
django-autocomplete-light Documentation

Release 3.5.0

James Pic & contributors

Mar 18, 2020

1	Features	3
2	Upgrading	5
3	Resources	7
4	Basics	9
4.1	Install django-autocomplete-light v3	9
4.2	django-autocomplete-light tutorial	10
4.3	Types of forwarded values	19
4.4	Building blocks for custom logic	21
5	External app support	25
5.1	Autocompletion for GenericForeignKey	25
5.2	Autocompletion for django-gm2m's GM2MField	27
5.3	Autocompletion for django-generic-m2m's RelatedObjectsDescriptor	28
5.4	Autocompletion for django-tagging's TagField	29
5.5	Autocompletion for django-taggit's TaggableManager	30
6	API	33
6.1	dal: django-autocomplete-light3 API	33
6.2	FutureModelForm	35
6.3	dal_select2: Select2 support for DAL	36
6.4	dal_contenttypes: GenericForeignKey support	39
6.5	dal_select2_queryset_sequence: Select2 for QuerySetSequence choices	39
6.6	dal_queryset_sequence: QuerySetSequence choices	40
6.7	dal_gm2m_queryset_sequence	42
6.8	dal_genericm2m_queryset_sequence	42
6.9	dal_gm2m: django-gm2m support	43
6.10	dal_genericm2m: django-genericm2m support	43
6.11	dal_select2_taggit: django-taggit support	43
6.12	dal_select2_tagging: django-tagging support	44
7	Indices and tables	45
	Python Module Index	47
	Index	49

pypi package 3.5.1

build failing

CHAPTER 1

Features

- Python 2.7, 3.4, Django 2.0+ support (Django 1.11 (LTS), is supported until django-autocomplete-light-3.2.10),
- Django (multiple) choice support,
- Django (multiple) model choice support,
- Django generic foreign key support (through django-querysetsequence),
- Django generic many to many relation support (through django-generic-m2m and django-gm2m)
- Multiple widget support: select2.js, easy to add more.
- Creating choices that don't exist in the autocomplete,
- Offering choices that depend on other fields in the form, in an elegant and innovant way,
- Dynamic widget creation (ie. inlines), supports YOUR custom scripts too,
- Provides a test API for your awesome autocompletes, to support YOUR custom use cases too,
- A documented, automatically tested example for each use case in test_project.

CHAPTER 2

Upgrading

See [CHANGELOG..](#)

For v2 users and experts, a [blog post](#) was published with plenty of details.

CHAPTER 3

Resources

- ****Live demo**** graciously hosted by Repl.it
- ****Documentation**** graciously hosted by RTFD
- Video demo graciously hosted by Youtube,
- Mailing list graciously hosted by Google
- For **Security** issues, please contact yourlabs-security@googlegroups.com
- Git graciously hosted by GitHub,
- Package graciously hosted by PyPi,
- Continuous integration graciously hosted by Travis-ci
- ****Online paid support**** provided via HackHands,

4.1 Install django-autocomplete-light v3

4.1.1 Install in your project

Install version 3 with `pip install`:

```
pip install django-autocomplete-light
```

Or, install the dev version with git:

```
pip install -e git+https://github.com/yourlabs/django-autocomplete-light.git  
↪#egg=django-autocomplete-light
```

Note: If you are trying to install from git, please make sure you are not using **zip/archive** url of the repo `django-autocomplete-light` since it will not contain required submodules automatically. Otherwise these submodules will then need to be updated separately using `git submodule update --init`.

4.1.2 Configuration

Then, to let Django find the static files we need by adding to `INSTALLED_APPS`, **before** `django.contrib.admin` and `grappelli` if present:

```
'dal',  
'dal_select2',  
# 'grappelli',  
'django.contrib.admin',
```

This is to override the `jquery.init.js` script provided by the admin, which sets up jQuery with `noConflict`, making jQuery available in `django.jQuery` only and not `$`.

To enable more DAL functionalities we will have to add other DAL apps to `INSTALLED_APPS`, such as `'dal_queryset_sequence'` ...

Django versions earlier than 2.0

You will need to add `dal_legacy_static` to your `INSTALLED_APPS` settings. This adds in `select2` static files that are included with Django 2.x but missing in earlier versions.

JQuery 3.x

JQuery 3.x comes with a “slim” version. This version is not compatible with DAL since the slim version does not contain Ajax functionality.

4.1.3 Install the demo project

Install the demo project in a temporary virtualenv for testing purpose:

```
cd /tmp
virtualenv -p python3 dal_env
source dal_env/bin/activate
pip install django
pip install -e git+https://github.com/yourlabs/django-autocomplete-light.git
↪ #egg=django-autocomplete-light
cd dal_env/src/django-autocomplete-light/test_project/
pip install -r requirements.txt
./manage.py migrate
./manage.py createsuperuser
./manage.py runserver
# go to http://localhost:8000/admin/ and login
```

4.2 django-autocomplete-light tutorial

4.2.1 Overview

Autocompletes are based on 3 moving parts:

- widget compatible with the model field, does the initial rendering,
- javascript widget initialization code, to trigger the autocomplete,
- and a view used by the widget script to get results from.

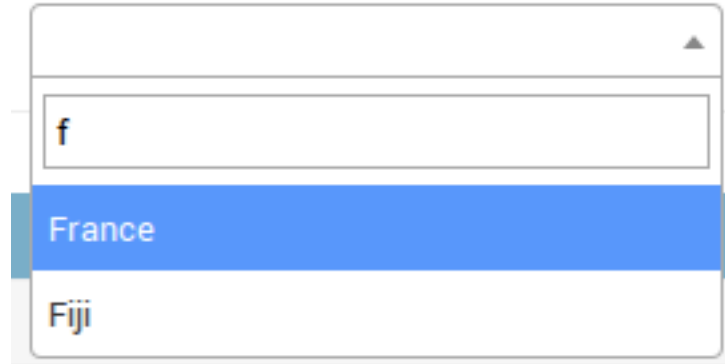
4.2.2 Create an autocomplete view

- Example source code: `test_project/select2_foreign_key`
- Live demo: `/select2_foreign_key/test-autocomplete/?q=test`

The only purpose of the autocomplete view is to serve relevant suggestions for the widget to propose to the user. DAL leverages Django’s `class based views` and `Mixins` to for code reuse.

Note: Do not miss the [Classy Class-Based Views](#) website which helps a lot to work with class-based views in general.

In this tutorial, we'll first learn to make autocompletes backed by a [QuerySet](#). Suppose we have a [Country Model](#) which we want to provide a [Select2](#) autocomplete widget for in a form. If a users types an "f" it would propose "Fiji", "Finland" and "France", to authenticated users only:



The base view for this is [Select2QuerySetView](#).

```
from dal import autocomplete

from your_countries_app.models import Country

class CountryAutocomplete(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        # Don't forget to filter out results depending on the visitor !
        if not self.request.user.is_authenticated():
            return Country.objects.none()

        qs = Country.objects.all()

        if self.q:
            qs = qs.filter(name__istartswith=self.q)

        return qs
```

Note: For more complex filtering, refer to official documentation for the [QuerySet API](#).

4.2.3 Register the autocomplete view

Create a `named` url for the view, ie:

```
from your_countries_app.views import CountryAutocomplete

urlpatterns = [
    url(
        r'^country-autocomplete/$',
        CountryAutocomplete.as_view(),
        name='country-autocomplete',
```

(continues on next page)

(continued from previous page)

```
),
]
```

Ensure that the url can be reversed, ie:

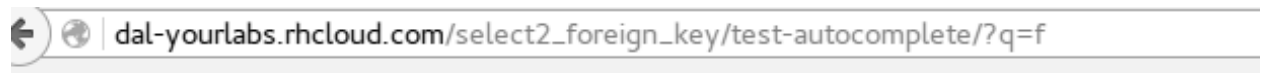
```
./manage.py shell
In [1]: from django.urls import reverse
In [2]: #older django versions: from django.core.urlresolvers import reverse

In [3]: reverse('country-autocomplete')
Out[2]: u'/country-autocomplete/'
```

Danger: As you might have noticed, we have just exposed data through a public URL. Please don't forget to do proper permission checks in `get_queryset`.

4.2.4 Use the view in a Form widget

You should be able to open the view at this point:



```
"pagination": {"more": false}, "results": [{"text": "France", "id": 50}, {"text": "Fiji", "id": 51}]}
```

We can now use the autocomplete view our Person form, for its `birth_country` field that's a `ForeignKey`. So, we're going to *override the default ModelForm fields*, to use a widget to select a Model with `Select2`, in our case by passing the name of the url we have just registered to `ModelSelect2`.

One way to do it is by overriding the form field, ie:

```
from dal import autocomplete

from django import forms

class PersonForm(forms.ModelForm):
    birth_country = forms.ModelChoiceField(
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2(url='country-autocomplete')
    )

    class Meta:
        model = Person
        fields = ('__all__')
```

Another way to do this is directly in the `Form.Meta.widgets` dict, if overriding the field is not needed:

```
from dal import autocomplete

from django import forms

class PersonForm(forms.ModelForm):
```

(continues on next page)

(continued from previous page)

```
class Meta:
    model = Person
    fields = ('__all__')
    widgets = {
        'birth_country': autocomplete.ModelSelect2(url='country-autocomplete')
    }
```

If we need the country autocomplete view for a widget used for a ManyToMany relation instead of a ForeignKey, with a model like that:

```
class Person(models.Model):
    visited_countries = models.ManyToManyField('your_countries_app.country')
```

Then we would use the *ModelSelect2Multiple* widget, ie.:

```
widgets = {
    'visited_countries': autocomplete.ModelSelect2Multiple(url='country-autocomplete')
}
```

4.2.5 Passing options to select2

Select2 supports a bunch of options. These options may be set in data-* attributes. For example:

```
# Instantiate a widget with a bunch of options for select2:
autocomplete.ModelSelect2(
    url='select2_fk',
    attrs={
        # Set some placeholder
        'data-placeholder': 'Autocomplete ...',
        # Only trigger autocompletion after 3 characters have been typed
        'data-minimum-input-length': 3,
    },
)
```

Note: Setting a placeholder will result in generation of an empty option tag, which select2 requires.

4.2.6 Using autocompletes in the admin

We can make ModelAdmin to use our form, ie:

```
from django.contrib import admin

from your_person_app.models import Person
from your_person_app.forms import PersonForm

class PersonAdmin(admin.ModelAdmin):
    form = PersonForm
admin.site.register(Person, PersonAdmin)
```

Note that this also works with inlines, ie:

```
class PersonInline(admin.TabularInline):
    model = Person
    form = PersonForm
```

4.2.7 Using autocompletes outside the admin

- Example source code: `test_project/select2_outside_admin`,
- Live demo: `/select2_outside_admin/`.

Ensure that jquery is loaded before `{{ form.media }}`:

```
{% extends 'base.html' %}
{# Don't forget that one ! #}
{% load static %}

{% block content %}
<div>
    <form action="" method="post">
        {% csrf_token %}
        {{ form.as_p }}

        <table style="display: none">
            {{ view.formset.empty_form }}
        </table>

        <table>
            {{ view.formset }}
        </table>

        <span id="add-form" class="button">Add form</span>

        <input type="submit" />
    </form>
</div>
{% endblock %}

{% block footer %}
<script type="text/javascript" src="{% static 'admin/js/vendor/jquery/jquery.js' %}">
</script>

{{ form.media }}

<script>
(function($) {
    $('#add-form').click(function() {
        var index = $('#id_inline_test_models-TOTAL_FORMS').val()
        var newTable = $('#id_inline_test_models-__prefix__-DELETE').parents('table').
        clone()
        newTable.find(':input').each(function() {
            for (attr of ['name', 'id'])
                $(this).attr(
                    attr,
                    $(this).attr(attr).replace('__prefix__', index)
                )
        })
    })
})
```

(continues on next page)

(continued from previous page)

```

newTable.insertBefore($(this))
$('#id_inline_test_models-TOTAL_FORMS').val(
    parseInt($('#id_inline_test_models-TOTAL_FORMS').val()) + 1
)
newTable.slideDown()
})
})($
</script>
{% endblock %}

```

4.2.8 Displaying results using custom HTML

You can display custom HTML code for results by setting the `data-html` attribute on your widget and overriding the view `get_result_label()` method to return HTML code.

```

from django.utils.html import format_html

class CountryAutocomplete(autocomplete.Select2QuerySetView):
    def get_result_label(self, item):
        return format_html(' {}'.format(item.name, item.name))

class PersonForm(forms.ModelForm):
    class Meta:
        widgets = {
            'birth_country': autocomplete.ModelSelect2(
                url='country-autocomplete',
                attrs={'data-html': True}
            )
        }

```

Note: Take care to escape anything you put in HTML code to avoid XSS attacks when displaying data that may have been input by a user! `format_html` helps.

4.2.9 Displaying selected result differently than in list

You can display selected result in different way than results in list by overriding the view `get_selected_result_label()` method.

```

class CountryAutocomplete(autocomplete.Select2QuerySetView):
    def get_result_label(self, item):
        return item.full_name

    def get_selected_result_label(self, item):
        return item.short_name

```

Setting the `data-html` attribute affects both selected result and results in list. If you want to enable HTML separately set `data-selected-html` or `data-result-html` attribute respectively.

4.2.10 Overriding javascript code

We need javascript initialization for the widget both when:

- the page is loaded,
- a widget is dynamically added, ie. with formsets.

This is handled by `autocomplete.init.js`, which is going to trigger an event called `autocompleteLightInitialize` on any HTML element with attribute `data-autocomplete-light-function` both on page load and DOM node insertion. It also keeps track of initialized elements to prevent double-initialization.

Take `dal_select2` for example, it is initialized by `dal_select2/static/autocomplete_light/select2.js` as such:

```
$(document).on('autocompleteLightInitialize', '[data-autocomplete-light-
↪function=select2]', function() {
    // do select2 configuration on $(this)
})
```

This example defines a callback that does `// do select2 configuration on $(this)` when the `autocompleteLightInitialize` event is triggered on any element with an attribute `data-autocomplete-light-function` of value `select2`. `Select2 Widgets` have an `autocomplete_function` of value `select2`, and that's rendered as the value of the `data-autocomplete-light-function` attribute.

So, you can replace the default callback by doing two things:

- change the Widget's `autocomplete_function` attribute,
- add a callback for the `autocompleteLightInitialize` event for that function,

Example widget:

```
class YourWidget (ModelSelect2):
    autocomplete_function = 'your-autocomplete-function'
```

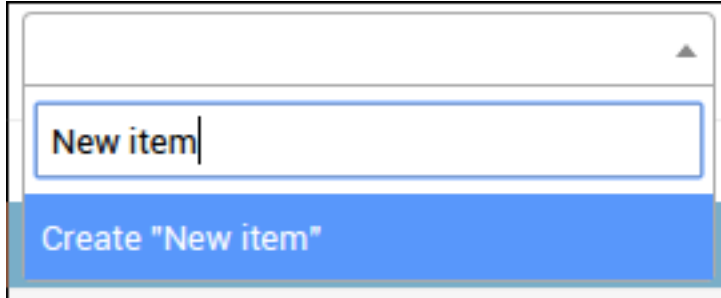
Example script:

```
$(document).on(
    'autocompleteLightInitialize',
    '[data-autocomplete-light-function=your-autocomplete-function]',
    function() {
        // do your own script setup here
    })
```

4.2.11 Creation of new choices in the autocomplete form

- Example source code: `test_project/select2_one_to_one`,
- Live demo: `/admin/select2_one_to_one/tmodel/add/`,

The view may provide an extra option when it can't find any result matching the user input. That option would have the label `Create "query"`, where `query` is the content of the input and corresponds to what the user typed in. As such:



This allows the user to create objects on the fly from within the AJAX widget. When the user selects that option, the autocomplete script will make a POST request to the view. It should create the object and return the pk, so the item will then be added just as if it already had a PK:



To enable this, first the view must know how to create an object given only `self.q`, which is the variable containing the user input in the view. Set the `create_field` view option to enable creation of new objects from within the autocomplete user interface, ie:

```
urlpatterns = [
    url(
        r'^country-autocomplete/$',
        CountryAutocomplete.as_view(create_field='name'),
        name='country-autocomplete',
    ),
]
```

This way, the option ‘Create “Tibet”’ will be available if a user inputs “Tibet” for example. When the user clicks it, it will make the post request to the view which will do `Country.objects.create(name='Tibet')`. It will be included in the server response so that the script can add it to the widget.

Note that creating objects is allowed to logged-in users with `add` permission on the resource. If you want to grant `add` permission to a user, you have to explicitly set it with something like:

```
permission = Permission.objects.get(name='Can add your-model-name')
user.user_permissions.add(permission)
```

4.2.12 Filtering results based on the value of other fields in the form

- Example source code: [test_project/linked_data](#).
- Live demo: [Admin / Linked Data / Add](#).

In the live demo, create a `TestModel` with `owner=None`, and another with `owner=test` (test being the user you log in with). Then, in a new form, you’ll see both options if you leave the owner select empty:

But if you select `test` as an owner, and open the autocomplete again, you'll only see the option with `owner=test`:

Let's say we want to add a “Continent” choice field in the form, and filter the countries based on the value on this field. We then need the widget to pass the value of the continent field to the view when it fetches data. We can use the `forward` widget argument to do this:

```
class PersonForm(forms.ModelForm):
    continent = forms.ChoiceField(choices=CONTINENT_CHOICES)

    class Meta:
        model = Person
        fields = ('__all__')
        widgets = {
            'birth_country': autocomplete.ModelSelect2(url='country-autocomplete',
                                                    forward=['continent'])
        }
```

DAL's `Select2` configuration script will get the value for the form field named `'continent'` and add it to the autocomplete HTTP query. This will pass the value for the “continent” form field in the AJAX request, and we can then filter as such in the view:

```
class CountryAutocomplete(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        if not self.request.user.is_authenticated():
```

(continues on next page)

(continued from previous page)

```

    return Country.objects.none()

qs = Country.objects.all()

continent = self.forwarded.get('continent', None)

if continent:
    qs = qs.filter(continent=continent)

if self.q:
    qs = qs.filter(name__startswith=self.q)

return qs

```

4.3 Types of forwarded values

There are three possible types of value which you can get from `self.forwarded` field: boolean, string or list of strings. DAL forward JS applies the following rules when figuring out which type to use when you forward particular field:

- if there is only one field in the form or subform with given name

and this field is a checkbox without `value` HTML-attribute, then a boolean value indicating if this checkbox is checked is forwarded;

- if there is only one field in the form or subform with given name

and it has `multiple` HTML-attribute, then this field is forwarded as a list of strings, containing values from this field. - if there are one or more fields in the form with given name and all of them are checkboxes with HTML-attribute `value` set, then the list of strings containing checked checkboxes is forwarded. - Otherwise field value forwarded as a string.

- Example source code: [test_project/rename_forward](#).
- Live demo: [Admin / Rename Forward/ Add](#).

Let's assume that you have the following form using linked autocomplete fields:

```

class ShippingForm(forms.Form):
    src_continent = forms.ModelChoiceField(
        queryset=Continent.objects.all(),
        widget=autocomplete.ModelSelect2(url='continent-autocomplete'))
    src_country = forms.ModelChoiceField(
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2(
            url='country-autocomplete',
            forward=('src_continent',)))

```

And the following autocomplete view for country:

```

class CountryAutocomplete(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        if not self.request.is_authenticated():
            return Country.objects.none()

qs = Country.objects.all()

```

(continues on next page)

(continued from previous page)

```
continent = self.forwarded.get('continent', None)

if continent:
    qs = qs.filter(continent=continent)

if self.q:
    qs = qs.filter(name__istartswith=self.q)

return qs
```

You cannot use this autocomplete view together with your form because the name forwarded from the form differs from the name that autocomplete view expects.

You can rename forwarded fields using class-based forward declaration to pass `src_continent` value as `continent`:

```
from dal import forward

class ShippingForm(forms.Form):
    src_continent = forms.ModelChoiceField(
        queryset=Continent.objects.all(),
        widget=autocomplete.ModelSelect2(url='continent-autocomplete'))
    src_country = forms.ModelChoiceField(
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2(
            url='country-autocomplete',
            forward=(forward.Field('src_continent', 'continent'),)))
```

Of course, you can mix up string-based and class-based forwarding declarations:

```
some_field = forms.ModelChoiceField(
    queryset=SomeModel.objects.all(),
    widget=autocomplete.ModelSelect2(
        url='some-autocomplete',
        forward=(
            'f1', # String based declaration
            forward.Field('f2'), # Works the same way as above declaration
            forward.Field('f3', 'field3'), # With rename
            forward.Const(42, 'f4') # Constant forwarding (see below)
        )
    )
```

The other thing you can do with class-based forwarding declaration is to forward an arbitrary constant without adding extra hidden fields to your form.

```
from dal import forward

class EuropeanShippingForm(forms.Form):
    src_country = forms.ModelChoiceField(
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2(
            url='country-autocomplete',
            forward=(forward.Const('europe', 'continent'),)))
```

For `src_country` field “europe” will always be forwarded as `continent` value.

Quite often (especially in multiselect) you may want to exclude value which is already selected from autocomplete dropdown. Usually it can be done by forwarding a field by name. The forward argument expects a tuple, so don't forget the trailing comma if the tuple only has one element.


```

from dal import forward

class SomeForm(forms.Form):
    countries = forms.ModelMultipleChoiceField(
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2Multiple(
            url='country-autocomplete',
            forward=("countries", )

```

For this special case DAL provides a shortcut named `Self()`.

```

from dal import forward

class SomeForm(forms.Form):
    countries = forms.ModelMultipleChoiceField(
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2Multiple(
            url='country-autocomplete',
            forward=(forward.Self(), )

```

In this case the value from `countries` will be available from autocomplete view as `self.forwarded['self']`. Of course, you can customize destination name by passing `dst` parameter to `Self` constructor.

DAL tries hard to reasonably forward any standard HTML form field. For some non-standard fields DAL logic could be not good enough. For these cases DAL provides a way to customize forwarding logic using JS callbacks. You can register JS forward handler on your page:

Then you should add forward declaration to your field as follows:

```

from dal import forward

class ShippingForm(forms.Form):
    country = forms.ModelChoiceField(
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2(
            url='country-autocomplete',
            forward=(forward.JavaScript('my_awesome_handler', 'magic_number'),))

```

In this case the value returned from your registered handler will be forwarded to autocomplete view as `magic_number`.

4.4 Building blocks for custom logic

Javascript logic for forwarding field values is a bit sophisticated. In order to forward field value DAL searches for the field considering form prefixes and then decides how to forward it to the server (should it be list, string or boolean value). When you implement your own logic for forwarding you may want to reuse this logic from DAL.

For this purpose DAL provides two JS functions:

- `getFieldRelativeTo(element, name)` - get field by name relative to this

autocomplete field just like DAL does when forwarding a field.

- `getValueFromField(field)` - get value to forward from `field` just like

DAL does when forwarding a field.

For the purpose of understanding the logic: you can implement forwarding of some standard field by yourself as follows (you probably should never write this code yourself):

You can use the `$.getFormPrefix()` jQuery plugin used by DAL to clear the `birth_country` autocomplete widget from the above example when the `continent` field changes with such a snippet:

```
$(document).ready(function() {
    // Bind on continent field change
    $(':input[name$=continent]').on('change', function() {
        // Get the field prefix, ie. if this comes from a formset form
        var prefix = $(this).getFormPrefix();

        // Clear the autocomplete with the same prefix
        $(':input[name=' + prefix + 'birth_country]').val(null).trigger('change');
    });
});
```

To autoload the script with the form, you can use `Form.Media`.

4.4.1 Autocompleting based on a List of Strings

Sometimes it is useful to specify autocomplete choices based on a list of strings rather than a `QuerySet`. This can be achieved with the `Select2ListView` class:

```
class CountryAutocompleteFromList(autocomplete.Select2ListView):
    def get_list(self):
        return ['France', 'Fiji', 'Finland', 'Switzerland']
```

This class can then be registered as in the previous example. Suppose we register it under URL `'country-list-autocomplete'`. We can then create a `ListSelect2` widget with:

```
widget = autocomplete.ListSelect2(url='country-list-autocomplete')
```

With this in place, if a user types the letter `f` in the widget, choices `'France'`, `'Fiji'`, and `'Finland'` would be offered. Like the `Select2QuerySetView`, the `Select2ListView` is case insensitive.

Two fields are provided, `Select2ListChoiceField`, `Select2ListCreateChoiceField` that can be used to make it easier to avoid problems when using `Select2ListView`. For example:

```
def get_choice_list():
    return ['France', 'Fiji', 'Finland', 'Switzerland']

class CountryForm(forms.ModelForm):
    country = autocomplete.Select2ListChoiceField(
        choice_list=get_choice_list,
        widget=autocomplete.ListSelect2(url='country-list-autocomplete')
    )
```

Since the selections in `Select2ListView` map directly to a list, there is no built-in support for choices in a `ChoiceField` that do not have the same value for every text. `Select2ListCreateChoiceField` allows you to provide custom text from a `Select2List` widget and should be used if you define `Select2ListViewAutocomplete.create`.

It is better to use the same source for `Select2ListViewAutocomplete.get_list` in your view and the `Select2ListChoiceField` `choice_list` kwarg to avoid unexpected behavior.

An opt-group version is available in a similar fashion by inheriting `Select2GroupListView` :

```
class CountryAutocompleteFromList (autocomplete.Select2GroupListView):
    def get_list(self):
        return [
            ("Country", ['France', 'Fiji', 'Finland', 'Switzerland'])
        ]
```


5.1 Autocompletion for GenericForeignKey

5.1.1 Model example

Consider such a model:

```
from django.contrib.contenttypes.fields import GenericForeignKey
from django.db import models

class TestModel(models.Model):
    name = models.CharField(max_length=200)
    language = models.CharField(max_length=200)

    content_type = models.ForeignKey(
        'contenttypes.ContentType',
        null=True,
        blank=True,
        editable=False,
    )

    object_id = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )

    location = GenericForeignKey('content_type', 'object_id')

    def __str__(self):
        return self.name
```

5.1.2 Form example

To enable the use of automatic views we need to add `'dal_queryset_sequence'` to `INSTALLED_APPS`.

First, we can't use Django's `ModelForm` because it doesn't support non-editable fields, which `GenericForeignKey` is. Instead, we'll use `FutureModelForm`.

Then we need to add the `Select2GenericForeignKeyModelField` field, with `model_choice` as keyword: this is a list of tuple, with the models you want in the autocompletion and the validation, and the value of the attribute of the model you want to query in the widget searchbox. Optionally, you can forward an existing field in the form to filter an attribute of the model, by adding a list of tuple containing the field to forward and the value to filter. In this example, the text inserted in the language field will filter the country models by their `'spoken_language'` attribute.

Result:

```
from dal import autocomplete

class TestForm(autocomplete.FutureModelForm):

    location = autocomplete.Select2GenericForeignKeyModelField(
        # Model with values to filter, linked with the name field
        model_choice=[(Country, 'country_code', [('language', 'spoken_language')]),
                     (City, 'name')],
    )

    class Meta:
        model = TestModel
```

If you want to use your own widgets and views, assuming the widget takes an url as argument and the view takes a queryset in its `"as_view()"` method, you can use `GenericForeignKeyModelField`:

```
from dal import autocomplete

class TestForm(autocomplete.FutureModelForm):

    location = autocomplete.GenericForeignKeyModelField(
        model_choice=[(Country,), (City,)], # Models
        widget=autocomplete.QuerySetSequenceSelect2,
        view=autocomplete.Select2QuerySetSequenceView,
    )

    class Meta:
        model = TestModel
```

In this example, we took `QuerySetSequenceSelect2` as the custom widget and `Select2QuerySetSequenceView`.

5.1.3 Register the view for the form

In `url.py`: .. code-block:: python

```
from .forms import TestForm

urlpatterns = [...] # your regular url patterns
urlpatterns.extend(TestForm.as_urls())
```

It will enable the search box to query and filter the results

5.2 Autocompletion for django-gm2m's GM2MField

5.2.1 Model example

Consider such a model, using `django-gm2m` to handle generic many-to-many relations:

```
from django.db import models

from gm2m import GM2MField

class TestModel(models.Model):
    name = models.CharField(max_length=200)

    locations = GM2MField()

    def __str__(self):
        return self.name
```

5.2.2 View example

The *Form example* works here too: we're relying on `Select2` and `QuerySetSequence` again.

5.2.3 Form example

As usual, we need a backend-aware widget that will make only selected choices to render initially, to avoid butchering the database. As we're using a `QuerySetSequence` and `Select2`, we'll try `QuerySetSequenceSelect2Multiple` widget.

Also, we need a field that's able to use a `QuerySetSequence` for choices to validate multiple models, and then update the `GM2MField` relations: `GM2MQuerySetSequenceField`.

Finally, we can't use Django's `ModelForm` because it doesn't support non-editable fields, which `GM2MField` is. Instead, we'll use `FutureModelForm`.

Example:

```
class TestForm(autocomplete.FutureModelForm):
    locations = autocomplete.GM2MQuerySetSequenceField(
        queryset=autocomplete.QuerySetSequence(
            Country.objects.all(),
            City.objects.all(),
        ),
        required=False,
        widget=autocomplete.QuerySetSequenceSelect2Multiple(
            'location-autocomplete'),
    )

    class Meta:
        model = TestModel
        fields = ('name',)
```

5.3 Autocompletion for django-generic-m2m's RelatedObjectsDescriptor

5.3.1 Model example

Consider such a model, using `django-generic-m2m` to handle generic many-to-many relations:

```
from django.db import models

from genericm2m.models import RelatedObjectsDescriptor

class TestModel(models.Model):
    name = models.CharField(max_length=200)

    locations = RelatedObjectsDescriptor()

    def __str__(self):
        return self.name
```

5.3.2 View example

The *Form example* works here too: we're relying on `Select2` and `QuerySetSequence` again.

5.3.3 Form example

As usual, we need a backend-aware widget that will make only selected choices to render initially, to avoid butchering the database. As we're using a `QuerySetSequence` and `Select2` for multiple selections, we'll try `QuerySetSequenceSelect2Multiple` widget.

Also, we need a field that's able to use a `QuerySetSequence` for choices to validate multiple models, and then update the `RelatedObjectsDescriptor` relations: `GenericM2MQuerySetSequenceField`.

Finally, we can't use Django's `ModelForm` because it doesn't support non-editable fields, which `RelatedObjectsDescriptor` is. Instead, we'll use `FutureModelForm`.

Example:

```
class TestForm(autocomplete.FutureModelForm):
    locations = autocomplete.GenericM2MQuerySetSequenceField(
        queryset=autocomplete.QuerySetSequence(
            Country.objects.all(),
            City.objects.all(),
        ),
        required=False,
        widget=autocomplete.QuerySetSequenceSelect2Multiple(
            'location-autocomplete'),
    )

    class Meta:
        model = TestModel
        fields = ('name',)
```


5.4 Autocompletion for django-tagging's TagField

5.4.1 Model example

Consider such a model, using `django-tagging` to handle tags for a model:

```
from django.db import models

from tagging.fields import TagField

class TestModel(models.Model):
    name = models.CharField(max_length=200)

    tags = TagField()

    def __str__(self):
        return self.name
```

5.4.2 View example

The *QuerySet* view works here too: we're relying on `Select2` and a `QuerySet` of `Tag` objects:

```
from dal import autocomplete

from tagging.models import Tag

class TagAutocomplete(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        # Don't forget to filter out results depending on the visitor !
        if not self.request.user.is_authenticated():
            return Tag.objects.none()

        qs = Tag.objects.all()

        if self.q:
            qs = qs.filter(name__startswith=self.q)

        return qs
```

Note: Don't forget to *Register the autocomplete view*.

5.4.3 Form example

As usual, we need a backend-aware widget that will make only selected choices to render initially, to avoid butchering the database.

As we're using a `QuerySet` of `Tag` and `Select2` in its "tag" appearance, we'll use `TaggitSelect2`. It is compatible with the default form field created by the model field: `TagField`.

Example:

```
class TestForm(autocomplete.FutureModelForm):
    class Meta:
        model = TestModel
        fields = ('name',)
        widgets = {
            'tags': autocomplete.TaggingSelect2(
                'your-taggit-autocomplete-url'
            )
        }
```

5.5 Autocompletion for django-taggit's TaggableManager

5.5.1 Model example

Consider such a model, using `django-taggit` to handle tags for a model:

```
from django.db import models

from taggit.managers import TaggableManager

class TestModel(models.Model):
    name = models.CharField(max_length=200)

    tags = TaggableManager()

    def __str__(self):
        return self.name
```

5.5.2 View example

The *QuerySet view* works here too: we're relying on `Select2` and a `QuerySet` of `Tag` objects:

```
from dal import autocomplete

from taggit.models import Tag

class TagAutocomplete(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        # Don't forget to filter out results depending on the visitor !
        if not self.request.user.is_authenticated():
            return Tag.objects.none()

        qs = Tag.objects.all()

        if self.q:
            qs = qs.filter(name__startswith=self.q)

        return qs
```

Don't forget to *Register the autocomplete view*.

Note: For more complex filtering, refer to official documentation for the `QuerySet` API.

5.5.3 Form example

As usual, we need a backend-aware widget that will make only selected choices to render initially, to avoid butchering the database.

As we're using a `QuerySet` of `Tag` and `Select2` in its "tag" appearance, we'll use `TaggitSelect2`. It is compatible with the default form field created by the model field: `TaggableManager` - which actually inherits `django.db.models.fields.Field` and `django.db.models.fields.related.RelatedField` and **not** from `django.db.models.Manager`.

Example:

```
class TestForm(autocomplete.FutureModelForm):
    class Meta:
        model = TestModel
        fields = ('name',)
        widgets = {
            'tags': autocomplete.TaggitSelect2(
                'your-taggit-autocomplete-url'
            )
        }
```


6.1 dal: django-autocomplete-light3 API

6.1.1 Views

Base views for autocomplete widgets.

```
class dal.views.BaseQuerySetView (**kwargs)
```

Base view to get results from a QuerySet.

create_field

Name of the field to use to create missing values. For example, if `create_field='title'`, and the user types in “foo”, then the autocomplete view will propose an option ‘Create “foo”’ if it can’t find any value matching “foo”. When the user does click ‘Create “foo”’, the autocomplete script should POST to this view to create the object and get back the newly created object id.

model_field_name

Name of the Model field to run filter against.

create_object (*text*)

Create an object given a text.

get_queryset ()

Filter the queryset with GET[‘q’].

get_result_label (*result*)

Return the label of a result.

get_result_value (*result*)

Return the value of a result.

get_selected_result_label (*result*)

Return the label of a selected result.

has_add_permission (*request*)

Return True if the user has the permission to add a model.

has_more (*context*)

For widgets that have infinite-scroll feature.

post (*request*)

Create an object given a text after checking permissions.

class `dal.views.ViewMixin`

Common methods for autocomplete views.

It is assumed this view will be used in conjunction with a Django `View` based class that will that will implement `OPTIONS`.

forwarded

Dict of field values that were forwarded from the form, may be used to filter autocomplete results based on the form state. See `linked_data` example for reference.

q

Query string as typed by the user in the autocomplete field.

dispatch (*request*, **args*, ***kwargs*)

Set `forwarded` and `q`.

6.1.2 Widgets

Autocomplete widgets bases.

class `dal.widgets.QuerySetSelectMixin` (*url=None*, *forward=None*, **args*, ***kwargs*)

QuerySet support for choices.

filter_choices_to_render (*selected_choices*)

Filter out un-selected choices if choices is a QuerySet.

class `dal.widgets.Select` (*url=None*, *forward=None*, **args*, ***kwargs*)

Replacement for Django's `Select` to render only selected choices.

class `dal.widgets.SelectMultiple` (*url=None*, *forward=None*, **args*, ***kwargs*)

Replacement `SelectMultiple` to render only selected choices.

class `dal.widgets.WidgetMixin` (*url=None*, *forward=None*, **args*, ***kwargs*)

Base mixin for autocomplete widgets.

url

Absolute URL to the autocomplete view for the widget. It can be set to a a URL name, in which case it will be reversed when the attribute is accessed.

forward

List of field names to forward to the autocomplete view, useful to filter results using values of other fields in the form.

Items of the list must be one of the following:

- string (e. g. "some_field"): forward a value from the field with named "some_field";
- `dal.forward.Field("some_field")`: the same as above;
- `dal.forward.Field("some_field", "dst_field")`: forward a value from the field with named "some_field" as "dst_field";
- `dal.forward.Const("some_value", "dst_field")`: forward a constant value "some_value" as "dst_field".

autocomplete_function

Identifier of the javascript callback that should be executed when such a widget is loaded in the DOM, either on page load or dynamically.

build_attrs (**args, **kwargs*)

Build HTML attributes for the widget.

filter_choices_to_render (*selected_choices*)

Replace self.choices with selected_choices.

optgroups (*name, value, attrs=None*)

Exclude unselected self.choices before calling the parent method.

Used by Django>=1.10.

render (*name, value, attrs=None, renderer=None, **kwargs*)

Call Django render together with *render_forward_conf*.

render_forward_conf (*id*)

Render forward configuration for the field.

render_options (**args*)

Django-compatibility method for option rendering.

Should only render selected options, by setting self.choices before calling the parent method.

Remove this code when dropping support for Django<1.10.

6.1.3 Fields

6.2 FutureModelForm

tl;dr: See FutureModelForm's docstring.

Many apps provide new related managers to extend your django models with. For example, django-tagulous provides a TagField which abstracts an M2M relation with the Tag model, django-gm2m provides a GM2MField which abstracts an relation, django-taggit provides a TaggableManager which abstracts a relation too, django-generic-m2m provides RelatedObjectsDescriptor which abstracts a relation again.

While that works pretty well, it gets a bit complicated when it comes to encapsulating the business logic for saving such data in a form object. This is three-part problem:

- getting initial data,
- saving instance attributes,
- saving relations like reverse relations or many to many.

Django's ModelForm calls the model field's `value_from_object()` method to get the initial data. FutureModelForm tries the `value_from_object()` method from the form field instead, if defined. Unlike the model field, the form field doesn't know its name, so FutureModelForm passes it when calling the form field's `value_from_object()` method.

Django's ModelForm calls the form field's `save_form_data()` in two occasions:

- in `_post_clean()` for model fields in `Meta.fields`,
- in `_save_m2m()` for model fields in `Meta.virtual_fields` and `Meta.many_to_many`, which then operate on an instance which as a PK.

If we just added `save_form_data()` to form fields like `for_value_from_object()` then it would be called twice, once in `_post_clean()` and once in `_save_m2m()`. Instead, `FutureModelForm` would call the following methods from the form field, if defined:

- `save_object_data()` in `_post_clean()`, to set object attributes for a given value,
- `save_relation_data()` in `_save_m2m()`, to save relations for a given value.

For example:

- a generic foreign key only sets instance attributes, its form field would do that in `save_object_data()`,
- a tag field saves relations, its form field would do that in `save_relation_data()`.

class `dal.forms.FutureModelForm(*args, **kwargs)`
 ModelForm which adds extra API to form fields.

Form fields may define new methods for `FutureModelForm`:

- `FormField.value_from_object(instance, name)` should return the initial value to use in the form, overrides `ModelField.value_from_object()` which is what `ModelForm` uses by default,
- `FormField.save_object_data(instance, name, value)` should set instance attributes. Called by `save()` **before** writing the database, when `instance.pk` may not be set, it overrides `ModelField.save_form_data()` which is normally used in this occasion for non-m2m and non-virtual model fields.
- `FormField.save_relation_data(instance, name, value)` should save relations required for value on the instance. Called by `save()` **after** writing the database, when `instance.pk` is necessarily set, it overrides `ModelField.save_form_data()` which is normally used in this occasion for m2m and virtual model fields.

For complete rationale, see this module's docstring.

classmethod `as_urls()`

Create a list of url patterns, to be called in `url.py`.

`urlpatterns.append(*ModelForm.as_url())` Iterate over the fields to call the `as_url()` method from the `GenericForeignKeyField`

save (*commit=True*)

Backport from Django 1.9+ for 1.8.

6.3 dal_select2: Select2 support for DAL

This is a front-end module: it provides views and widgets.

6.3.1 Views

Select2 view implementation.

class `dal_select2.views.Select2GroupListView(**kwargs)`
 View mixin for grouped options.

get (*request, *args, **kwargs*)

Return option list with children(s) json response.

get_item_as_group (*entry*)

Return the item with its group.


```

class dal_select2.views.Select2ListView (**kwargs)
    Autocomplete from a list of items rather than a QuerySet.

    autocomplete_results (results)
        Return list of strings that match the autocomplete query.

    get (request, *args, **kwargs)
        Return option list json response.

    get_list ()
        Return the list strings from which to autocomplete.

    post (request)
        Add an option to the autocomplete list.

        If 'text' is not defined in POST or self.create(text) fails, raises bad request. Raises ImproperlyConfigured
        if self.create if not defined.

    results (results)
        Return the result dictionary.

class dal_select2.views.Select2QuerySetView (**kwargs)
    List options for a Select2 widget.

class dal_select2.views.Select2ViewMixin
    View mixin to render a JSON response for Select2.

    get_create_option (context, q)
        Form the correct create_option to append to results.

    get_results (context)
        Return data for the 'results' key of the response.

    render_to_response (context)
        Return a JSON response in Select2 format.

```

6.3.2 Widgets

Select2 widget implementation module.

```

class dal_select2.widgets.ListSelect2 (url=None, forward=None, *args, **kwargs)
    Select widget for regular choices and Select2.

class dal_select2.widgets.ModelSelect2 (url=None, forward=None, *args, **kwargs)
    Select widget for QuerySet choices and Select2.

class dal_select2.widgets.ModelSelect2Multiple (url=None, forward=None, *args,
                                                **kwargs)
    SelectMultiple widget for QuerySet choices and Select2.

class dal_select2.widgets.Select2 (url=None, forward=None, *args, **kwargs)
    Select2 widget for regular choices.

class dal_select2.widgets.Select2Multiple (url=None, forward=None, *args, **kwargs)
    Select2Multiple widget for regular choices.

class dal_select2.widgets.Select2WidgetMixin
    Mixin for Select2 widgets.

    build_attrs (*args, **kwargs)
        Set data-autocomplete-light-language.

```

media

Return JS/CSS resources for the widget.

class `dal_select2.widgets.TagSelect2` (*url=None, forward=None, *args, **kwargs*)

Select2 in tag mode.

build_attrs (**args, **kwargs*)

Automatically set data-tags=1.

format_value (*value*)

Return the list of HTML option values for a form field value.

optgroup (*name, value, attrs=None*)

Return a list of one optgroup and selected values.

option_value (*value*)

Return the HTML option value attribute for a value.

options (*name, value, attrs=None*)

Return only select options.

value_from_datadict (*data, files, name*)

Return a comma-separated list of options.

This is needed because Select2 uses a multiple select even in tag mode, and the model field expects a comma-separated list of tags.

`dal_select2.widgets.get_i18n_name`

Ensure lang_code is supported by Select2.

6.3.3 Fields

Select2 field implementation module.

class `dal_select2.fields.Select2ListChoiceField` (*choice_list=None, required=True, widget=None, label=None, initial=None, help_text="", *args, **kwargs*)

Allows a list of values to be used with a ChoiceField.

Avoids unusual things that can happen if Select2ListView is used for a form where the text and value for choices are not the same.

class `dal_select2.fields.Select2ListCreateChoiceField` (*choice_list=None, required=True, widget=None, label=None, initial=None, help_text="", *args, **kwargs*)

Skips validation of choices so any value can be used.

validate (*value*)

Do not validate choices but check for empty.

6.3.4 Test tools

Helpers for DAL user story based tests.

class `dal_select2.test.Select2Story`

Define Select2 CSS selectors.

clean_label (*label*)

Remove the “remove” character used in select2.

wait_script ()

Wait for scripts to be loaded and ready to work.

6.4 dal_contenttypes: GenericForeignKey support

6.4.1 Fields

Model choice fields that take a ContentType too: for generic relations.

class `dal_contenttypes.fields.ContentTypeModelFieldMixin`

Common methods for form fields for GenericForeignKey.

ModelChoiceFieldMixin expects options to look like:

```
<option value="4">Model #4</option>
```

With a ContentType of id 3 for that model, it becomes:

```
<option value="3-4">Model #4</option>
```

prepare_value (*value*)

Return a ctypeid-objpk string for value.

class `dal_contenttypes.fields.ContentTypeModelMultipleFieldMixin`

Same as ContentTypeModelFieldMixin, but supports value list.

prepare_value (*value*)

Run the parent’s method for each value.

class `dal_contenttypes.fields.GenericModelMixin`

GenericForeignKey support for form fields, with FutureModelForm.

GenericForeignKey enforce `editable=False`, this class implements `save_object_data()` and `value_from_object()` to allow FutureModelForm to compensate.

save_object_data (*instance, name, value*)

Set the attribute, for FutureModelForm.

value_from_object (*instance, name*)

Get the attribute, for FutureModelForm.

6.5 dal_select2_queryset_sequence: Select2 for QuerySetSequence choices

6.5.1 Views

View for a Select2 widget and QuerySetSequence-based business logic.

class `dal_select2_queryset_sequence.views.Select2QuerySetSequenceAutoView` (**kwargs)

Select2QuerySetSequenceAutoView class.

Filter the queryset based on the models and filter attributes of the GenericForeignKeyModelField

self.model_choice is generated from the Select2GenericForeignKeyModelField, see it's docstring

get_queryset ()
Return queryset.

class dal_select2_queryset_sequence.views.**Select2QuerySetSequenceView** (**kwargs)
Combines support QuerySetSequence and Select2 in a single view.

Example usage:

```
url (
    '^your-generic-autocomplete/$',
    autocomplete.Select2QuerySetSequenceView.as_view (
        queryset=autocomplete.QuerySetSequence (
            Group.objects.all (),
            TestModel.objects.all (),
        )
    ),
    name='your-generic-autocomplete',
)
```

It is compatible with the *widgets* and the fields of dal_contenttypes, suits generic relation autocompletes.

get_results (context)
Return a list of results usable by Select2.

It will render as a list of one <optgroup> per different content type containing a list of one <option> per model.

6.5.2 Widgets

Widgets for Select2 and QuerySetSequence.

They combine *Select2WidgetMixin* and *QuerySetSequenceSelectMixin* with Django's *Select* and *SelectMultiple* widgets, and are meant to be used with generic model form fields such as those in dal_contenttypes.

class dal_select2_queryset_sequence.widgets.**QuerySetSequenceSelect2** (url=None, for-ward=None, *args, **kwargs)

Single model select for a generic select2 autocomplete.

class dal_select2_queryset_sequence.widgets.**QuerySetSequenceSelect2Multiple** (url=None, for-ward=None, *args, **kwargs)

Multiple model select for a generic select2 autocomplete.

6.6 dal_queryset_sequence: QuerySetSequence choices

6.6.1 Views

View that supports QuerySetSequence.

```
class dal_queryset_sequence.views.BaseQuerySetSequenceView (**kwargs)
```

Base view that uses a QuerySetSequence.

Compatible with form fields which use a ContentType id as well as a model pk to identify a value.

```
get_model_name (model)
```

Return the name of the model, fetch parent if model is a proxy.

```
get_paginate_by (queryset)
```

Don't paginate if mixup.

```
get_queryset ()
```

Mix results from all querysets in QuerySetSequence if self.mixup.

```
get_result_value (result)
```

Return ctypeid-objectid for result.

```
has_more (context)
```

Return False if mixup.

```
mixup_querysets (qs)
```

Return a queryset with different model types.

6.6.2 Fields

Autocomplete fields for QuerySetSequence choices.

```
class dal_queryset_sequence.fields.GenericForeignKeyModelField (*args,
                                                                **kwargs)
```

Field that generate automatically the view for compatible widgets.

```
as_url (form)
```

Return url.

```
class dal_queryset_sequence.fields.QuerySetSequenceFieldMixin
```

Base methods for QuerySetSequence fields.

```
get_content_type_id_object_id (value)
```

Return a tuple of ctype id, object id for value.

```
get_queryset_for_content_type (content_type_id)
```

Return the QuerySet from the QuerySetSequence for a ctype.

```
raise_invalid_choice (params=None)
```

Raise a ValidationError for invalid_choice.

The validation error left unprecise about the exact error for security reasons, to prevent an attacker doing information gathering to reverse valid content type and object ids.

```
class dal_queryset_sequence.fields.QuerySetSequenceModelField (queryset, *,
                                                                empty_label='—',
                                                                required=True,
                                                                widget=None,
                                                                label=None,
                                                                initial=None,
                                                                help_text='',
                                                                to_field_name=None,
                                                                limit_choices_to=None,
                                                                **kwargs)
```

Replacement for ModelChoiceField supporting QuerySetSequence choices.

to_python (*value*)

Given a string like '3-5', return the model of ctype #3 and pk 5.

Note that in the case of ModelChoiceField, to_python is also in charge of security, it's important to get the results from self.queryset.

class dal_queryset_sequence.fields.**QuerySetSequenceModelMultipleField** (*queryset,*
***kwargs*)
 ModelMultipleChoiceField with support for QuerySetSequence choices.

6.6.3 Widgets

Widget mixin that only renders selected options with QuerySetSequence.

For details about why this is required, see *dal.widgets*.

class dal_queryset_sequence.widgets.**QuerySetSequenceSelect** (*url=None,* *for-*
*ward=None, *args,*
***kwargs*)
 Select widget for QuerySetSequence choices.

class dal_queryset_sequence.widgets.**QuerySetSequenceSelectMixin** (*url=None, for-*
ward=None,
**args,*
***kwargs*)
 Support QuerySetSequence in WidgetMixin.

filter_choices_to_render (*selected_choices*)

Overwrite self.choices to exclude unselected values.

label_from_instance (*obj*)

Convert an object into string. Override it to customize display.

class dal_queryset_sequence.widgets.**QuerySetSequenceSelectMultiple** (*url=None,*
for-
ward=None,
**args,*
***kwargs*)
 SelectMultiple widget for QuerySetSequence choices.

6.7 dal_gm2m_queryset_sequence

6.7.1 Fields

Form fields for using django-gm2m with QuerySetSequence.

class dal_gm2m_queryset_sequence.fields.**GM2MQuerySetSequenceField** (*queryset,*
***kwargs*)
 Form field for QuerySetSequence to django-generic-m2m relation.

6.8 dal_genericm2m_queryset_sequence

6.8.1 Fields

Autocomplete fields for django-queryset-sequence and django-generic-m2m.

class `dal_genericm2m_queryset_sequence.fields.GenericM2MQuerySetSequenceField` (*queryset*,
***kwargs*
Autocomplete field for GM2MField() for QuerySetSequence choices.

6.9 dal_gm2m: django-gm2m support

6.9.1 Fields

GM2MField support for autocomplete fields.

class `dal_gm2m.fields.GM2MFieldMixin`
GM2MField for FutureModelForm.

save_relation_data (*instance, name, value*)
Save the relation into the GM2MField.

value_from_object (*instance, name*)
Return the list of objects in the GM2MField relation.

6.10 dal_genericm2m: django-genericm2m support

6.10.1 Fields

django-generic-m2m field mixin for FutureModelForm.

class `dal_genericm2m.fields.GenericM2MFieldMixin`
Form field mixin able to get / set instance generic-m2m relations.

save_relation_data (*instance, name, value*)
Update the relation to be value.

value_from_object (*instance, name*)
Return the list of related objects.

6.11 dal_select2_taggit: django-taggit support

6.11.1 Fields

Widgets for Select2 and django-taggit.

class `dal_select2_taggit.widgets.TaggitSelect2` (*url=None*, *forward=None*, **args*,
***kwargs*)
Select2 tag widget for taggit's TagField.

build_attrs (**args, **kwargs*)
Add data-tags="".

option_value (*value*)
Return tag.name attribute of value.

render_options (**args*)
Render only selected tags.

Remove when Django < 1.10 support is dropped.

value_from_datadict (*data, files, name*)

Handle multi-word tag.

Insure there's a comma when there's only a single multi-word tag, or tag "Multi word" would end up as "Multi" and "word".

6.12 dal_select2_tagging: django-tagging support

6.12.1 Fields

Widgets for Select2 and django-taggit.

class dal_select2_tagging.widgets.**TaggingSelect2** (*url=None, forward=None, *args, **kwargs*)

Select2 tag widget for tagging's TagField.

render_options (**args*)

Render only selected tags.

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

d

`dal.forms`, 35
`dal.views`, 33
`dal.widgets`, 34
`dal_contenttypes.fields`, 39
`dal_genericm2m.fields`, 43
`dal_genericm2m_queryset_sequence.fields`,
42
`dal_gm2m.fields`, 43
`dal_gm2m_queryset_sequence.fields`, 42
`dal_queryset_sequence.fields`, 41
`dal_queryset_sequence.views`, 40
`dal_queryset_sequence.widgets`, 42
`dal_select2.fields`, 38
`dal_select2.test`, 38
`dal_select2.views`, 36
`dal_select2.widgets`, 37
`dal_select2_queryset_sequence.views`, 39
`dal_select2_queryset_sequence.widgets`,
40
`dal_select2_tagging.widgets`, 44
`dal_select2_taggit.widgets`, 43

A

`as_url()` (*dal_queryset_sequence.fields.GenericForeignKeyModelField* method), 41
`as_urls()` (*dal.forms.FutureModelForm* class method), 36
`autocomplete_function` (*dal.widgets.WidgetMixin* attribute), 34
`autocomplete_results()` (*dal_select2.views.Select2ListView* method), 37

B

`BaseQuerySetSequenceView` (class in *dal_queryset_sequence.views*), 40
`BaseQuerySetView` (class in *dal.views*), 33
`build_attrs()` (*dal.widgets.WidgetMixin* method), 35
`build_attrs()` (*dal_select2.widgets.Select2WidgetMixin* method), 37
`build_attrs()` (*dal_select2.widgets.TagSelect2* method), 38
`build_attrs()` (*dal_select2_taggit.widgets.TaggitSelect2* method), 43

C

`clean_label()` (*dal_select2.test.Select2Story* method), 38
`ContentTypeModelFieldMixin` (class in *dal_contenttypes.fields*), 39
`ContentTypeModelMultipleFieldMixin` (class in *dal_contenttypes.fields*), 39
`create_field` (*dal.views.BaseQuerySetView* attribute), 33
`create_object()` (*dal.views.BaseQuerySetView* method), 33

D

`dal.forms` (module), 35
`dal.views` (module), 33
`dal.widgets` (module), 34

`dal_contenttypes.fields` (module), 39
`dal_genericm2m.fields` (module), 43
`dal_genericm2m_queryset_sequence.fields` (module), 42
`dal_gm2m.fields` (module), 43
`dal_gm2m_queryset_sequence.fields` (module), 42
`dal_queryset_sequence.fields` (module), 41
`dal_queryset_sequence.views` (module), 40
`dal_queryset_sequence.widgets` (module), 42
`dal_select2.fields` (module), 38
`dal_select2.test` (module), 38
`dal_select2.views` (module), 36
`dal_select2.widgets` (module), 37
`dal_select2_queryset_sequence.views` (module), 39
`dal_select2_queryset_sequence.widgets` (module), 40
`dal_select2_tagging.widgets` (module), 44
`dal_select2_taggit.widgets` (module), 43
`dispatch()` (*dal.views.ViewMixin* method), 34

F

`filter_choices_to_render()` (*dal.widgets.QuerySetSelectMixin* method), 34
`filter_choices_to_render()` (*dal.widgets.WidgetMixin* method), 35
`filter_choices_to_render()` (*dal_queryset_sequence.widgets.QuerySetSequenceSelectMixin* method), 42
`format_value()` (*dal_select2.widgets.TagSelect2* method), 38
`forward` (*dal.widgets.WidgetMixin* attribute), 34
`forwarded` (*dal.views.ViewMixin* attribute), 34
`FutureModelForm` (class in *dal.forms*), 36

G

`GenericForeignKeyModelField` (class in *dal_queryset_sequence.fields*), 41

GenericM2MFieldMixin (class in *dal.views.BaseQuerySetView* method), 33
dal_genericm2m.fields), 43
 GenericM2MQuerySetSequenceField (class in *dal_genericm2m_queryset_sequence.fields*), 42
 GenericModelMixin (class in *dal_contenttypes.fields*), 39
 get () (*dal_select2.views.Select2GroupListView* method), 36
 get () (*dal_select2.views.Select2ListView* method), 37
 get_content_type_id_object_id () (*dal_queryset_sequence.fields.QuerySetSequenceFieldMixin* method), 41
 get_create_option () (*dal_select2.views.Select2ViewMixin* method), 37
 get_il8n_name (in module *dal_select2.widgets*), 38
 get_item_as_group () (*dal_select2.views.Select2GroupListView* method), 36
 get_list () (*dal_select2.views.Select2ListView* method), 37
 get_model_name () (*dal_queryset_sequence.views.BaseQuerySetSequenceView* method), 41
 get_paginate_by () (*dal_queryset_sequence.views.BaseQuerySetSequenceView* method), 41
 get_queryset () (*dal.views.BaseQuerySetView* method), 33
 get_queryset () (*dal_queryset_sequence.views.BaseQuerySetSequenceView* method), 41
 get_queryset () (*dal_select2_queryset_sequence.views.Select2QuerySetSequenceAutoView* method), 40
 get_queryset_for_content_type () (*dal_queryset_sequence.fields.QuerySetSequenceFieldMixin* method), 41
 get_result_label () (*dal.views.BaseQuerySetView* method), 33
 get_result_value () (*dal.views.BaseQuerySetView* method), 33
 get_result_value () (*dal_queryset_sequence.views.BaseQuerySetSequenceView* method), 41
 get_results () (*dal_select2.views.Select2ViewMixin* method), 37
 get_results () (*dal_select2_queryset_sequence.views.Select2QuerySetSequenceView* method), 40
 get_selected_result_label () (*dal.views.BaseQuerySetView* method), 33
 GM2MFieldMixin (class in *dal_gm2m.fields*), 43
 GM2MQuerySetSequenceField (class in *dal_gm2m_queryset_sequence.fields*), 42

H
 has_add_permission ()

L
 label_from_instance () (*dal_queryset_sequence.widgets.QuerySetSequenceSelectMixin* method), 42
 LabelSelect2 (class in *dal_select2.widgets*), 37

M
 media (*dal_select2.widgets.Select2WidgetMixin* attribute), 37
 mixup_querysets () (*dal_queryset_sequence.views.BaseQuerySetSequenceView* method), 41
 model_field_name (*dal.views.BaseQuerySetView* attribute), 33
 ModelSelect2 (class in *dal_select2.widgets*), 37
 ModelSelect2Multiple (class in *dal_select2.widgets*), 37

O
 optgroups () (*dal.widgets.WidgetMixin* method), 35
 optgroups () (*dal_select2.widgets.TagSelect2* method), 38
 option_value () (*dal_select2.widgets.TagSelect2* method), 38
 option_value () (*dal_select2_taggit.widgets.TaggitSelect2* method), 43
 options () (*dal_select2.widgets.TagSelect2* method), 38

P
 post () (*dal.views.BaseQuerySetView* method), 34
 post () (*dal_select2.views.Select2ListView* method), 37
 prepare_value () (*dal_contenttypes.fields.ContentTypeModelFieldMixin* method), 39
 prepare_value () (*dal_contenttypes.fields.ContentTypeModelMultipleFieldMixin* method), 39

Q
 q (*dal.views.ViewMixin* attribute), 34
 QuerySetSelectMixin (class in *dal.widgets*), 34
 QuerySetSequenceFieldMixin (class in *dal_queryset_sequence.fields*), 41
 QuerySetSequenceModelField (class in *dal_queryset_sequence.fields*), 41
 QuerySetSequenceModelMultipleField (class in *dal_queryset_sequence.fields*), 42
 QuerySetSequenceSelect (class in *dal_queryset_sequence.widgets*), 42

QuerySetSequenceSelect2 (class in *dal_select2_queryset_sequence.widgets*), 40
 QuerySetSequenceSelect2Multiple (class in *dal_select2_queryset_sequence.widgets*), 40
 QuerySetSequenceSelectMixin (class in *dal_queryset_sequence.widgets*), 42
 QuerySetSequenceSelectMultiple (class in *dal_queryset_sequence.widgets*), 42

R

raise_invalid_choice() (*dal_queryset_sequence.fields.QuerySetSequenceFieldMixin* method), 41
 render() (*dal.widgets.WidgetMixin* method), 35
 render_forward_conf() (*dal.widgets.WidgetMixin* method), 35
 render_options() (*dal.widgets.WidgetMixin* method), 35
 render_options() (*dal_select2_tagging.widgets.TaggingSelect2* method), 44
 render_options() (*dal_select2_taggit.widgets.TaggitSelect2* method), 43
 render_to_response() (*dal_select2.views.Select2ViewMixin* method), 37
 results() (*dal_select2.views.Select2ListView* method), 37

S

save() (*dal.forms.FutureModelForm* method), 36
 save_object_data() (*dal_contenttypes.fields.GenericModelMixin* method), 39
 save_relation_data() (*dal_genericm2m.fields.GenericM2MFieldMixin* method), 43
 save_relation_data() (*dal_gm2m.fields.GM2MFieldMixin* method), 43
 Select (class in *dal.widgets*), 34
 Select2 (class in *dal_select2.widgets*), 37
 Select2GroupListView (class in *dal_select2.views*), 36
 Select2ListChoiceField (class in *dal_select2.fields*), 38
 Select2ListCreateChoiceField (class in *dal_select2.fields*), 38
 Select2ListView (class in *dal_select2.views*), 36
 Select2Multiple (class in *dal_select2.widgets*), 37
 Select2QuerySetSequenceAutoView (class in *dal_select2_queryset_sequence.views*), 39
 Select2QuerySetSequenceView (class in *dal_select2_queryset_sequence.views*), 40

Select2QuerySetView (class in *dal_select2.views*), 37
 Select2Story (class in *dal_select2.test*), 38
 Select2ViewMixin (class in *dal_select2.views*), 37
 Select2WidgetMixin (class in *dal_select2.widgets*), 37
 SelectMultiple (class in *dal.widgets*), 34

T

TaggingSelect2 (class in *dal_select2_tagging.widgets*), 44
 TaggitSelect2 (class in *dal_select2_taggit.widgets*), 43
 TagSelect2 (class in *dal_select2.widgets*), 38
 to_python() (*dal_queryset_sequence.fields.QuerySetSequenceModelFieldMixin* method), 41

U

url (*dal.widgets.WidgetMixin* attribute), 34

V

validate() (*dal_select2.fields.Select2ListCreateChoiceField* method), 38
 value_from_datadict() (*dal_select2.widgets.TagSelect2* method), 38
 value_from_datadict() (*dal_select2_taggit.widgets.TaggitSelect2* method), 43
 value_from_object() (*dal_contenttypes.fields.GenericModelMixin* method), 39
 value_from_object() (*dal_genericm2m.fields.GenericM2MFieldMixin* method), 43
 value_from_object() (*dal_gm2m.fields.GM2MFieldMixin* method), 43
 ViewMixin (class in *dal.views*), 34

W

wait_script() (*dal_select2.test.Select2Story* method), 39
 WidgetMixin (class in *dal.widgets*), 34