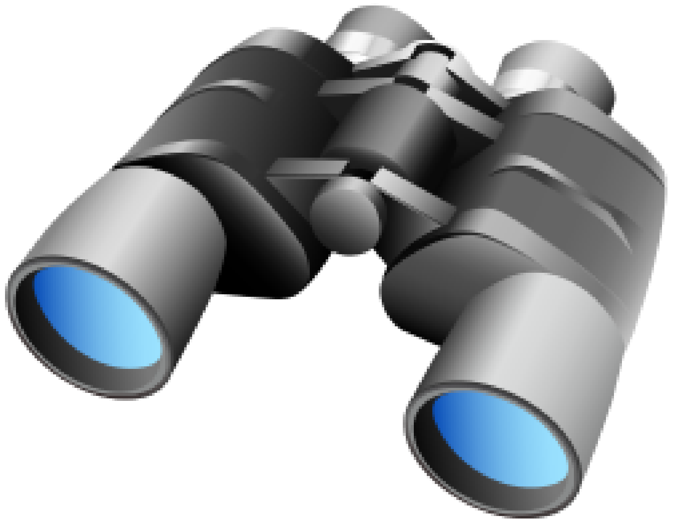
django-find Documentation

Release 0.3.4

Samuel Abels

Feb 19, 2020

Contents



CHAPTER 1

What is django-find?

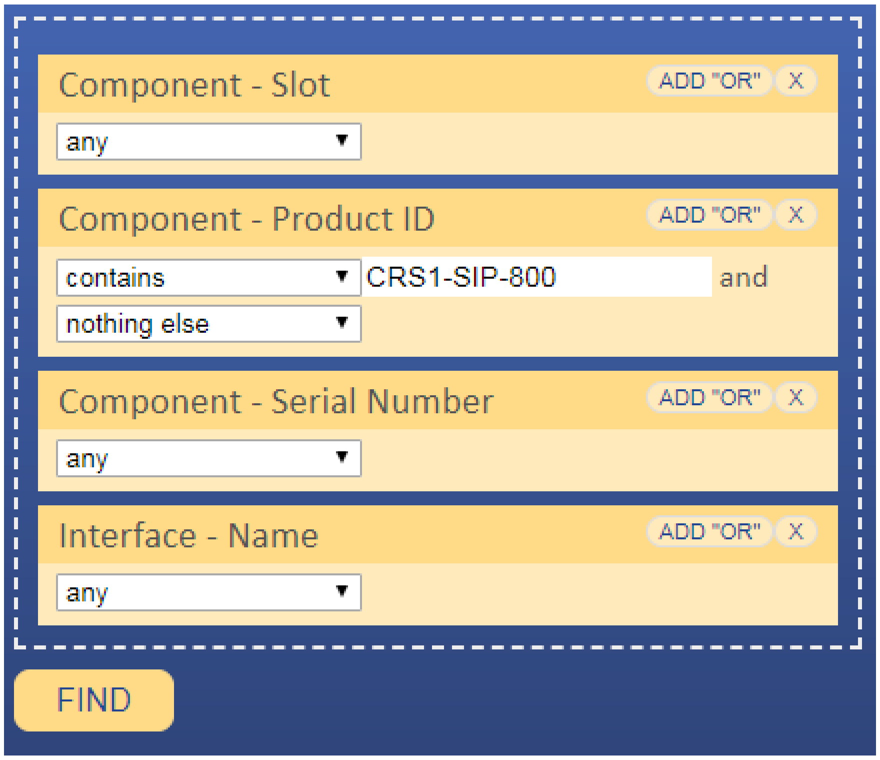
django-find is a Django app that makes it easy to add complex search functionality for the models in your project.

django-find supports two different ways to search your Django models: Query-based, or JSON-based.

By query-based, we mean that you can use statements like these to search your model:

```
author:"robert frost" and (title:road or chapter:2)
```

To make it easy to do complex searches spanning multiple models, another method is provided. For example, you may want to allow for custom searches that let the user choose which models and columns to include. In other words, a user interface like this:



For this, a JSON-based search functionality is provided:

```
{
    "Author":{"name":[[["equals","test"]]]},
    "Book": {"title":[[["notcontains","c"]]]},
    "Chapter": {"content":[[["startswith","The "]]]}
}
```

django-find is smart in figuring out how to join those models together and return a useful result.

django-find also provides a template tag that you can use to render a search field:

```
{% load find_tags %}
{% find object_list %}
{% for obj in object_list %}
    {{ obj.name }}
{% endfor %}
```

CHAPTER 2

What isn't django-find?

django-find is not a full text search engine, it searches the fields of your models. In other words, it searches and provides tabular data.

2.1 Contents

2.1.1 Installation

Prerequisites

django-find requires Python 2.7 or Python 3.5 or greater.

Getting started

Download and install the module

Download and install using PIP:

```
sudo pip3 install django-find
```

Alternatively, you may also install the latest development version from GitHub:

```
git clone git://github.com/knipknap/django-find
cd django-find
sudo make install
```

Add it to your Django project

Add "django_find" to your `INSTALLED_APPS` setting like this:

```
INSTALLED_APPS = [
    ...
    'django_find',
]
```

Make sure that the request object is available to templates!

If you haven't already, you should also install Django's django.template.context_processors.request and django.template.context_processors.i18n.

In other words, your settings need to set the TEMPLATES variable to include the context_processors like so:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            # ...
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # ...
                'django.template.context_processors.i18n',
                'django.template.context_processors.request',
            ],
        },
    },
]
```

Add it to your models

You are now ready to start using the Searchable mixin. For more information, please continue with the *tutorial*.

Running the automated test suite

If you installed from GitHub, you can run the integrated test suite:

```
make tests
```

There shouldn't be any errors, so if something comes up, please file a bug.

2.1.2 Tutorial

Introduction

We'll assume that django-find is already installed, and added to your Django project. The instructions are *here*.

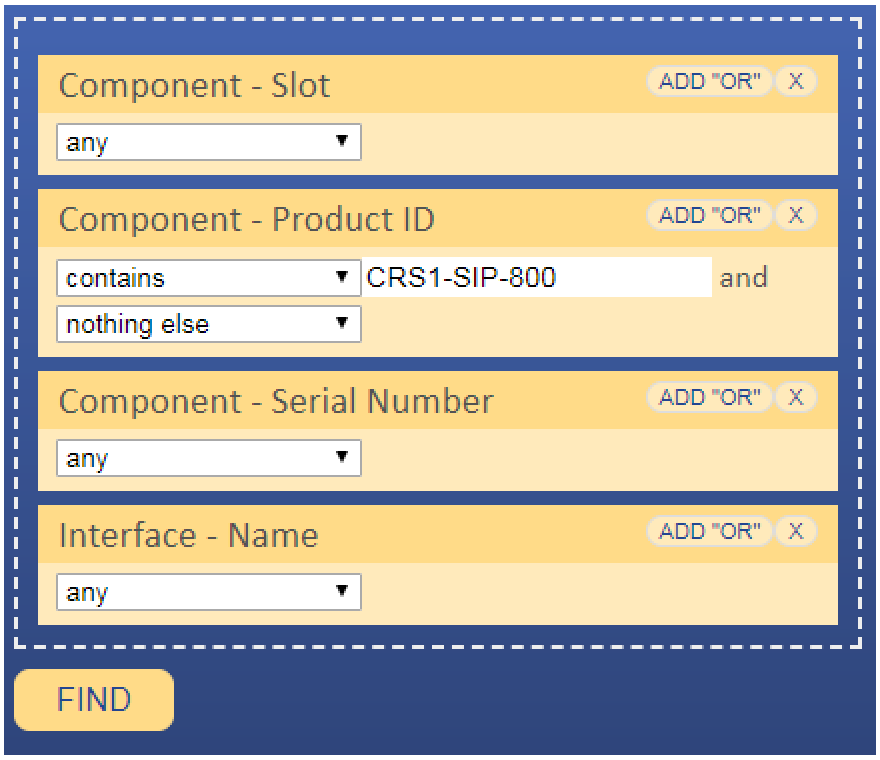
Motivation

Assume you want to add a search box to your user interface, where your users can search your models using a simple query language. For example:

- `hello world` (searches all fields for hello and world)
- `robert OR title:road` (searches all fields for "robert", and "title" for "road")

The documentation of the query language is *here*.

Alternatively, you may want to allow the user to specify the models and columns to display with a UI like this:



django-find supports JSON-based queries for this purpose.

Getting started

Enabling the functionality is as simple as adding the "Searchable" mixin to your models. Example:

```
from django.db import models
from django_find import Searchable

class Author(models.Model, Searchable):
    name = models.CharField("Author Name", max_length=10)

class Book(models.Model, Searchable):
    author = models.ForeignKey(Author, on_delete=models.CASCADE, verbose_name='Author
↪')
    title = models.CharField("Title", max_length=80)
    rating = models.IntegerField("Rating")
    internal_id = models.CharField(max_length=10)
```

That is all, you are now ready to query your models using your own code, or in your templates.

Query from your own code

All models having the Searchable mixin added provide the following methods:

```
# Query-based search returns a standard Django QuerySet that you
# can .filter() and work with as usual.
query = Book.by_query('author:"robert frost" and title:"the road"')

# You can also get a Django Q object for the statements.
q_obj = Book.q_from_query('author:"robert frost" and title:"the road"')

# JSON-based search exhausts what Django's ORM can do, so it does
# not return a Django QuerySet, but a row-based PaginatedRawQuerySet:
query, field_list = Book.by_json_raw('''{
    "Chapter": {"title":[[["contains","foo"]]]}
}''')
print('|'.join(field_list))
for row in query:
    print('|'.join(row))
```

You can pass the PaginatedRawQuerySet to Django templates as you would with a Django QuerySet, as it supports slicing and pagination.

In most cases, you also want to specify some other, related fields that can be searched, or exclude some columns from the search. The following example shows how to do that:

```
class Book(models.Model, Searchable):
    author = models.ForeignKey(Author, on_delete=models.CASCADE, verbose_name='Author
↪')
    title = models.CharField("Title", max_length=10)
    rating = models.IntegerField("Rating")
    internal_id = models.CharField(max_length=10)

    searchable = [
        ('author', 'author__name'),  # Search the name instead of the id of the␣
↪related model. Note the selector syntax
        ('stars', 'rating'),         # Add an extra alias for "rating" that can be␣
↪used in a query.
        ('internal_id', False),      # Exclude from search
    ]
```

In other words, add a "searchable" attribute to your models, that lists the aliases and maps them to a Django field using Django's selector syntax (underscore-separated field names).

Query from within templates

Using the template tag

django-find also provides a template tag that you can use to render a search field:

```
{% load find_tags %}
{% find object_list %}
{% for obj in object_list %}
    {{ obj.name }}
{% endfor %}
```

You will probably want to use this together with dj-pagination like so:

```
{% load find_tags %}
{% load pagination_tags %}
```

(continues on next page)

(continued from previous page)

```
{% find object_list %}
Found {{ object_list.count }} results.

{% autopaginate object_list %}
<table>
{% for obj in object_list %}
    <tr><td>{{ obj.name }}</td></tr>
{% endfor %}
</table>

{% paginate %}
```

Using provided templates

django-find comes with some templates that you may find useful:

```
{% include 'django_find/headers.html' with object_list=author.objects.all %}
```

This produces a `<tr>` that contains the column headers as returned by `Searchable.table_headers()`, e.g.:

```
<tr>
<th>Name</th><th>The title</th><th>Comment</th><th>Stars</th>
</tr>
```

Custom field types

To support your own field types, check the documentation for handlers.

2.1.3 The django-find Query Language

Introduction

In this chapter, we explain the query language that can be passed to `Searchable.by_query()`.

For example:

- `hello world` (searches all fields for hello and world)
- `robert OR title:road` (searches all fields for “robert”, and “title” for “road”)

The basics

To search all available fields, simply enter a word. For example, the following query searches all fields for “test”:

```
test
```

When using multiple words, the query returns all entries that match all of the words. In other words, the following query returns all entries that have both, ‘foo’ in any field AND ‘bar’ in any field:

```
foo bar
```

To search for strings including whitespace, use double quotes. The following string returns all models that have a field containing "foo bar" (without quotes):

```
"foo bar"
```

Search individual fields

To limit your search to a specific field, you can use the following syntax:

```
author:robert
author:"robert frost"
author:robert author:frost title:road
```

Limiting a search to the beginning or end of a string

To search a string in a particular location of a field, use the ^ and $ operators. To search at the beginning, use:

```
^test
author:^robert
```

To search a string at the end of a field, use:

```
test$
author:frost$
```

To look for an exact match, use either both, ^ and $, or use an equal sign (=) instead. The following queries all look for an exact match:

```
^test$
author:^frost$
author=frost
author:"^robert frost$"
author="robert frost"
```

Boolean operators

Boolean AND

When you specify multiple words, django-find by default returns all entries that match all of the words. In other words, django-find behaves like a boolean AND. The following queries are all equivalent:

```
foo bar
foo AND bar
foo and bar
"foo" and "bar"
```

Boolean OR

You can also use boolean OR operators. Here are some examples:

```
"robert frost" OR "mark twain"
robert or mark
^robert or twain$ or foo or title:test
author:^robert or author:twain$
```

Boolean NOT

To find fields that DON'T match a particular string, use NOT:

```
"robert frost" not title:"the road"
"robert frost" and not title:"the road"
not foo or not bar
```

Grouping

For more complex searches, you can use brackets to group sub-expressions. Arbitrary nesting is supported:

```
author:robert and not (title:"the road" or title:"yellow")
test (name:foo and (book:one or book:two) and (chapter:10 or chapter:12 or␣
↪chapter:13))
```

Searching dates and times

Date formats

django-find accepts all formats that are supported by the dateparser python module. Some examples:

```
12/12/12
2018-01-22
"2018-01-22 10:00"
"10:40 pm"
"August 14, 2015 EST"
"1 min ago"
"2 weeks ago"
"3 months, 1 week and 1 day ago"
"in 2 days"
tomorrow
```

For a full list of supported formats, please check the dateparser documentation.

Searching for ranges

You can use them to look for time ranges. The following query returns all entries that were updated after the beginning of January 1st, 12:00am:

```
updated>=2018-1-1
```

Similarly, you can get the entries that were updated before 2018:

```
updated<2018-1-1
```

To look for a range, use AND:

```
updated>=2018-1-1 updated<=2019-1-1
updated>=2018-1-1 AND updated<=2019-1-1
```

When searching for dates and times, the ^ and $ characters have special meanings: They are equivalent to <= and >=. In other words, the following queries are equivalent when used on a DateField or DateTimeField:

```
updated:^2018-1-1
updated>=2018-1-1
```

To look for an exact match, use both:

```
updated:"^2018-1-1 11:00$"
```

Operator list

Here is the full list of operators supported by **django-find**:

```
name=foo -> Name matching "foo" exactly
name:^foo$ -> Equivalent to the previous query
name!=foo -> Name not matching "foo" exactly
name<>foo -> Equivalent to the previous query

name:foo -> Name containing the substring "foo"
name!:foo -> Name not containing the substring "foo"
name:^foo -> Name starting with the substring "foo"
name!:^foo -> Name not starting the substring "foo"
name:foo$ -> Name ending with the substring "foo"
name!:foo$ -> Name not ending the substring "foo"

id>1 -> Greater than
id>=1 -> Greater than or equal
id=>1 -> Greater than or equal
id<5 -> Less than
id<=5 -> Less than or equal
id=>5 -> Less than or equal
id<>5 -> Unequal
```

2.1.4 django_find

django_find package

Subpackages

django_find.parsers package

Submodules

django_find.parsers.json module

class django_find.parsers.json.**JSONParser**
    Bases: `object`

Transforms a JSON string into a DOM. The DOM is identical to what QueryParser generates. Example JSON input:

```
{
    "Device":
    {
        "Hostname":
            [
                [["contains": "s-"],["contains": "-ea1"]],
                [["startswith", ""]]
            ],
        "Tags":
            [
                [["neq":"asdasd"]]
            ]
    }
    "Component":
    {
        "Slot": [[]]
    }
}
```

parse (*json_string*)

parse_criteria (*clsgroup*, *criteria*, *clsname*)

parse_operators (*termgroup*, *term*, *fieldname*)

parse_terms (*fieldgroup*, *terms*, *fieldname*)

django_find.parsers.parser module

class django_find.parsers.parser.**Parser** (*token_list*)

Bases: object

The base class for all parsers.

__init__ (*token_list*)

x.__init__(...) initializes x; see help(type(x)) for signature

django_find.parsers.query module

class django_find.parsers.query.**QueryParser** (*fields*, *default*)

Bases: *django_find.parsers.parser.Parser*

__init__ (*fields*, *default*)

Fields is a map that translates aliases to something like Book.author.

parse (*query*)

parse_and (*scopes*, *match*)

parse_boolean (*scopes*, *dom_cls*, *match*)

parse_closebracket (*scopes*, *match*)

parse_field (*scopes*, *match*)

parse_not (*scopes*, *match*)

parse_openbracket(*scopes*, *match*)

parse_or(*scopes*, *match*)

parse_whitespace(*scopes*, *match*)

parse_word(*scopes*, *match*)

django_find.parsers.query.**close_scope**(*scopes*)

django_find.parsers.query.**get_term_from_op**(*field*, *operator*, *value*)

django_find.parsers.query.**op_from_word**(*word*)

django_find.parsers.query.**open_scope**(*scopes*, *scope*)

django_find.serializers package

Submodules

django_find.serializers.django module

class django_find.serializers.django.**DjangoSerializer**(*model*)
Bases: *django_find.serializers.serializer.Serializer*

__init__(*model*)
x.__init__(...) initializes x; see help(type(x)) for signature

boolean_term(*selector*, *operator*, *data*)

date_datetime_common(*selector*, *operator*, *thedatetime*)

date_term(*selector*, *operator*, *data*)

datetime_term(*selector*, *operator*, *data*)

int_term(*selector*, *operator*, *data*)

lcstr_term(*selector*, *operator*, *data*)

logical_and(*terms*)

logical_not(*terms*)

logical_or(*terms*)

str_term(*selector*, *operator*, *data*)

term(*name*, *operator*, *data*)

django_find.serializers.serializer module

class django_find.serializers.serializer.**Serializer**
Bases: object

Base class for all serializers.

logical_group(*terms*)

logical_root_group(*root_group*, *terms*)

django_find.serializers.sql module

class django_find.serializers.sql.**SQLSerializer**(*model*, *mode=u'SELECT'*, *fullnames=None*, *extra_model=None*)

Bases: `django_find.serializers.serializer.Serializer`

`__init__`(*model*, *mode=u'SELECT'*, *fullnames=None*, *extra_model=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

boolean_term(*db_column*, *operator*, *data*)

date_datetime_common(*db_column*, *operator*, *thedatetime*)

date_term(*db_column*, *operator*, *data*)

datetime_term(*db_column*, *operator*, *data*)

int_term(*db_column*, *operator*, *data*)

lcstr_term(*db_column*, *operator*, *data*)

logical_and(*terms*)

logical_group(*terms*)

logical_not(*terms*)

logical_or(*terms*)

logical_root_group(*root_group*, *terms*)

str_term(*db_column*, *operator*, *data*)

term(*term_name*, *operator*, *data*)

django_find.serializers.util module

django_find.serializers.util.**parse_date**(*thedate*)

django_find.serializers.util.**parse_datetime**(*thedate*)

django_find.templatetags package

Submodules

django_find.templatetags.find_tags module

class django_find.templatetags.find_tags.**SearchNode**(*queryset_var*, *fields*)

Bases: django.template.base.Node

`__init__`(*queryset_var*, *fields*)
x.__init__(...) initializes x; see help(type(x)) for signature

render(*context*)

django_find.templatetags.find_tags.**find**(*parser*, *token*)

Submodules

django_find.apps module

class django_find.apps.**DjangoFindConfig**(*app_name*, *app_module*)

Bases: `django.apps.config.AppConfig`

name = 'django_find'

verbose_name = 'Django Find'

django_find.dom module

class django_find.dom.**And**(*children=None*, *is_root=False*)

Bases: *django_find.dom.Group*

auto_leave_scope()

serialize(*strategy*)

class django_find.dom.**Group**(*children=None*, *is_root=False*)

Bases: *django_find.tree.Node*

auto_leave_scope()

get_term_names()

Returns a flat list of the names of all Terms in the query, in the order in which they appear. Filters duplicates.

optimize()

serialize(*strategy*)

translate_term_names(*name_map*)

class django_find.dom.**Not**(*children=None*, *is_root=False*)

Bases: *django_find.dom.Group*

auto_leave_scope()

optimize()

serialize(*strategy*)

class django_find.dom.**Or**(*children=None*, *is_root=False*)

Bases: *django_find.dom.Group*

serialize(*strategy*)

class django_find.dom.**Term**(*name*, *operator*, *data*)

Bases: *django_find.tree.Node*

__init__(*name*, *operator*, *data*)

x.__init__(...) initializes x; see help(type(x)) for signature

dump(*indent=0*)

each(*func*, *node_type*)

Runs func once for every node in the object tree. If node_type is not None, only call func for nodes with the given type.

optimize()

serialize(*strategy*)

django_find.handlers module

class django_find.handlers.**BooleanFieldHandler**
Bases: *django_find.handlers.FieldHandler*

db_type = 'BOOL'

classmethod handles(*model, field*)

class django_find.handlers.**DateFieldHandler**
Bases: *django_find.handlers.FieldHandler*

db_type = 'DATE'

classmethod handles(*model, field*)

class django_find.handlers.**DateTimeFieldHandler**
Bases: *django_find.handlers.FieldHandler*

db_type = 'DATETIME'

classmethod handles(*model, field*)

class django_find.handlers.**FieldHandler**
Bases: object

Abstract base type for all field handlers.

A field handler is an object that you can use to define custom behavior when searching a field of a model.

You might want to use a field handler if you are using a custom model field, or if your query contains information that requires client-side processing before being passed to the database.

db_type = None

classmethod handles(*model, field*)

classmethod prepare(*value*)

class django_find.handlers.**IPAddressFieldHandler**
Bases: *django_find.handlers.LowerCaseStrFieldHandler*

classmethod handles(*model, field*)

class django_find.handlers.**IntegerFieldHandler**
Bases: *django_find.handlers.FieldHandler*

db_type = 'INT'

classmethod handles(*model, field*)

class django_find.handlers.**LowerCaseStrFieldHandler**
Bases: *django_find.handlers.StrFieldHandler*

db_type = 'LCSTR'

class django_find.handlers.**StrFieldHandler**
Bases: *django_find.handlers.FieldHandler*

db_type = 'STR'

classmethod handles(*model, field*)

django_find.model_helpers module

`django_find.model_helpers.`**`sql_from_dom`**(*cls*, *dom*, *mode='SELECT'*, *fullnames=None*, *extra_model=None*)

django_find.models module

This module contains the Searchable mixin, the main public API of django-find.

`class` `django_find.models.`**`Searchable`**

Bases: `object`

This class is a mixin for Django models that provides methods for searching the model using query strings and other tools.

`classmethod by_fullnames`(*fullnames*)

Returns a unfiltered values_list() of all given field names.

`classmethod by_json_raw`(*json_string*, *extra_model=None*)

`classmethod by_query`(*query*, *aliases=None*)

`classmethod by_query_raw`(*query*, *mode='SELECT'*, *fullnames=None*, *extra_model=None*)

Returns a PaginatedRawQuerySet for the given query.

`classmethod dom_from_query`(*query*, *aliases=None*)

`classmethod get_aliases`()

Returns a list of the aliases, that is, the names of the fields that can be used in a query.

`classmethod get_caption_from_selector`(*selector*)

`classmethod get_class_from_fullname`(*fullname*)

Given a name in the format "Model.hostname", this function returns a tuple, where the first element is the Model class, and the second is the field name "hostname".

The Model class must inherit from Searchable to be found.

`classmethod get_default_searchable`()

`classmethod get_field_from_selector`(*selector*)

Given a django selector, e.g. device__metadata__name, this returns the class and the Django field of the model, as returned by Model._meta.get_field(). Example:

```
device__metadata__name -> (SeedDevice, SeeDevice.name)
```

`classmethod get_field_handler_from_alias`(*alias*)

Given an alias, e.g. 'host', 'name', this function returns the handler.FieldHandler.

@type name: str @param name: e.g. 'address', or 'name'

`classmethod get_field_handler_from_field`(*field*)

`classmethod get_field_handler_from_fullname`(*fullname*)

Given a fullname, e.g. 'Device.host', 'Author.name', this function returns the handler.FieldHandler.

@type name: str @param name: e.g. 'address', or 'name'

`classmethod get_fullnames`()

Like get_aliases(), but returns the aliases prefixed by the class name.

`classmethod get_object_vector_for`(*search_cls_list*)

classmethod get_object_vector_to(*search_cls*)

classmethod get_primary_class_from_fullnames(*fullnames*)

classmethod get_searchable()

classmethod get_selector_from_alias(*alias*)
Given alias (not a fullname), this function returns the selector in the following form:

```
component__device__host
```

@type name: str @param name: e.g. 'address', or 'name'

classmethod get_selector_from_fullname(*fullname*)
Given a name in the form 'Unit.hostname', this function returns a Django selector that can be used for filtering. Example (assuming the models are Book and Author):

```
Book.get_selector_from_fullname('Author.birthdate')
# returns 'author__birthdate'
```

Example for the models Blog, Entry, Comment:

```
Blog.get_selector_from_fullname('Comment.author')
# returns 'entry__comment__author'
```

@type name: str @param name: The field to select for @rtype: str @return: The Django selector

classmethod q_from_query(*query*, *aliases=None*)
Returns a Q-Object for the given query.

searchable = ()

searchable_labels = {}

classmethod sql_from_json(*json_string*, *mode='SELECT'*, *extra_model=None*)

classmethod sql_from_query(*query*, *mode='SELECT'*, *fullnames=None*, *extra_model=None*)
Returns an SQL statement for the given query.

classmethod table_headers()

django_find.rawquery module

class `django_find.rawquery.`**PaginatedRawQuerySet**(*model*, *raw_query*, *args=None*, *limit=None*, *offset=None*)
Bases: `object`

__init__(*model*, *raw_query*, *args=None*, *limit=None*, *offset=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

count

query

`django_find.rawquery.`**assert_positive_slice**(*slc*)

django_find.refs module

`django_find.refs.`**child_classes**(*cls*)
Returns all models that have a foreign key pointing to cls.

django_find.refs.**get_field_to**(*cls*, *target_cls*)

django_find.refs.**get_join_for**(*vector*)

Given a vector as returned by get_object_vector_for(), this function returns a list of tuples that explain how to join the models (tables) together on the SQL layer. Each tuple has three elements:

```
(table_name, left_key, right_key)
```

In the first tuple of the list, left_key is always None. In the second tuple of the list, right_key is always None. All other tuples required both keys to join them.

Complete example (keep in mind that the connection between Component and Unit is many-to-many, so there's a helper table here):

```
get_join_path_for((Device, Component, Unit))
```

This returns:

```
[
    ('inventory_device', None, None),
    ('inventory_component', 'device_id', 'inventory_device.metadata_id'),
    ('inventory_unit_component', 'component_id', 'inventory_component.id'),
    ('inventory_unit', 'id', 'inventory_unit_component.unit_id')
]
```

Which means that the following SQL JOIN could be used:

```
SELECT *
FROM inventory_device
LEFT JOIN inventory_component ON inventory_component.device_id=inventory_device.
↪metadata_id
LEFT JOIN inventory_unit_component ON inventory_unit_component.component_
↪id=inventory_component.id
LEFT JOIN inventory_unit ON inventory_unit.id=inventory_unit_component.unit_id
```

django_find.refs.**get_object_vector_for**(*cls*, *search_cls_list*, *subtype*, *avoid=None*)

Like get_object_vector_to(), but returns a single vector that reaches all of the given classes, if it exists. Only searches classes that are subtype of the given class.

django_find.refs.**get_object_vector_to**(*cls*, *search_cls*, *subtype*, *avoid=None*)

Returns a list of all possible paths to the given class. Only searches classes that are subtype of the given class.

django_find.refs.**get_subclasses**(*cls*)

Recursively finds all subclasses of the current class. Like Python's __class__.__subclasses__(), but recursive. Returns a list containing all subclasses.

@type cls: object @param cls: A Python class. @rtype: list(object) @return: A list containing all subclasses.

django_find.refs.**parent_classes**(*cls*)

Returns all models that are referenced by a foreign key of the given class.

django_find.refs.**sort_vectors_by_primary_cls**(*vectors*, *primary_cls*)

Sort the vectors by the position of the primary class, and the vector length.

django_find.refs.**yield_all_vectors**(*search_cls_list*, *subtype*)

Yields all possible vectors between all given classes.

django_find.refs.**yield_matching_vectors**(*vectors*, *search_cls_list*)

Yields all possible vectors that connect all of the given classes. The result is sorted by the position of the primary class, and the vector length.

django_find.tree module

class django_find.tree.**Node**(*children=None*, *is_root=False*)

Bases: `object`

__init__(*children=None*, *is_root=False*)

x.__init__(...) initializes x; see help(type(x)) for signature

add(*child*)

dump(*indent=0*)

each(*func*, *node_type=None*)

Runs func once for every node in the object tree. If node_type is not None, only call func for nodes with the given type.

pop()

django_find.version module

Warning: This file is automatically generated.

2.2 Development

django-find is on GitHub.

2.3 License

django-find is published under the MIT licence.

Python Module Index

Index