Django URL Filter Documentation

Release 0.1

Miroslav Shubernetskiy

August 28, 2015

Contents

CHAPTER 1

Contents

1.1 url_filter

1.1.1 url_filter package

Subpackages

url_filter.backends package

Submodules

url_filter.backends.django module

url_filter.filtersets package

Submodules

url_filter.filtersets.base module

url_filter.filtersets.django module

url_filter.integrations package

Submodules

url_filter.integrations.drf module

Submodules

url_filter.exceptions module

exception `url_filter.exceptions.`**`SkipFilter`**
Bases: `exceptions.Exception`

Exception to be used when any particular filter within the `FilterSet` should be skipped.

Possible reasons for skipping the field:

- filter lookup config is invalid (e.g. using wrong field name - field is not present in filter set)
- filter lookup value is invalid (e.g. submitted "a" for integer field)

url_filter.fields module

class `url_filter.fields.`**`MultipleValuesField`**(*child=<class 'django.forms.fields.CharField'>*, *min_values=2*, *max_values=None*, *many_validators=None*, *delimiter=u','*, **args*, ***kwargs*)

Bases: `django.forms.fields.CharField`

Custom Django field for validating/cleaning multiple values given in a single value separated by a delimiter.

Parameters

- **`child`** (*Field, optional*) – Another Django form field which should be used.
- **`min_values`** (*int, optional*) – Minimum number of values which must be provided. By default at least 2 values are required.
- **`max_values`** (*int, optional*) – Maximum number of values which can be provided. By default no maximum is enforced.
- **`max_validators`** (*list, optional*) – Additional validators which should be used to validate all values once wplit by the delimiter.
- **`delimiter`** (*str, optional*) – The delimiter by which the value will be split into multiple values. By default `,` is used.

`clean`(*value*)

Custom `clean` which first validates the value first by using standard `CharField` and if all passes, it applies similar validations for each value once its split.

`many_run_validators`(*values*)

Run each validation from `many_validators` for the cleaned values.

`many_to_python`(*value*)

Method responsible to split the value into multiple values by using the delimiter and cleaning each one as per the child field.

`many_validate`(*values*)

Hook for validating all values.

url_filter.filters module

class `url_filter.filters.`**`Filter`**(*source=None*, **args*, ***kwargs*)

Bases: `object`

Filter class which main job is to convert leaf `LookupConfig` to `FilterSpec`.

Each filter by itself is meant to be used a "field" in the `FilterSpec`.

Parameters

- **`source`** (*str*) – Name of the attribute for which which filter applies to within the model of the queryset to be filtered as given to the `FilterSet`.

- **form_field** (*Field*) – Instance of Django's `forms.Field` which will be used to clean the filter value as provided in the queryset. For example if field is `IntegerField`, this filter will make sure to convert the filtering value to integer before creating a `FilterSpec`.
- **lookups** (*list, optional*) – List of strings of allowed lookups for this filter. By default all supported lookups are allowed.
- **default_lookup** (*str, optional*) – If the lookup is not provided in the querystring lookup key, this lookup will be used. By default `exact` lookup is used. For example the default lookup is used when querystring key is `user__profile__email` which is missing the lookup so `exact` will be used.
- **is_default** (*bool, optional*) – Boolean specifying if this filter should be used as a default filter in the parent `FilterSet`. By default it is `False`. Primarily this is used when querystring lookup key refers to a nested `FilterSet` however it does not specify which filter to use. For example lookup key `user__profile` intends to filter something in the user's profile however it does not specify by which field to filter on. In that case the default filter within profile `FilterSet` will be used. At most, one default filter should be provided in the `FilterSet`.

bind(*name*, *parent*)
Bind the filter to the filterset.

This method should be used by the parent `FilterSet` since it allows to specify the parent and name of each filter within the filterset.

clean_value(*value*, *lookup*)
Clean the filter value as appropriate for the given lookup.

Parameters

- **value** (*str*) – Filter value as given in the querystring to be validated and cleaned by using appropriate Django form field
- **lookup** (*str*) – Name of the lookup

See also:

get_form_field()

components
List of all components (source names) of all parent filtersets.

get_form_field(*lookup*)
Get the form field for a particular lookup.

This method does not blindly return `form_field` attribute since some lookups require to use different validations. For example for if the `form_field` is `CharField` but the lookup is `isnull`, it makes more sense to use `BooleanField` as form field.

Parameters **lookup** (*str*) – Name of the lookup

Returns Instantiated form field appropriate for the given lookup.

Return type Field

get_spec(*config*)
Get the `FilterSpec` for the provided `config`.

Parameters **config** (LookupConfig) – Lookup configuration for which to build `FilterSpec`. The lookup should be a leaf configuration otherwise `ValidationError` is raised.

Returns spec constructed from the given configuration.

Return type *FilterSpec*

root
This gets the root filterset.

source
Source field/attribute in queryset model to be used for filtering.

This property is helpful when `source` parameter is not provided when instantiating `Filter` since it will use the filter name as it is defined in the `FilterSet`. For example:

```
>>> class MyFilterSet(FilterSet):
...     foo = Filter(form_field=CharField())
...     bar = Filter(source='stuff', form_field=CharField())
>>> fs = MyFilterSet()
>>> print(fs.fields['foo'].source)
foo
>>> print(fs.fields['bar'].source)
stuff
```

url_filter.utils module

class `url_filter.utils.`**FilterSpec**(*components*, *lookup*, *value*, *is_negated=False*)
Bases: `object`

Class for describing filter specification.

The main job of the `FilterSet` is to parse the submitted lookups into a list of filter specs. A list of these specs is then used by the filter backend to actually filter given queryset.

The reason why filtering is decoupled from the `FilterSet` is because this allows to implement filter backends not related to Django.

components
list

A list of strings which are names of the keys/attributes to be used in filtering of the queryset. For example lookup config with key `user__profile__email` will be components of ‘‘[’user’, ‘profile’, ‘email’].

lookup
str

Name of the lookup how final key/attribute from `components` should be compared. For example lookup config with key `user__profile__email__contains` will have a lookup `contains`.

value
Value of the filter.

is_negated
bool, optional

Whether this filter should be negated. By default its `False`.

class `url_filter.utils.`**LookupConfig**(*key*, *data*)
Bases: `object`

Lookup configuration which is used by `FilterSet` to create a `FilterSpec`.

The main purpose of this config is to allow the use if recursion in `FilterSet`. Each lookup key (the keys in the querystring) is parsed into a nested one-key dictionary which lookup config stores.

For example the querystring:

```
?user__profile__email__endswith=gmail.com
```

is parsed into the following config:

```
{
    'user': {
        'profile': {
            'email': {
                'endswith': 'gmail.com'
            }
        }
    }
}
```

key
str

Full lookup key from the querystring. For example `user__profile__email__endswith`

data
dict, str

Either:

- nested dictionary where the key is the next key within the lookup chain and value is another `LookupConfig`
- the filtering value as provided in the querystring value

Parameters

- **key** (*str*) – Full lookup key from the querystring.
- **data** (*dict, str*) – A regular vanilla Python dictionary. This class automatically converts nested dictionaries to instances of LookupConfig. Alternatively a filtering value as provided in the querystring.

as_dict()
Converts the nested `LookupConfig``s to a regular ``dict`.

is_key_value()

name
If the `data` is nested `LookupConfig`, this gets its first lookup key.

value
If the `data` is nested `LookupConfig`, this gets its first lookup value which could either be another `LookupConfig` or actual filtering value.

class `url_filter.utils.`**SubClassDict**
Bases: `dict`

Special-purpose `dict` with special getter for looking up values by finding matching subclasses.

This is better illustrated in an example:

```
>>> class Klass(object): pass
>>> class Foo(object): pass
>>> class Bar(Foo): pass
>>> mapping = SubClassDict({
...     Foo: 'foo',
...     Klass: 'klass',
```

```
... })
>>> print(mapping.get(Klass))
klass
>>> print(mapping.get(Foo))
foo
>>> print(mapping.get(Bar))
foo
```

get(*k*, *d=None*)
If no value is found by using Python's default implementation, try to find the value where the key is a base class of the provided search class.

url_filter.validators module

class url_filter.validators.**MaxLengthValidator**(*limit_value*, *message=None*)
Bases: django.core.validators.MaxLengthValidator

Customer Django max length validator with better-suited error message

clean(*x*)

code = u'max_length'

compare(*a*, *b*)

deconstruct(*obj*)
Returns a 3-tuple of class import path, positional arguments, and keyword arguments.

message = <django.utils.functional.__proxy__ object>

class url_filter.validators.**MinLengthValidator**(*limit_value*, *message=None*)
Bases: django.core.validators.MinLengthValidator

Customer Django min length validator with better-suited error message

clean(*x*)

code = u'min_length'

compare(*a*, *b*)

deconstruct(*obj*)
Returns a 3-tuple of class import path, positional arguments, and keyword arguments.

message = <django.utils.functional.__proxy__ object>

1.2 Indices and tables

- genindex
- modindex
- search

u

V