django-autocomplete-light Documentation

Release 3.0.0

James Pic & contributors

February 04, 2016

Contents

CHAPTER 1

Features

- Python 2.7+, Django 1.8+ complete 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

Resources

Resources include:

- **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 3

Basics

3.1 Install django-autocomplete-light v3

3.1.1 Install in your project

Install version 3 with `pip install`:

```
pip install -e git+https://github.com/yourlabs/django-autocomplete-light.git@v3#egg=dal
```

Then, let Django find static file we need by adding to `INSTALLED_APPS`:

```
'dal',
'dal_select2',
```

3.1.2 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@v3#egg=django-autocomple
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
```

3.2 django-autocomplete-light tutorial

3.2.1 Overview

Autocompletes are based on 3 moving parts:

- widget, does the initial rendering,
- javascript widget initialization code, to trigger the autocomplete,

- and a view used by the widget script to get results from.

3.2.2 Create an autocomplete view

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.is_authenticated():
            return Country.objects.none()

        qs = Country.objects.all()

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

        return self.q
```

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

3.2.3 Register the autocomplete view

Create a named url for the view, ie:

```
from your_countries_app.views import CountryAutocomplete

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

3.2.4 Use the view in a Form widget

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*:

```
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.ManyToMany('your_countries_app.country')
```

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

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

3.2.5 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
```

3.3 Creation of new choices in the autocomplete form

3.3.1 Auto-creation of one-to-one and one-to-many (foreign-key) relations

By default, Django's ModelChoiceField is used for validation and it only allows to choose existing choices. To enable creating choices during validation, we can use the CreateModelField'' form field, ie:

```
class YourCountryCreateField(autocomplete.CreateModelField):
    def create_value(self, value):
        return Country.objects.create(name=value).pk


class PersonForm(forms.ModelForm):
    birth_country = YourCountryCreateField(
        required=False,  # leave out if your model field doesn't have blank=True
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2(url='country-autocomplete')
    )

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

3.3.2 Auto-creation of many-to-many relations

Note that for we could do the same for a multiple relation, using autocomplete.CreateModelMultipleField and autocomplete.ModelSelect2Multiple, ie.:

```
class YourCountryCreateField(autocomplete.CreateModelMultipleField):
    def create_value(self, value):
        return Country.objects.create(name=value).pk


class PersonForm(forms.ModelForm):
    visited_countries = YourCountryCreateMultipleField(
        required=False,  # leave out if your model field doesn't have blank=True
        queryset=Country.objects.all(),
        widget=autocomplete.ModelSelect2Multiple(url='country-autocomplete')
    )

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

3.3.3 Deduplicating creation code with mixins

Of course, we could use a mixin to avoid duplicating code if we wanted both, ie.:

```
class CountryCreateFieldMixin(object):
    def create_value(self, value):
        return Country.objects.create(name=value).pk


class CountryCreateField(CountryCreateFieldMixin,
```

```
                                    autocomplete.CreateModelField):
    pass


class CountryCreateMultipleField(CountryCreateFieldMixin,
                                    autocomplete.CreateModelMultipleField):
    pass
```

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

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'])
        }
```

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 self.q
```

CHAPTER 4

External app support

4.1 Autocompletion for GenericForeignKey

4.1.1 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)

    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
```

4.1.2 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(guitars, trumpets)

        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'
    ),
]
```

4.1.3 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 `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 = autocomplete.QuerySetSequenceModelField(
        queryset=autocomplete.QuerySetSequence(
            Country.objects.all(),
            City.objects.all(),
        ),
        required=False,
        widget=autocomplete.QuerySetSequenceSelect2('location-autocomplete'),
```

```
    )

    class Meta:
        model = TestModel
```

4.2 Autocompletion for django-gm2m's GM2MField

4.2.1 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
```

4.2.2 View example

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

4.2.3 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`.

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',)
```

4.3 Autocompletion for django-generic-m2m's RelatedObjectsDescriptor

4.3.1 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
```

4.3.2 View example

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

4.3.3 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`.

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',)
```

4.4 Autocompletion for django-taggit's TaggableManager

4.4.1 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
```

4.4.2 View example

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

```
from dal import autocomplete

from 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.is_authenticated():
            return Tag.objects.none()

        qs = Tag.objects.all()

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

        return self.q
```

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

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

4.4.3 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 *TagSelect2*.

Also, we need a field that works with a queryset of Tag and use the TaggableManager: `TaggitField`.

Finnaly, we can't use Django's ModelForm because it django-taggit's field is made to be edited in a text input with a comma-separated list of fields, which isn't what Select2 supports even in its tag mode. So, we'll use *FutureModelForm*.

Example:

```
class TestForm(autocomplete.FutureModelForm):
    tags = autocomplete.TaggitField(
        required=False,
        widget=autocomplete.TagSelect2(url='your-tag-autocomplete-url'),
    )

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

4.5 Autocompletion for django-tagulous TagField

Note that django-tagulous provides autocompletion features. Check them out, if it doesn't work for you or whatever reason then feel free to use dal_tagulous.

4.5.1 Model example

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

```
from django.db import models

from tagulous.models import TagField


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

    tags = TagField()

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

4.5.2 View example

The *QuerySet view* works here too: we're relying on Select2 and a QuerySet of Tag objects. However, in django-tagulous, a specific Tag model is made for every instance of the field. So we have to get the Tag model class dynamically:

```
from dal import autocomplete
```

```
class TagAutocomplete(autocomplete.Select2QuerySetView):
    def get_queryset(self):
        # Get the tag model dynamically
        Tag = TestModel.tags.tag_model

        # Don't forget to filter out results depending on the visitor !
        if not self.request.is_authenticated():
            return Tag.objects.none()

        qs = Tag.objects.all()

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

        return self.q
```

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

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

4.5.3 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 *`TagSelect2`*.

Also, we need a field that works with a queryset of tagulous tag, and is able to update tagulous TagField: `TagulousField`.

Finnaly, we can't use Django's ModelForm because it django-taggit's field is made to be edited in a text input with a comma-separated list of fields, which isn't what Select2 supports even in its tag mode. So, we'll use *`FutureModelForm`*.

Example:

```
class TestForm(autocomplete.FutureModelForm):
    tags = autocomplete.TagulousField(
        required=False,
        queryset=TestModel.test.tag_model.objects.all(),
        widget=autocomplete.TagSelect2(url='your-view-url-name'),
    )

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

CHAPTER 5

API

5.1 dal: django-autocomplete-light3 API

5.1.1 Views

Base views for autocomplete widgets.

class `dal.views.`**`BaseQuerySetView`**(***kwargs*)
Base view to get results from a QuerySet.

`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_more`(*context*)
For widgets that have infinite-scroll feature.

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.

`get`(*request*, **args*, ***kwargs*)
Wrap around get to set *`forwarded`*.

5.1.2 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.

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.

5.1.3 Fields

Form fields which may create missing models.

class dal.fields.**CreateModelField**(*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*)
This field allows creating instances.

clean(*value*)
Try the default clean method, else create if allowed.

class dal.fields.**CreateModelFieldMixin**
Mixin for autocomplete form fields with create power.

create_value(*value*)
Create and return a model from a user value.

widget_attrs(*widget*)
Override to data-autocomplete-light-create to ‘true’.

class dal.fields.**CreateModelMultipleField**(*queryset*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=u''*, **args*, ***kwargs*)
This field allows creating instances.

clean(*value*)
Try the default clean method, else create if allowed.

5.2 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 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.

5.3 dal_select2: Select2 support for DAL

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

5.3.1 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.

5.3.2 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*)
ModelSelect2Multiple for tags.

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

5.3.3 Test tools

Helpers for DAL user story based tests.

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

5.4 dal_contenttypes: GenericForeignKey support

5.4.1 Fields

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

5.5 dal_select2_queryset_sequence: Select2 for QuerySetSequence choices

5.5.1 Views

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

5.5.2 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.

5.6 dal_queryset_sequence: QuerySetSequence choicse

5.6.1 Views

View that supports QuerySetSequence.

5.6.2 Fields

Autocomplete fields for QuerySetSequence choices.

5.6.3 Widgets

Widget mixin that only renders selected options with QuerySetSequence.

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

5.7 dal_gm2m_queryset_sequence

5.7.1 Fields

Form fields for using django-gm2m with QuerySetSequence.

5.8 dal_genericm2m_queryset_sequence

5.8.1 Fields

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

5.9 dal_gm2m: django-gm2m support

5.9.1 Fields

GM2MField support for autocomplete fields.

5.10 dal_genericm2m: django-genericm2m support

5.10.1 Fields

django-generic-m2m field mixin for FutureModelForm.

5.11 dal_taggit: django-taggit support

5.11.1 Fields

Autocomplete form fields for django-taggit.

5.12 dal_tagulous: django-tagulous support

5.12.1 Fields

Tagulous TagField support.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

d

B

C

D

F

G

H

M

Q

R

S

T

U

V

W