors* column through the `field` property:

```
class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'num_authors',
            'dynamic': True,
            'field': models.IntegerField(),
        },
    )
```

So you can write queries like this:

```
GET /books?ge(num_authors,2)
```

And obtain all the books that have two or more authors.

null_values

In some circumstances you may have some of the values for a field that you would like to consider equivalent to a database NULL.

In this case you can specify which values can considered equivalent to NULL so you can use the `null()` contant to filter:

```
from dj_rql.filter_cls import RQLFilterClass, RQL_NULL

class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'filter': 'isbn',
            'null_values': {RQL_NULL, '0-0000000-0-0'}
        },
    )
```

So if you issue the following query:

```
GET /books?eq(isbn,null())
```

The resulting queries will contains both records where the *isbn* column is NULL and records that has the *isbn* column equal to *0-0000000-0-0*.

namespace

You can allow users to filter by fields on related models. Namespaces allow to do that and is usefull for api consistency.

Consider the following filter class:

```
class Author(models.Model):
    name = models.CharField(max_length=50)
    surname = models.CharField(max_length=100)

class Book(models.Model):
    title = models.CharField(max_length=255)
    autor = models.ForeignKey(Author, on_delete=models.CASCADE)

class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        'title',
        {
            'namespace': 'author',
            'filters': ('name', 'surname'),
        },
    )
```

With this filters definition you can filter also for author's name and surname the following way:

```
GET /books?and(eq(author.name,Ken),eq(author.surname,Follett))
```

custom

Sometimes you may want to apply your specific filtering logic for a filter.

To do so, you have to set the `custom` property for that filter to True and override the `build_q_for_custom_filter` method of your filter class.

```
class BookFilters(RQLFilterClass):

    MODEL = Book
    FILTERS = (
        {
            'filter': 'title',
            'custom': True,
        },
    )

    def build_q_for_custom_filter(self, filter_name, operator, str_value, **kwargs):
        pass  #  Put your filtering logic here and return a ``django.db.models.Q``␣
↪object.
```

1.2.3 Django Rest Framework extensions

Pagination

django-rql supports pagination for your api view through the *`dj_rql.drf.paginations.RQLLimitOffsetPagination`*.

OpenAPI specifications

If you are using *django-rql* with Django Rest Framework to expose filters for your REST API, the `openapi` property allow you to describe the filter as long as control how specs for that filter will be generated.

```
'openapi': {
    'description': 'Good description',
}
```

Additional properties are:

- `required`: You can do a filter mandatory by set it to True.
- `deprecated`: You can mark a filter as deprecated set it to True.
- `hidden`: Set it to True if you don't want this filter to be included in specs.
- `type`: Allow overriding the filter data type inferred by the underlying model field.
- `format`: Allow overriding the default field format inferred by the underlying model field.

For the `type` and `format` attributes please refers to the Data Types section of the OpenAPI specifications.

1.3 The "Power of Select"

1.3.1 The `select` operator

The `select` operator is very powerful and is expecially useful for REST APIs.

Suppose you have the following models:

```
class Category(models.Model):
    name = models.CharField(max_length=100)

class Company(models.Model):
    name = models.CharField(max_length=100)
    vat_number = models.CharField(max_length=15)

class Product(models.Model):
    name = models.CharField(max_length=100)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
    manufacturer = models.ForeignKey(Company, on_delete=models.CASCADE)
```

and the following filter class:

```
from dj_rql.filter_cls import RQLFilterClass
from dj_rql.qs import SelectRelated

class ProductFilters(RQLFilterClass):

    MODEL = Product
    SELECT = True
    FILTERS = (
        'name',
        {
            'namespace': 'category',
            'filters': ('name',),
            'qs': SelectRelated('category'),
        },
        {
            'namespace': 'manufacturer',
            'filters': ('name', 'vat_number'),
            'hidden': True,
            'qs': SelectRelated('manufacturer'),
        }
    )
```

Issuing the following query:

```
GET /products?ilike(name,*rql*)
```

Behind the scenes *django-rql* applies a `select_releted` optimization to the queryset to retrive the category of each product doing a SQL JOIN.

Since the *manufacturer* has been declared `hidden` django-rql doesn't retrive the related manufacturer unless you write:

```
GET /products?ilike(name,*rql*)&select(manufacturer)
```

if you issue such query, *django-rql* apply the `qs` database optimization so it adds a JOIN with the *Company* model to optimize database access.

The `select` operator can also be used to exclude fields so if you want to retrieve products without retrieving the associated category you can write:

```
GET /products?ilike(name,*rql*)&select(-category)
```

So the category will be not fetched.

1.3.2 Django Rest Framework support

If you are writing a REST API with Django Rest Framework, *django-rql* offers an utility mixin (dj_rql.drf.serializers.RQLMixin) for your model serializers to automatically adjust the serialization of related models depending on select.

```
from rest_framework import serializers

from dj_rql.drf.serializers import RQLMixin

from ..models import Category, Company, Product


class CategorySerializer(RQLMixin, serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = ('id', 'name')


class CompanySerializer(RQLMixin, serializers.ModelSerializer):
    class Meta:
        model = Company
        fields = ('id', 'name')


class ProductSerializer(RQLMixin, serializers.ModelSerializer):
    category = CategorySerializer()
    company = CompanySerializer()

    class Meta:
        model = Product
        fields = ('id', 'name', 'category', 'company')
```

Note: A complete working example of how the `select` operator works can be found at:

https://github.com/maxipavlovic/django_rql_select_example.

1.4 API Reference

1.4.1 Default lookups by field type

Model fields	Lookup operators
AutoField	`eq`, `ne`, `ge`, `gt`, `le`, `lt`, `in`, `out`
BigAutoField	
BigIntegerField	
DateField	
DateTimeField	
DecimalField	
FloatField	
IntegerField	
PositiveIntegerField	
PositiveSmallIntegerField	
SmallIntegerField	
BooleanField	`eq`, `ne`
NullBooleanField	
CharField	`eq`, `ne`, `in`, `out`, `like`, `ilike`
EmailField	
SlugField	
TextField	
URLField	
UUIDField	

1.4.2 Constants

class `dj_rql.constants.`**`FilterLookups`**

`EQ = 'eq'`
Equal operator

`NE = 'ne'`
Not equal operator

`GE = 'ge'`
Greater or equal operator

`GT = 'gt'`
Greater than operator

`LE = 'le'`
Less or equal operator

`LT = 'lt'`
Less then operator

`IN = 'in'`
In operator

`OUT = 'out'`
Not in operator

NULL = 'null'
null operator

LIKE = 'like'
like operator

I_LIKE = 'ilike'
Case-insensitive like operator

classmethod numeric(*with_null=True*)
Returns the default lookups for numeric fields.

Parameters **with_null** (*bool, optional*) – if true, includes the *null* lookup, defaults to True

Returns a set with the default lookups.

Return type set

classmethod string(*with_null=True*)
Returns the default lookups for string fields.

Parameters **with_null** (*bool, optional*) – if true, includes the *null* lookup, defaults to True

Returns a set with the default lookups.

Return type set

classmethod boolean(*with_null=True*)
Returns the default lookups for boolean fields.

Parameters **with_null** (*bool, optional*) – if true, includes the *null* lookup, defaults to True

Returns a set with the default lookups.

Return type set

1.4.3 Exceptions

exception dj_rql.exceptions.**RQLFilterError**(*details=None*)
Base class for RQL errors.

exception dj_rql.exceptions.**RQLFilterParsingError**(*details=None*)
Parsing errors are raised only at query parsing time.

exception dj_rql.exceptions.**RQLFilterLookupError**(*details=None*)
Lookup error is raised when provided lookup is not supported by the associated filter.

exception dj_rql.exceptions.**RQLFilterValueError**(*details=None*)
Value error is raised when provided values can't be converted to DB field types.

1.4.4 Filter class

class `dj_rql.filter_cls.`**`RQLFilterClass`**(*queryset*, *instance=None*)

Base class for filter classes.

`MODEL = None`

The model this filter is for.

`FILTERS = None`

A list or tuple of filters definitions.

`DISTINCT = False`

If True, a *SELECT DISTINCT* will always be executed.

`SELECT = False`

If True, this FilterClass supports the `select` operator.

`OPENAPI_SPECIFICATION`

alias of *`dj_rql.openapi.RQLFilterClassSpecification`*

`build_q_for_custom_filter`(*data*)

Django Q() builder for custom filter.

Parameters **`data`** (*`FilterArgs`*) – Prepared filter data for custom filtering.

Return type django.db.models.Q

`build_name_for_custom_ordering`(*filter_name*)

Builder for ordering name of custom filter.

Parameters **`filter_name`** (*`str`*) – Full filter name (f.e. ns1.ns2.filter1)

Returns Django field str path

Return type str

`optimize_field`(*data*)

This method can be overridden to apply complex DB optimization logic.

Parameters **`data`** (*`OptimizationArgs`*) –

Returns Optimized queryset

Return type django.db.models.QuerySet or None

`apply_annotations`(*filter_names*, *queryset=None*)

This method is used from RQL Transformer to apply annotations before filtering on queryset, but after it's understood which filters are used. Also, it's used to apply annotations for select() optimization.

Parameters

- **`of str filter_names`** (*`set`*) – Set of filter names
- **`or None queryset`** (*`django.db.models.QuerySet`*) – Queryset for annotation

`apply_filters`(*query*, *request=None*, *view=None*)

Main entrypoint for request filtering.

Parameters

- **`query`** (*`str`*) – RQL query string
- **`request`** – Request from API view
- **`view`** – API view

Returns Lark AST, Filtered QuerySet

build_q_for_filter(*data*)
Django Q() builder for extracted from query RQL expression. In general, this method should not be overridden.

Parameters **data** (*FilterArgs*) – Prepared filter data for custom filtering.

Return type django.db.models.Q

1.4.5 DB optimization

class dj_rql.qs.**SelectRelated**(**relations*, ***kwargs*)
Apply a select_related optimization to the queryset.

class dj_rql.qs.**PrefetchRelated**(**relations*, ***kwargs*)
Apply a prefetch_related optimization to the queryset.

dj_rql.qs.**SR**
alias of *dj_rql.qs.SelectRelated*

dj_rql.qs.**PR**
alias of *dj_rql.qs.PrefetchRelated*

1.4.6 Django Rest Framework extensions

Filter backend

class dj_rql.drf.backend.**RQLFilterBackend**
RQL filter backend for DRF GenericAPIViews.

Examples:

class ViewSet(mixins.ListModelMixin, GenericViewSet): filter_backends = (RQLFilterBackend,)
rql_filter_class = ModelFilterClass

filter_queryset(*request*, *queryset*, *view*)
Return a filtered queryset.

Pagination

class dj_rql.drf.paginations.**RQLLimitOffsetPagination**(**args*, ***kwargs*)
RQL limit offset pagination.

class dj_rql.drf.paginations.**RQLContentRangeLimitOffsetPagination**(**args*, ***kwargs*)
RQL RFC2616 limit offset pagination.

Examples: Response

200 OK Content-Range: items <FIRST>-<LAST>/<TOTAL>

Serialization

OpenAPI

class dj_rql.openapi.**RQLFilterClassSpecification**

classmethod get(*filter_instance*)

Returns OpenAPI specification for filters. Filter sorting is alphabetic with deprecated filters in the end.

Parameters **filter_instance** (dj_rql.filter_cls.RQLFilterClass) – Instance of Filter Class

Returns OpenAPI compatible specification of Filter Class Filters

Return type list of dict

classmethod get_for_field(*filter_item*, *filter_instance*)

This method can be overridden to support custom specs for certain filters.

Parameters

- **filter_item** (*dict*) – Extended Filter Item
- **filter_instance** (dj_rql.filter_cls.RQLFilterClass) – Instance of Filter Class

Return type dict or None

class dj_rql.openapi.**RQLFilterDescriptionTemplate**

classmethod render(*filter_item*, *filter_instance*)

Parameters

- **filter_item** (*dict*) – Extended Filter item
- **filter_instance** (dj_rql.filter_cls.RQLFilterClass) – Instance of Filter Class

Returns Rendered description for filter item

Return type str

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

R

S