django-autocomplete-light Documentation

Release 3.1.3

James Pic & contributors

November 12, 2017

Contents

CHAPTER 1

Features

- Python 2.7, 3.4, Django 1.8+ support,
- 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

- **Documentation** graciously hosted by RTFD
- Live demo graciously hosted by RedHat, thanks to PythonAnywhere for hosting it in the past,
- Video demo graciously hosted by Youtube,
- Mailing list graciously hosted by Google
- Git graciously hosted by GitHub,
- Package graciously hosted by PyPi,
- Continuous integration graciously hosted by Travis-ci
- **Online paid support** provided via HackHands,

CHAPTER 4

Basics

Install django-autocomplete-light v3

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-
```

Then, let Django find static file we need by adding to `INSTALLED_APPS`, **before** `django.contrib.admin`:

```
'dal',
'dal_select2',
'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 `$`.

Install the demo project

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

```
cd /tmp
virtualenv 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-
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
```

django-autocomplete-light tutorial

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.

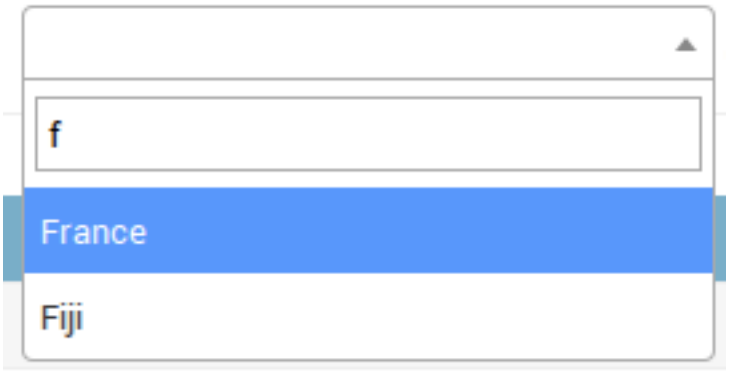
Create an autocomplete view

- 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 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.

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',
    ),
]
```

Ensure that the url can be reversed, ie:

```
./manage.py shell
In [1]: from django.core.urlresolvers import reverse

In [2]: 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.

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):
    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')
}
```

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
```

Using autocompletes outside the admin

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

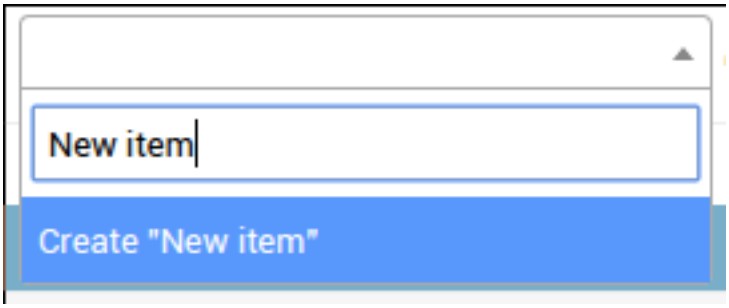
Ensure that jquery is loaded before `{{ form.media }}`:

```
{% extends 'base.html' %}

{% block content %}
<div>
    <form action="" method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" />
    </form>
</div>
{% endblock %}

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

{{ form.media }}
{% endblock %}
```

Creation of new choices in the autocomplete form

- Example source code: test_project/select2_one_to_one,
- Live demo: /admin/select2_one_to_one/testmodel/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.

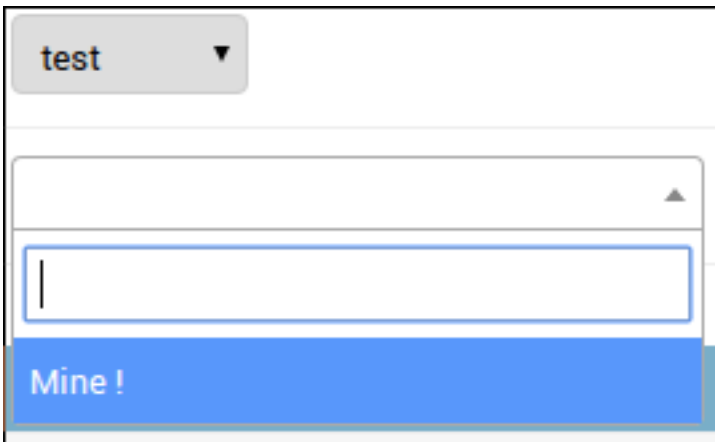
Note that creating objects is only allowed to staff users with add permission by default.

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

- Example source code: test_project/select2_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 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 fo 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.is_authenticated():
            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__istartswith=self.q)

        return qs
```

CHAPTER 5

External app support

Autocompletion for GenericForeignKey

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)

    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
```

View example for QuerySetSequence and Select2

We'll need a view that will provide results for the select2 frontend, and that uses QuerySetSequence as the backend. Let's try *Select2QuerySetSequenceView* for this:

```
from dal_select2_queryset_sequence.views import Select2QuerySetSequenceView

from queryset_sequence import QuerySetSequence
```

```
from your_models import Country, City


class LocationAutocompleteView(Select2QuerySetSequenceView):
    def get_queryset(self):
        countries = Country.objects.all()
        cities = City.objects.all()

        if self.q:
            countries = countries.filter(continent__incontains=self.q)
            cities = cities.filter(country__name__icontains=self.q)

        # Aggregate querysets
        qs = QuerySetSequence(countries, cities)

        if self.q:
            # This would apply the filter on all the querysets
            qs = qs.filter(name__icontains=self.q)

        # This will limit each queryset so that they show an equal number
        # of results.
        qs = self.mixup_querysets(qs)

        return qs
```

Register the view in urlpatterns as usual, ie.:

```
from .views import LocationAutocompleteView

urlpatterns = [
    url(
        r'^location-autocomplete/$',
        LocationAutocompleteView.as_view(),
        name='location-autocomplete'
    ),
]
```

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 *QuerySetSequenceSelect2* widget.

Also, we need a field that's able to use a QuerySetSequence for choices to do validation on a single model choice, we'll use *QuerySetSequenceModelField*.

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

Result:

```
class TestForm(autocomplete.FutureModelForm):
    location = dal_queryset_sequence.fields.QuerySetSequenceModelField(
        queryset=autocomplete.QuerySetSequence(
            Country.objects.all(),
            City.objects.all(),
        ),
        required=False,
        widget=dal_select2_queryset_sequence.widgets.QuerySetSequenceSelect2('location-autocomplete')
```

```
    )

    class Meta:
        model = TestModel
```

Autocompletion for django-gm2m's GM2MField

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
```

View example

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

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*.

Finnaly, 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',)
```

Autocompletion for django-generic-m2m's RelatedObjectsDescriptor

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
```

View example

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

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*.

Finnaly, 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',)
```

Autocompletion for django-tagging's TagField

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
```

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__istartswith=self.q)

        return qs
```

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

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'
            )
        }
```

Autocompletion for django-taggit's TaggableManager

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
```

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__istartswith=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`.

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: TaggeableManager - 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'
            )
        }
```

CHAPTER 6

API

dal: django-autocomplete-light3 API

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.

`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.

`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.

`forwarded`
Dict of field values that were forwarded from the form, may be used to filter autocompletion 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`.

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.

build_attrs(**args*, ***kwargs*)
Build HTML attributes for the widget.

filter_choices_to_render(*selected_choices*)
Replace self.choices with selected_choices.

render_options(**args*)
Django-compatibility method for option rendering.

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

Fields

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 form 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** writting 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** writting the database, when `instance.pk` is necessarely 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.

`save`(*commit=True*)
Backport from Django 1.9+ for 1.8.

dal_select2: Select2 support for DAL

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

Views

Select2 view implementation.

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

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

Widgets

Select2 widget implementation module.

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.**Select2WidgetMixin**
Mixin for Select2 widgets.

class Media
Automatically include static files for the admin.

class dal_select2.widgets.**TagSelect2**(*url=None*, *forward=None*, **args*, ***kwargs*)
Select2 in tag mode.

build_attrs(**args*, ***kwargs*)
Automatically set data-tags=1.

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.

Test tools

Helpers for DAL user story based tests.

class dal_select2.test.**Select2Story**
Define Select2 CSS selectors.

dal_contenttypes: GenericForeignKey support

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.

dal_select2_queryset_sequence: Select2 for QuerySetSequence choices

Views

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

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.

Wigets

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*, *forward=None*, **args*, ***kwargs*)
Single model select for a generic select2 autocomplete.

class `dal_select2_queryset_sequence.widgets.`**QuerySetSequenceSelect2Multiple**(*url=None*, *forward=None*, **args*, ***kwargs*)
Multiple model select for a generic select2 autocomplete.

dal_queryset_sequence: QuerySetSequence choices

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_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.

Fields

Autocomplete fields for QuerySetSequence choices.

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=u'———'*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=u''*, *to_field_name=None*, *limit_choices_to=None*, **args*, ***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*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=u''*, **args*, ***kwargs*)
ModelMultipleChoiceField with support for QuerySetSequence choices.

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*, *forward=None*, **args*, ***kwargs*)

Select widget for QuerySetSequence choices.

class dal_queryset_sequence.widgets.**QuerySetSequenceSelectMixin**(*url=None*, *forward=None*, **args*, ***kwargs*)

Support QuerySetSequence in WidgetMixin.

filter_choices_to_render(*selected_choices*)

Overwrite self.choices to exclude unselected values.

class dal_queryset_sequence.widgets.**QuerySetSequenceSelectMultiple**(*url=None*, *forward=None*, **args*, ***kwargs*)

SelectMultiple widget for QuerySetSequence choices.

dal_gm2m_queryset_sequence

Fields

Form fields for using django-gm2m with QuerySetSequence.

class dal_gm2m_queryset_sequence.fields.**GM2MQuerySetSequenceField**(*queryset*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=u''*, **args*, ***kwargs*)

Form field for QuerySetSequence to django-generic-m2m relation.

dal_genericm2m_queryset_sequence

Fields

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

class dal_genericm2m_queryset_sequence.fields.**GenericM2MQuerySetSequenceField**(*queryset*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=u''*, **args*, ***kwargs*)

Autocomplete field for GM2MField() for QuerySetSequence choices.

dal_gm2m: django-gm2m support

Fields

GM2MField support for autocomplete fields.

class dal_gm2m.fields.**GM2MFieldMixin**
GM2MField ror 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.

dal_genericm2m: django-genericm2m support

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.

dal_select2_taggit: django-taggit support

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.

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

dal_select2_tagging: django-tagging support

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

B

C

D

F

G