Django Natural Language Filter

Hodossy, Szabolcs

Jan 21, 2021

USER GUIDE

The goal of Django NLF is to provide a simple and easy way to express complex filtering criteria. This natural language approach enables building nested complex queries quickly for your users, which are otherwise cumbersome with other libraries.

It provides an intuitive way to start with simpler criteria, but tries not to get in the way of more advanced use cases that need regular expressions, annotations or aggregations etc.

> **Warning:** This project is still in development, please use with this in mind!

CHAPTER

ONE

INSTALLATION

Install using `pip`,

```
$ pip install django-nlf
```

And add `django_nlf` to your `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    ...
    "django_nlf",
]
```

Then you can use the `DjangoNLFilter` with a queryset and a string, containing the filter expression. Please see the *Language Reference* for more details.

```
from django_nlf.filters import DjangoNLFilter
from .models import Article

nl_filter = DjangoNLFilter()
qs = Article.objects.all()
q = "author.username is john or title ~ news"
# equivalent to Article.objects.filter(Q(author__username="user") | Q(title__
↪icontains="news"))
articles = nl_filter.filter(qs, q)

# Nested logical operators are also supported:
q = "author.username is john and (title ~ news or created_at <= 2020-06-05)"
# equivalent to
# Article.objects.filter(
#   Q(author__username="user") & (Q(title__icontains="news") | Q(created_at__lte=
↪"2020-06-05"))
# )
articles = nl_filter.filter(qs, q)
```

CHAPTER

TWO

REST FRAMEWORK INTEGRATION

You just need to simply add the natural language filter backend to your filter backends list.

```
REST_FRAMEWORK = {
  "DEFAULT_FILTER_BACKENDS": (
    ...
    "django_nlf.rest_framework.DjangoNLFilterBackend",
  ),
}
```

2.1 Language Reference

Warning: This project is in development, please expect changes in the language syntax!

Warning: Only the *a-z, A-Z, 0-9, '.', '_', '-', '/', ':'* characters are supported by the language right now.

2.1.1 Terminology

The whole natural language expression, as a string, is referred to as *filter expression*. The atomic building blocks of the language are the *expressions* which can be composed via *operators* and grouped together to articulate the filtering criteria.

The most simple *expression* is of the following form:

```
#<field name>  <lookup> <value>
"title         contains science"
```

Note: As a convenience, an expression targeting a boolean field can take the following form: `"is archived"` or negated as `"is not archived"`, where the last part is the field name.

Fields

All fields are available for a given model, including relationships as well. You can follow each path with the *path separator*, by default it looks like

```
"author.username contains john"
```

Values

Values can be anything, but if you need whitespace in it, you must quote the value. For some *lookups*, a list of values can be defined as well. List of values are defined as a coma separated list within parenthesis. Regular expressions can be defined between two forward slashes.

```
'title contains "science news"'
"author.username is in (john, jane)"
"payment_details matches /[\d]{4}(-[\d]{4}){3}/"
```

Complicating things

These *expressions* can then be combined in any way with logical operators. The precedence of the operators are respected, i.e. *and* has higher precedence over *or*.

```
"title contains science and published > 2020-01-01"
```

You can group these expressions as well:

```
"title contains science and (author is john or published > 2020-01-01)"
```

Note: You can nest these groups as you like.

Advanced Use

To express the most complicated filtering criteria, functions can be used in the language as a *field*, a *value* or an *expression*. On how to develop such functions, see the *Writing your own function Guide*.

For example if we have an *articles* table for a science site, we could do the following, where `hasBeenPeerReviewed()` hides a nasty join detail to check if a submitted paper has already been reviewed.

Some functions are available by default. More info on the *Available functions*

```
q1 = "author is john and hasBeenPeerReviewed()"
q2 = "published > startOfYear()"
```

See Also

Supported Lookups

Note: The lookups are all case insensitive.

Equals

Can be expressed as `is`, `equals` or =, and means a case insensitive equality check. Can be negated as `is not`, `do(es) not equal` or != respectively.

Contains

Can be expressed as `contains`, and means a case insensitive check. Can be negated as `do(es) not contain`.

Regex

Can be expressed as `matches`, and means a case insensitive regular expression match. Can be negated as `do(es) not match`.

In

Can be expressed as `in`, and means a case sensitive equality check against the given list of values. Can be negated as `not in`.

Greater than (or equal)

Can be expressed as > and >=, and means a comparison against the given value.

Lower than (or equal)

Can be expressed as < and <=, and means a comparison against the given value.

Note: Custom lookups are not supported currently.

Supported Operators

Note: The operators are all case insensitive.

And

Can be expressed as `and`.

Or

Can be expressed as `or`.

Not

Can be expressed as `not` in front of *functions* and group of *expressions*. *Expressions* can be negated by negating the *lookup* (e.g. `is` -> `is not`).

Groups

Expressions can be grouped by parenthesis: `(`, `)`.

Available Functions

Functions can be used in three ways: as a *field*, a *value* or a whole *expression*.

Date functions

Default date functions

`django_nlf.functions.dates.`**`start_of_month`**`(`**args*, ***kwargs*`)`
Determines the first day of the current month. Time is set to *00:00:00*.

Returns A datetime object set to 00:00 on the first day of the current month.

Return type `datetime`

`django_nlf.functions.dates.`**`start_of_week`**`(`**args*, ***kwargs*`)`
Determines the first day of the the current week based on l10n settings. Time is set to *00:00:00*.

Returns A datetime object set to 00:00 on the first day of the current week.

Return type `datetime`

`django_nlf.functions.dates.`**`start_of_year`**`(`**args*, ***kwargs*`)`
Determines the first day of the current year. Time is set to *00:00:00*.

Returns A datetime object set to 00:00 on the first day of the current year.

Return type `datetime`

2.2 Customization

2.2.1 Converting field names

You may not like the snake case convention widely used in Python to be used in the filtering expressions your user write. Therefore you can use one of the built in case coverters or write your own.

Built in converters

`django_nlf.utils.`**`camel_to_snake_case`**(*value: str*) → str

Converts strings in camelCase to snake_case.

Parameters **`value`** (*str*) – camalCase value.

Returns snake_case value.

Return type str

Custom converter

To support automatic case conversion, a custom implementation can be provided.

```
# app/utils.py
def my_converter(field_name: str) -> str:
    # do something with field_name
    return field_name
```

and in `settings.py`:

```
NLF_FIELD_NAME_CONVERTER = "app.utils.my_converter"
```

2.2.2 Writing your own function

To create your own custom function, you just need to register it. The first argument to `nlf_function` will determine how the function can be referenced in *filter expressions*, the role parameters determines where the function can be used, while the model parameter can be used to restrict usability to certain models.

See the following example:

```
from django_nlf.functions import nlf_function

@nlf_function("myFunction")
def my_function(*args, **kwargs):
  pass
```

The arguments are passed as strings as positional arguments, only quotes are removed. Additional context is available through key word arguments.

Currently the `Model class` being filtered, the `Request` and the `View` are passed as `model`, `request` and `view` respectively.

Value functions

If the function is used as a value, it can return anything appropriate for the field.

Field functions

If the function is used as a field, it must return a dictionary with a single key, the field name, and an annotation. This can be an `F object`, `Aggragation`, `Subquery` or even a `Window`.

> **Warning:** Annotations are applied **BEFORE** all other filtering is done, therefore if you need to filter on a group, that must be handled as an expression function with a `Subquery`.

Expression functions

Expression functions are passed an additional keyword argument `exclude` to specify if the function has been negated (`exclude=True`) or not (`exclude=False`). It must return a tuple of a dictionary holding annotations as for field functions and a `Q object`.

> **Warning:** Annotations are applied **BEFORE** all other filtering is done, therefore if you need to filter on a group, that must be handled as an expression function with a `Subquery`.

2.3 Configuration

Here is a list of all available settings of django-nlf and their default values. All settings are prefixed with `NLF_`.

2.3.1 NLF_EMPTY_VALUE

Default: `"EMPTY"`

The string that is translated to a lookup with the `NULL` database value.

2.3.2 NLF_FALSE_VALUES

Default: `("0", "f")`

Used in boolean coercion to determine the boolean value of a string. If the first character of the value coerced to boolean matches any listed character, the value is considered `False`, otherwise `True`.

2.3.3 NLF_FIELD_NAME_CONVERTER

Default: `None`

A function or an import path to a function that applies a conversion to the field name. Can be used to automatically convert between cases, e.g. *camelCase* to *snake_case*.

One such converter function is readily available as `django_nlf.utils.camel_to_snake_case`.

2.3.4 NLF_FIELD_SHORTCUTS

Default: `{}`

A simple mapping of models and field name shortcuts to full field path. The key must be a model identifier in a form of `app.Model` and its value is mapping of shortcut to full path. The special key `__all__` applies to all models, and has a lower precedence. For example if you would like to identify you users by their username in the language for your model `Article`, and you have an `author` field on your model (pointing to the Primary Key of the users), you can do the following:

```
NLF_FIELD_SHORTCUTS = {
    "blog.Article": {"author": "author.username"},  # Do this for shortcuts for a␣
↪specific model
    "__all__": {}                                   # Do this for generic shortcuts,␣
↪applicable to all models
}
```

2.3.5 NLF_PATH_SEPARATOR

Default: `"."`

The character that separates path elements for fields. Used when applying filter following Foreign Key or Many-to-Many relations.

2.3.6 NLF_QUERY_PARAM

Default: `"q"`

This applies to the Django RESTFramework Backend. This parameter is used for extracting the filter expression from the `GET` query parameters.

2.4 Get the source code

Please first fork the repository, then clone it. Every Pull Request is more than welcome!

Note: The following assumes that you have `git`, `Python 3.7+`, `virtualenv` and optionally `make` installed.

2.5 Setup

It is advised to develop a project in a virtual environment.

```
$ python -m virtualenv venv
```

All development dependencies are listed in `dev_requirements.txt`.

```
$ pip install -r dev_requirements.txt
```

A `Makefile` is available with the most common operations that are needed during development. The following targets are available:

- `make lint`: Runs black in check mode and pylint
- `make format`: Runs black and formats each file
- `make test`: Runs the test suite
- `make coverage`: Runs the test suite and measures coverage
- `make docs`: Builds the documentation
- `make publish`: Builds and publishes the package. Should not be used, the same happens for tag creation as a Github Action.
- `make lang`: Builds the language with a listener. See below for further information.

If you are not familiar with GNU Make, use that file as reference on how to perform each operation.

2.6 Improving the language

When changing the grammar file, the whole runtime should be re-generated by running `make lang`. That however overwrites some files and introduces a lot of `pylint` messages. It also generates a new `django_nlf/antlr/generated/DjangoNLFListener.py` file. There are several things that should be done (currently manually, but any automation is greatly appreciated here):

- If there were a new parser rule added, the corresponding `enter*` and `exit*` functions must be implemented in `django_nlf/antlr/listener.py`. After that the generated file can be safely deleted.
- The introduces errors should be corrected, which most of the cases can be simply done by reseting the appropriate changes in the files, namely:
  - reset the changes made to the import section of the generated `*.py` files
  - remove unnecessary `pass` statements
  - remove `else:` after `return` cases
  - reformat some `while` and `if` conditions according to `pylint` suggestions

The test suite then can verify if old functionality has been kept intact.

2.7 Focus points

2.7.1 Functions

Functions are now an experimental feature, and needs a hell lot more testing, and feedback from real life usage. Any reported issue or feature idea is greatly appreciated.

2.7.2 Autocomplete

The biggest bottleneck for quickly introducing the language for your end users is the lack of Autocomplete functionality for the form fields. Any idea on how the supporting APIs should look and how the JS implementation should work is greatly appreciated.

2.7.3 Language housekeeping

The language was developed with no prior language engineering experience, so probably a lot of rationalizations and refactors can be made.

2.7.4 MyPy integration

Typing support is better with every release, but `mypy` integration is still missing.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX