# django-ajax-selects Documentation

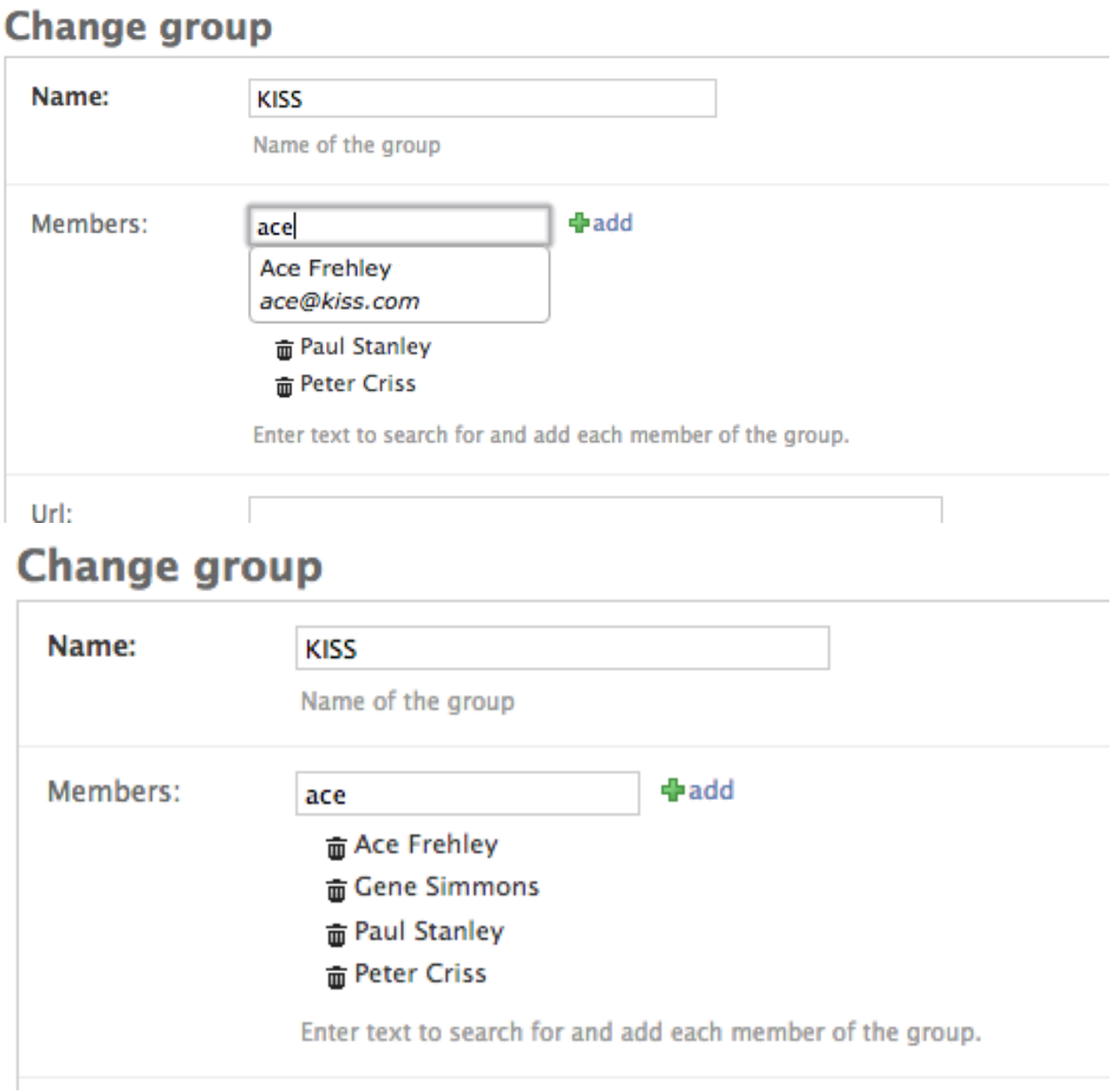
***Release 1.4.0***

**Chris Sattinger**

November 06, 2015

Contents

Edit *ForeignKey*, *ManyToManyField* and *CharField* in Django Admin using jQuery UI AutoComplete.



CHAPTER 1

# Quick Usage

Define a lookup channel:

```
# yourapp/lookups.py
from ajax_select import register, LookupChannel
from .models import Tag

@register('tags')
class TagsLookup(LookupChannel):

    model = Tag

    def get_query(self, q, request):
        return self.model.objects.filter(name__icontains=q).order_by('name')[:50]

    def format_item_display(self, item):
        return u"<span class='tag'>%s</span>" % item.name
```

Add field to a form:

```
# yourapp/forms.py
class DocumentForm(ModelForm):

    class Meta:
        model = Document

    tags = AutoCompleteSelectMultipleField('tags')
```

CHAPTER 2

# Fully customizable

- search query
- query other resources besides Django ORM
- format results with HTML
- custom styling
- customize security policy
- customize UI
- integrate with other UI elements on page using the javascript API

CHAPTER 3

# Assets included by default

- ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js
- code.jquery.com/ui/1.10.3/jquery-ui.js
- code.jquery.com/ui/1.10.3/themes/smoothness/jquery-ui.css

CHAPTER 4

# Compatibility

- Django >=1.5, <=1.9
- Python >=2.7, <=3.4

CHAPTER 5

# Index

## 5.1 Install

Install:

```
pip install django-ajax-selects
```

Add the app:

```
# settings.py
INSTALLED_APPS = (
    ...
    'ajax_select',  # <-   add the app
    ...
)
```

Include the urls in your project:

```
# urls.py
from django.conf.urls import url, include
from django.conf.urls.static import static
from django.contrib import admin
from django.conf import settings
from ajax_select import urls as ajax_select_urls

admin.autodiscover()

urlpatterns = [

    # place it at whatever base url you like
    url(r'^ajax_select/', include(ajax_select_urls)),

    url(r'^admin/', include(admin.site.urls)),
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

Write a LookupChannel to specify the models, search queries, formatting etc. and register it with a channel name:

```
from ajax_select import register, LookupChannel
from .models import Tag

@register('tags')
class TagsLookup(LookupChannel):

    model = Tag
```

```
    def get_query(self, q, request):
        return self.model.objects.filter(name=q)

    def format_item_display(self, item):
        return u"<span class='tag'>%s</span>" % item.name
```

If you are using Django >= 1.7 then it will automatically loaded on startup. For previous Djangos you can import them manually to your urls or views.

Add ajax lookup fields in your admin.py:

```
from django.contrib import admin
from ajax_select import make_ajax_form
from .models import Document

@admin.register(Document)
class DocumentAdmin(AjaxSelectAdmin):

    form = make_ajax_form(Document, {
        # fieldname: channel_name
        'tags': 'tags'
    })
```

Or add the fields to a ModelForm:

```
# forms.py
from ajax_select.fields import AutoCompleteSelectField, AutoCompleteSelectMultipleField

class DocumentForm(ModelForm):

    class Meta:
        model = Document

    category = AutoCompleteSelectField('categories', required=False, help_text=None)
    tags = AutoCompleteSelectMultipleField('tags', required=False, help_text=None)

# admin.py
from django.contrib import admin
from .forms import DocumentForm
from .models import Document

@admin.register(Document)
class DocumentAdmin(AjaxSelectAdmin):
    form = DocumentForm
```

## 5.2 Lookup Channels

A LookupChannel defines how to search and how to format found objects for display in the interface.

LookupChannels are registered with a "channel name" and fields refer to that name.

You may have only one LookupChannel for a model, or you might define several for the same Model each with different queries, security policies and formatting styles.

Custom templates can be created for channels. This enables adding extra javascript or custom UI. See Customizing Templates

### 5.2.1 lookups.py

Write your LookupChannel classes in a file name *yourapp/lookups.py*

(note: inside your app, not your top level project)

Use the @register decorator to register your LookupChannels by name

*example/lookups.py*:

```
from ajax_select import register, LookupChannel

@register('things')
class ThingsLookup(LookupChannel):

    model = Things

    def get_query(self, q, request):
          return self.model.objects.filter(title__icontains=q).order_by('title')
```

If you are using Django >= 1.7 then all *lookups.py* in all of your apps will be automatically imported on startup.

If Django < 1.7 then you can import each of your lookups in your views or urls. Or you can register them in settings (see below).

### 5.2.2 Customize

**class** `ajax_select.lookup_channel.`**`LookupChannel`**

Subclass this, setting the model and implementing methods to taste.

**`model`**

*Model*

The Django Model that this lookup channel will search for.

**`plugin_options`**

*dict*

Options passed to jQuery UI plugin that are specific to this channel.

**`min_length`**

*int*

Minimum number of characters user types before a search is initiated.

This is passed to the jQuery plugin_options. It is used in jQuery's UI when filtering results from its own cache.

It is also used in the django view to prevent expensive database queries. Large datasets can choke if they search too often with small queries. Better to demand at least 2 or 3 characters.

**`can_add`**(*user*, *other_model*)

Check if the user has permission to add a ForeignKey or M2M model.

This enables the green popup + on the widget. Default implentation is the standard django permission check.

**Parameters**

- **`(User)`** (*user*) –
- **`other_model`** (*Model*) – the ForeignKey or M2M model to check if the User can add.

**Returns** bool

**check_auth**(*request*)
By default only request.user.is_staff have access.

This ensures that nobody can get your data by simply knowing the lookup URL.

This is called from the ajax_lookup view.

Public facing forms (outside of the Admin) should implement this to allow non-staff to use this LookupChannel.

**Parameters** **(Request)** (*request*) –

**Raises** PermissionDenied –

**format_item_display**(*obj*)
(HTML) format item for displaying item in the selected deck area.

**Parameters** **obj** (*Model*) –

**Returns** formatted string, may contain HTML.

**Return type** str

**format_match**(*obj*)
(HTML) Format item for displaying in the dropdown.

**Parameters** **obj** (*Model*) –

**Returns** formatted string, may contain HTML.

**Return type** str

**get_objects**(*ids*)
This is used to retrieve the currently selected objects for either ManyToMany or ForeignKey.

Note that the order of the ids supplied for ManyToMany fields is dependent on how the objects manager fetches it. ie. what is returned by *YourModel.{fieldname}_set.all()*

In most situations (especially postgres) this order is indeterminate – not the order that you originally added them in the interface. See Ordered ManyToMany fields for a solution to this.

**Parameters** **ids** (*list*) – list of primary keys

**Returns** list of Model objects

**Return type** list

**get_query**(*q*, *request*)
Return a QuerySet searching for the query string *q*.

Note that you may return any iterable so you can return a list or even use yield and turn this method into a generator.

**Parameters**

- **q** (*str, unicode*) – The query string to search for.
- **request** (*Request*) – This can be used to customize the search by User or to use additional GET variables.

**Returns** iterable of related_models

**Return type** (QuerySet, list, generator)

**get_result**(*obj*)
The text result of autocompleting the entered query.

For a partial string that the user typed in, each matched result is here converted to the fully completed text.

This is currently displayed only for a moment in the text field after the user has selected the item. Then the item is displayed in the item_display deck and the text field is cleared.

**Parameters** **obj** (*Model*) –

**Returns** The object as string

**Return type** str

### 5.2.3 settings.py

Versions previous to 1.4 loaded the LookupChannels according to *settings.AJAX_LOOKUP_CHANNELS*

This will still work. Your LookupChannels will continue to load without having to add them with the new @register decorator.

Example:

```
# settings.py

  AJAX_LOOKUP_CHANNELS = {
      # auto-create a channel named 'person' that searches by name on the model Person
      # str: dict
      'person': {'model': 'example.person', 'search_field': 'name'}

      # specify a lookup to be loaded
      # str: tuple
      'song': ('example.lookups', 'SongLookup'),

      # delete a lookup channel registered by an app/lookups.py
      # str: None
      'users': None
  }
```

One situation where it is still useful: if a resuable app defines a LookupChannel and you want to override that or turn it off. Pass None as in the third example above.

Anything in *settings.AJAX_LOOKUP_CHANNELS* overwrites anything previously registered by an app.

## 5.3 Admin

If your application does not otherwise require a custom Form class then you can create the form directly in your admin using *make_ajax_form*

ajax_select.helpers.**make_ajax_form**(*model*, *fieldlist*, *superclass=<class 'django.forms.models.ModelForm'>*, *show_help_text=False*, ***kwargs*)
Creates a ModelForm subclass with AutoComplete fields.

**Parameters**

- **model** (*type*) – Model class for which you are making the ModelForm
- **fieldlist** (*dict*) – {field_name -> channel_name, ...}

- **superclass** (*type*) – optional ModelForm superclass
- **show_help_text** (*bool*) – suppress or show the widget help text

**Returns** a ModelForm suitable for use in an Admin

**Return type** ModelForm

Usage:

```
from django.contrib import admin
from ajax_select import make_ajax_form
from yourapp.models import YourModel

@admin.register(YourModel)
class YourModelAdmin(Admin):

    form = make_ajax_form(YourModel, {
        'contacts': 'contact',  # ManyToManyField
        'author':'contact'      # ForeignKeyField
    })
```

Where 'contacts' is a ManyToManyField specifying to use the lookup channel 'contact' and 'author' is a ForeignKeyField specifying here to also use the same lookup channel 'contact'

## 5.4 Forms

Forms can be used either for an Admin or in normal Django views.

Subclass ModelForm as usual and define fields:

```
from ajax_select.fields import AutoCompleteSelectField, AutoCompleteSelectMultipleField

class DocumentForm(ModelForm):

    class Meta:
        model = Document

    category = AutoCompleteSelectField('categories', required=False, help_text=None)
    tags = AutoCompleteSelectMultipleField('tags', required=False, help_text=None)
```

### 5.4.1 make_ajax_field

There is also a helper method available here.

ajax_select.helpers.**make_ajax_field**(*related_model*, *fieldname_on_model*, *channel*, *show_help_text=False*, ***kwargs*)

Makes an AutoComplete field for use in a Form.

**Parameters**

- **related_model** (*Model*) – model of the related object
- **fieldname_on_model** (*str*) – field name on the model being edited
- **channel** (*str*) – channel name of a registered LookupChannel

- **show_help_text** (*bool*) – show or supress help text below the widget Django admin will show help text below the widget, but not for ManyToMany inside of admin inlines This setting will show the help text inside the widget itself.
- **kwargs** – optional args
  - **help_text: default is the model db field's help_text.** None will disable all help text
  - label: default is the model db field's verbose name
  - required: default is the model db field's (not) blank

**Returns** field

**Return type** (AutoCompleteField, AutoCompleteSelectField, AutoCompleteSelectMultipleField)

Example:

```
from ajax_select import make_ajax_field

class DocumentForm(ModelForm):

    class Meta:
        model = Document

    category  = make_ajax_field(Category, 'categories', 'category', help_text=None)
    tags  = make_ajax_field(Tag, 'tags', 'tags', help_text=None)
```

### 5.4.2 FormSet

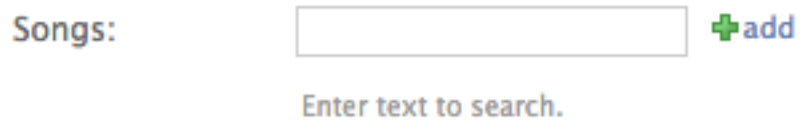
There is possibly a better way to do this, but here is an initial example:

*forms.py*:

```
from django.forms.models import modelformset_factory
from django.forms.models import BaseModelFormSet
from ajax_select.fields import AutoCompleteSelectMultipleField, AutoCompleteSelectField

from models import Task

# create a superclass
class BaseTaskFormSet(BaseModelFormSet):

    # that adds the field in, overwriting the previous default field
    def add_fields(self, form, index):
        super(BaseTaskFormSet, self).add_fields(form, index)
        form.fields["project"] = AutoCompleteSelectField('project', required=False)

# pass in the base formset class to the factory
TaskFormSet = modelformset_factory(Task, fields=('name', 'project', 'area'), extra=0, formset=BaseTas
```

## 5.5 Admin add popup

This enables that green + icon:




The user can click, a popup window lets them create a new object, they click save, the popup closes and the AjaxSelect field is set.

Your *Admin* must inherit from *AjaxSelectAdmin*:

```
class YourModelAdmin(AjaxSelectAdmin):
    pass
```

or be used as a mixin class:

```
class YourModelAdmin(AnotherAdminClass, AjaxSelectAdmin):
    pass
```

or you must implement *get_form* yourself:

```
from ajax_selects.fields import autoselect_fields_check_can_add

class YourModelAdmin(admin.ModelAdmin):

  def get_form(self, request, obj=None, **kwargs):
      form = super(YourModelAdmin, self).get_form(request, obj, **kwargs)
      autoselect_fields_check_can_add(form, self.model, request.user)
      return form
```

The *User* must also have permission to create an object of that type, as determined by the standard Django permission system. Otherwise the resulting pop up will just deny them.

### 5.5.1 Custom Permission Check

You could implement a custom permission check in the *LookupChannel*:

```
from permissionz import permissions

class YourLookupChannel(LookupChannel):

    def can_add(self, user, model):
        return permissions.has_perm('can_add', model, user)
```

### 5.5.2 Inline forms in the Admin

If you are using ajax select fields on an Inline you can use these superclasses:

> from ajax_select.admin import AjaxSelectAdminTabularInline, AjaxSelectAdminStackedInline # or use this mixin if already have a superclass from ajax_select.admin import AjaxSelectAdminInlineFormset-Mixin

## 5.6 Media Assets

If *jQuery* or *jQuery.ui* are not already loaded on the page, then these will be loaded from CDN:

```
ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js
code.jquery.com/ui/1.10.3/jquery-ui.js
code.jquery.com/ui/1.10.3/themes/smoothness/jquery-ui.css
```

If you want to prevent this and load your own then set:

```
# settings.py
AJAX_SELECT_BOOTSTRAP  = False
```

### 5.6.1 Customizing the style sheet

By default *css/ajax_select.css* is included by the Widget's media. This specifies a simple basic style.

If you would prefer not to have *css/ajax_select.css* loaded at all then you can implement your own *yourapp/static/ajax_select/css/ajax_select.css* and put your app before *ajax_select* in *INSTALLED_APPS*.

Your version will take precedence and Django will serve your *css/ajax_select.css*

The markup is simple and you can just add more css to override unwanted styles.

The trashcan icon comes from the jQueryUI theme by the css classes:

```
"ui-icon ui-icon-trash"
```

The following css declaration:

```
.results_on_deck .ui-icon.ui-icon-trash { }
```

would be "stronger" than jQuery's style declaration and thus you could make trash look less trashy.

The loading indicator is in *ajax_select/static/ajax_select/images/loading-indicator.gif*

*yourapp/static/ajax_select/images/loading-indicator.gif* would override that.

## 5.7 Customizing Templates

Each form field widget is rendered using a template:

- autocomplete.html
- autocompleteselect.html
- autocompleteselectmultiple.html

You may write a custom template for your channel:

- yourapp/templates/ajax_select/{channel}_autocomplete.html
- yourapp/templates/ajax_select/{channel}_autocompleteselect.html
- yourapp/templates/ajax_select/{channel}_autocompleteselectmultiple.html

And customize these blocks:

```
{% block extra_script %}
<script type="text/javascript">
    $("#{{html_id}}_on_deck").bind('added', function() {
      var id = $("#{{html_id}}").val();
      console.log('added id:' + id );
    });
    $("#{{html_id}}_on_deck").bind('killed', function() {
```

```
      var current = $("#{{html_id}}").val()
      console.log('removed, current is:' + current);
    });
</script>
{% endblock %}

{% block help %}
<p>
  You could put additional UI or help text here.
</p>
{% endblock %}
```

## 5.8 jQuery Plugin Options

https://jqueryui.com/autocomplete/

- minLength The minimum number of characters a user must type before a search is performed. min_length is also an attribute of the LookupChannel so you need to also set it there.
- autoFocus
- delay
- disabled
- position
- source - By default this is the ajax_select view. Setting this would overide the normal url used for lookups (*ajax_select.views.ajax_lookup*). This could be used to add URL custom query params.

See http://docs.jquery.com/UI/Autocomplete#options

Setting plugin options:

```
from ajax_select.fields import AutoCompleteSelectField

class DocumentForm(ModelForm):

    category = AutoCompleteSelectField('category',
        required=False,
        help_text=None,
        plugin_options={'autoFocus': True, 'minLength': 4})
```

This Python dict will be passed as JavaScript to the plugin.

## 5.9 jQuery events

Triggers are a great way to keep code clean and untangled.

see: http://docs.jquery.com/Events/trigger

If you need integrate with other javascript on your page, you can write a custom template for your channel and hook into the 'added' and 'killed' events.

'killed' means 'removed' (silly name, sorry)

Two triggers/signals are sent: 'added' and 'killed'. These are sent to the *$("#{{html_id}}_on_deck")* element. That is the area that surrounds the currently selected items.

Extend the template, implement the extra_script block and bind functions that will respond to the trigger:

AutoCompleteSelectMultipleField:

```
// yourapp/templates/ajax_select/autocompleteselectmultiple_{channel}.html

{% block extra_script %}
<script type="text/javascript">
    $("#{{html_id}}_on_deck").bind('added', function() {
      var id = $("#{{html_id}}").val();
      console.log('added id:' + id );
    });
    $("#{{html_id}}_on_deck").bind('killed', function() {
      var current = $("#{{html_id}}").val()
      console.log('removed, current is:' + current);
    });
</script>
{% endblock %}
```

AutoCompleteSelectField:

```
// yourapp/templates/ajax_select/autocompleteselect_{channel}.html

{% block extra_script %}
<script type="text/javascript">
  $("#{{html_id}}_on_deck").bind('added', function() {
    var id = $("#{{html_id}}").val();
    console.log('added id:' + id );
  });
  $("#{{html_id}}_on_deck").bind('killed', function() {
    console.log('removed');
  });
</script>
{% endblock %}
```

AutoCompleteField (simple text field):

```
// yourapp/templates/ajax_select/autocomplete_{channel}.html

{% block extra_script %}
<script type="text/javascript">
  $('#{{ html_id }}').bind('added', function() {
    var entered = $('#{{ html_id }}').val();
    console.log('text entered:' + entered);
  });
</script>
{% endblock %}
```

There is no remove with this one as there is no kill/delete button in a simple text auto-complete. The user may clear the text themselves but there is no javascript involved. Its just a text field.

### 5.9.1 Re-initializing

If you are dynamically adding forms to the page (eg. by loading a page using Ajax) then you can trigger the newly added ajax selects widgets to be activated:

```
$(window).trigger('init-autocomplete');
```

## 5.10 Ordered ManyToMany fields

When re-editing a previously saved model that has a ManyToMany field, the order of the recalled ids can be somewhat random.

The user sees Arnold, Bosco, Cooly in the interface; saves; comes back later to edit it and he sees Bosco, Cooly, Arnold. So he files a bug report.

### 5.10.1 Problem

Given these models:

```
class Agent(models.Model):
    name = models.CharField(blank=True, max_length=100)

class Apartment(models.Model):
    agents = models.ManyToManyField(Agent)
```

When the AutoCompleteSelectMultipleField saves it does so by saving each relationship in the order they were added in the interface.

But when Django ORM retrieves them, the order is not guaranteed:

```
# This query does not have a guaranteed order (especially on postgres)
# and certainly not the order that we added them.
apartment.agents.all()

# This retrieves the joined objects in the order of the join table pk
# and thus gets them in the order they were added.
apartment.agents.through.objects.filter(apt=self).select_related('agent').order_by('id')
```

### 5.10.2 Solution

A proper solution would be to use a separate Through model, an order field and the ability to drag the items in the interface to rearrange. But a proper Through model would also introduce extra fields and that would be out of the scope of ajax_selects.

However this method will also work.

Make a custom ManyToManyField:

```
from django.db import models

class AgentOrderedManyToManyField(models.ManyToManyField):

    """This fetches from the join table, then fetches the Agents in the fixed id order."""

    def value_from_object(self, object):
        rel = getattr(object, self.attname)
        qry = {self.related.var_name: object}
        qs = rel.through.objects.filter(**qry).order_by('id')
        aids = qs.values_list('agent_id', flat=True)
        agents = dict((a.pk, a) for a in Agent.objects.filter(pk__in=aids))
        return [agents[aid] for aid in aids if aid in agents]

class Agent(models.Model):
    name = models.CharField(blank=True, max_length=100)
```

```
class Apartment(models.Model):
    agents = AgentOrderedManyToManyField()
```

## 5.11 Outside of the Admin

ajax_selects does not need to be in a Django admin.

Popups will still use an admin view (the registered admin for the model being added), even if the form from where the popup was launched does not.

In your view, after creating your ModelForm object:

```
autoselect_fields_check_can_add(form, model, request.user)
```

This will check each widget and enable the green + for them iff the User has permission.

## 5.12 Example App

Clone this repository and see the example app for a full working admin site with many variations and comments. It installs quickly using virtualenv and sqllite and comes fully configured.

install:

```
cd example
./install.sh 1.8.5
./manage.py runserver
```

## 5.13 Upgrading from previous versions

1.4

### 5.13.1 Custom Templates

Move your custom templates from:

```
yourapp/templates/channel_autocomplete.html
yourapp/templates/channel_autocompleteselect.html
yourapp/templates/channel_autocompleteselectmultiple.html
```

to:

```
yourapp/templates/ajax_select/channel_autocomplete.html
yourapp/templates/ajax_select/channel_autocompleteselect.html
yourapp/templates/ajax_select/channel_autocompleteselectmultiple.html
```

And change your extends from:

```
{% extends "autocompleteselect.html" %}
```

to:

```
{% extends "ajax_select/autocompleteselect.html" %}
```

### 5.13.2 Removed options

**make_ajax_field:** show_m2m_help -> show_help_text

### 5.13.3 settings

LookupChannels are still loaded from *settings.AJAX_LOOKUP_CHANNELS* as previously.

If you are on Django >= 1.7 you may switch to using the @register decorator and you can remove that setting.

## 5.14 Contributing

### 5.14.1 Testing

For any pull requests you should run the unit tests first. Travis CI will also run all tests across all supported versions against your pull request and github will show you the failures.

Its much faster to run them yourself locally.:

```
pip install -r requirements-test.txt
```

run tox::

```
make test

# or just
tox
```

With all supported combinations of Django and Python.

You will need to have different Python interpreters installed which you can do with:

https://github.com/yyuu/pyenv

It will skip tests for any interpreter you don't have installed.

Most importantly you should have at least 2.7 and 3.4

### 5.14.2 Documentation

Docstrings use Google style: http://sphinx-doc.org/ext/example_google.html#example-google

## 5.15 ajax_select

### 5.15.1 ajax_select package

#### Submodules

#### ajax_select.admin module

**class** ajax_select.admin.**AjaxSelectAdmin**(*model*, *admin_site*)
Bases: django.contrib.admin.options.ModelAdmin

in order to get + popup functions subclass this or do the same hook inside of your get_form

**get_form**(*request*, *obj=None*, ***kwargs*)

**media**

**class** ajax_select.admin.**AjaxSelectAdminInlineFormsetMixin**
Bases: object

**get_formset**(*request*, *obj=None*, ***kwargs*)

**class** ajax_select.admin.**AjaxSelectAdminStackedInline**(*parent_model*, *admin_site*)
Bases: *ajax_select.admin.AjaxSelectAdminInlineFormsetMixin*, django.contrib.admin.options.StackedInline

**media**

**class** ajax_select.admin.**AjaxSelectAdminTabularInline**(*parent_model*, *admin_site*)
Bases: *ajax_select.admin.AjaxSelectAdminInlineFormsetMixin*, django.contrib.admin.options.TabularInline

**media**

#### ajax_select.apps module

**class** ajax_select.apps.**AjaxSelectConfig**(*app_name*, *app_module*)
Bases: django.apps.config.AppConfig

Django 1.7+ enables initializing installed applications and autodiscovering modules

On startup, search for and import any modules called *lookups.py* in all installed apps. Your LookupClass subclass may register itself.

**name = 'ajax_select'**

**ready**()

**verbose_name = 'Ajax Selects'**

#### ajax_select.fields module

**class** ajax_select.fields.**AutoCompleteField**(*channel*, **args*, ***kwargs*)
Bases: django.forms.fields.CharField

A CharField that uses an AutoCompleteWidget to lookup matching and stores the result as plain text.

**channel = None**

**class** ajax_select.fields.**AutoCompleteSelectField**(*channel*, **args*, ***kwargs*)
Bases: django.forms.fields.CharField

Form field to select a Model for a ForeignKey db field.

**channel = None**

**check_can_add**(*user*, *model*)

**clean**(*value*)

**has_changed**(*initial_value*, *data_value*)

**class** ajax_select.fields.**AutoCompleteSelectMultipleField**(*channel*, **args*, ***kwargs*)
Bases: django.forms.fields.CharField

form field to select multiple models for a ManyToMany db field

**channel = None**

**check_can_add**(*user*, *model*)

**clean**(*value*)

**has_changed**(*initial_value*, *data_value*)

**class** ajax_select.fields.**AutoCompleteSelectMultipleWidget**(*channel*, *help_text=u''*, *show_help_text=True*, *plugin_options={}*, **args*, ***kwargs*)
Bases: django.forms.widgets.SelectMultiple

Widget to select multiple models for a ManyToMany db field.

**add_link = None**

**id_for_label**(*id_*)

**media**

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

**value_from_datadict**(*data*, *files*, *name*)

**class** ajax_select.fields.**AutoCompleteSelectWidget**(*channel*, *help_text=u''*, *show_help_text=True*, *plugin_options={}*, **args*, ***kwargs*)
Bases: django.forms.widgets.TextInput

Widget to search for a model and return it as text for use in a CharField.

**add_link = None**

**id_for_label**(*id_*)

**media**

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

**value_from_datadict**(*data*, *files*, *name*)

**class** ajax_select.fields.**AutoCompleteWidget**(*channel*, **args*, ***kwargs*)
Bases: django.forms.widgets.TextInput

Widget to select a search result and enter the result as raw text in the text input field. the user may also simply enter text and ignore any auto complete suggestions.

**channel = None**

**help_text = u'‘**

**html_id = u'‘**

**media**

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

ajax_select.fields.**autoselect_fields_check_can_add**(*form*, *model*, *user*)
Check the form's fields for any autoselect fields and enable their widgets with green + button if permissions allow then to create the related_model.

ajax_select.fields.**plugin_options**(*lookup*, *channel_name*, *widget_plugin_options*, *initial*)
Make a JSON dumped dict of all options for the jQuery ui plugin.

### ajax_select.helpers module

ajax_select.helpers.**make_ajax_field**(*related_model*, *fieldname_on_model*, *channel*, *show_help_text=False*, ***kwargs*)
Makes an AutoComplete field for use in a Form.

**Parameters**

- **related_model** (*Model*) – model of the related object
- **fieldname_on_model** (*str*) – field name on the model being edited
- **channel** (*str*) – channel name of a registered LookupChannel
- **show_help_text** (*bool*) – show or supress help text below the widget Django admin will show help text below the widget, but not for ManyToMany inside of admin inlines This setting will show the help text inside the widget itself.
- **kwargs** – optional args
  - **help_text: default is the model db field's help_text.** None will disable all help text
  - label: default is the model db field's verbose name
  - required: default is the model db field's (not) blank

**Returns** field

**Return type** (AutoCompleteField, AutoCompleteSelectField, AutoCompleteSelectMultipleField)

ajax_select.helpers.**make_ajax_form**(*model*, *fieldlist*, *superclass=<class 'django.forms.models.ModelForm'>*, *show_help_text=False*, ***kwargs*)
Creates a ModelForm subclass with AutoComplete fields.

**Parameters**

- **model** (*type*) – Model class for which you are making the ModelForm
- **fieldlist** (*dict*) – {field_name -> channel_name, ...}
- **superclass** (*type*) – optional ModelForm superclass
- **show_help_text** (*bool*) – suppress or show the widget help text

**Returns** a ModelForm suitable for use in an Admin

**Return type** ModelForm

Usage:

```
from django.contrib import admin
from ajax_select import make_ajax_form
from yourapp.models import YourModel

@admin.register(YourModel)
class YourModelAdmin(Admin):

    form = make_ajax_form(YourModel, {
        'contacts': 'contact',  # ManyToManyField
        'author':'contact'      # ForeignKeyField
    })
```

Where 'contacts' is a ManyToManyField specifying to use the lookup channel 'contact' and 'author' is a ForeignKeyField specifying here to also use the same lookup channel 'contact'

### ajax_select.lookup_channel module

**class** ajax_select.lookup_channel.**LookupChannel**
Bases: object

Subclass this, setting the model and implementing methods to taste.

**model**
*Model*

The Django Model that this lookup channel will search for.

**plugin_options**
*dict*

Options passed to jQuery UI plugin that are specific to this channel.

**min_length**
*int*

Minimum number of characters user types before a search is initiated.

This is passed to the jQuery plugin_options. It is used in jQuery's UI when filtering results from its own cache.

It is also used in the django view to prevent expensive database queries. Large datasets can choke if they search too often with small queries. Better to demand at least 2 or 3 characters.

**can_add**(*user*, *other_model*)
Check if the user has permission to add a ForeignKey or M2M model.

This enables the green popup + on the widget. Default implentation is the standard django permission check.

**Parameters**

- **(User)** (*user*) –
- **other_model** (*Model*) – the ForeignKey or M2M model to check if the User can add.

**Returns** bool

**check_auth**(*request*)
By default only request.user.is_staff have access.

This ensures that nobody can get your data by simply knowing the lookup URL.

This is called from the ajax_lookup view.

Public facing forms (outside of the Admin) should implement this to allow non-staff to use this LookupChannel.

**Parameters** **(Request)** (*request*) –

**Raises** PermissionDenied –

**format_item_display**(*obj*)
(HTML) format item for displaying item in the selected deck area.

**Parameters** **obj** (*Model*) –

**Returns** formatted string, may contain HTML.

**Return type** str

**format_match**(*obj*)
(HTML) Format item for displaying in the dropdown.

**Parameters** **obj** (*Model*) –

**Returns** formatted string, may contain HTML.

**Return type** str

**get_objects**(*ids*)
This is used to retrieve the currently selected objects for either ManyToMany or ForeignKey.

Note that the order of the ids supplied for ManyToMany fields is dependent on how the objects manager fetches it. ie. what is returned by *YourModel.{fieldname}_set.all()*

In most situations (especially postgres) this order is indeterminate – not the order that you originally added them in the interface. See Ordered ManyToMany fields for a solution to this.

**Parameters** **ids** (*list*) – list of primary keys

**Returns** list of Model objects

**Return type** list

**get_query**(*q*, *request*)
Return a QuerySet searching for the query string *q*.

Note that you may return any iterable so you can return a list or even use yield and turn this method into a generator.

**Parameters**

- **q** (*str, unicode*) – The query string to search for.
- **request** (*Request*) – This can be used to customize the search by User or to use additional GET variables.

**Returns** iterable of related_models

**Return type** (QuerySet, list, generator)

**get_result**(*obj*)
The text result of autocompleting the entered query.

For a partial string that the user typed in, each matched result is here converted to the fully completed text.

This is currently displayed only for a moment in the text field after the user has selected the item. Then the item is displayed in the item_display deck and the text field is cleared.

**Parameters** **obj** (*Model*) –

**Returns** The object as string

**Return type** str

**min_length = 1**

**model = None**

**plugin_options = {}**

### ajax_select.models module

Blank file so that Django recognizes the app.

This is only required for Django < 1.7

### ajax_select.registry module

**class** ajax_select.registry.**LookupChannelRegistry**
Bases: object

Registry for LookupChannels activated for your django project.

This includes any installed apps that contain lookup.py modules (django 1.7+) and any lookups that are explicitly declared in *settings.AJAX_LOOKUP_CHANNELS*

**get**(*channel*)
Find the LookupChannel class for the named channel and instantiate it.

**Parameters** **channel** (*string*) –

- name that the lookup channel was registered at

**Returns** LookupChannel

**Raises**

- **ImproperlyConfigured - if channel is not found.** –
- **Exception - invalid lookup_spec was stored in registery** –

**is_registered**(*channel*)

**load_channels**()

**make_channel**(*app_model*, *arg_search_field*)
Automatically make a LookupChannel.

**Parameters**

- **app_model** (*str*) – app_name.ModelName
- **arg_search_field** (*str*) – the field to search against and to display in search results

**Returns** LookupChannel

**register**(*lookup_specs*)
Register a set of lookup definitions.

**Parameters** **lookup_specs** (*dict*) – One or more LookupChannel specifications - *{'channel': LookupChannelSubclass} - {'channel': ('module.of.lookups', 'MyLookupClass')} - {'channel': {'model': 'MyModelToBeLookedUp', 'search_field': 'field_to_search'}}*

ajax_select.registry.**can_autodiscover**()

ajax_select.registry.**get_model**(*app_label*, *model_name*)
Loads the model given an 'app_label' 'ModelName'

ajax_select.registry.**register**(*channel*)
Decorator to register a LookupClass.

#### Example

from ajax_select import LookupChannel, register

@register('agent') class AgentLookup(LookupClass):

**def get_query(self):** ...

**def format_item(self):** ...

### ajax_select.urls module

### ajax_select.views module

ajax_select.views.**add_popup**(*request*, *app_label*, *model*)
Presents the admin site popup add view (when you click the green +).

It serves the admin.add_view under a different URL and does some magic fiddling to close the popup window after saving and call back to the opening window.

**make sure that you have added ajax_select.urls to your urls.py::** (r'^ajax_select/', include('ajax_select.urls')),

this URL is expected in the code below, so it won't work under a different path TODO - check if this is still true.

This view then hijacks the result that the django admin returns and instead of calling django's dismissAddAnontherPopup(win,newId,newRepr) it calls didAddPopup(win,newId,newRepr) which was added inline with bootstrap.html

ajax_select.views.**ajax_lookup**(*request*, *channel*)
Load the named lookup channel and lookup matching models.

GET or POST should contain 'term'

**Returns** *[{pk: value: match: repr:}, ...]*

**Return type** HttpResponse - JSON

**Raises** **PermissionDenied - depending on the LookupChannel's implementation of check_auth**
–

### Module contents

JQuery-Ajax Autocomplete fields for Django Forms.

- genindex
- modindex

## a

## A



## C



## F



## G

## H



## I



## L



## M



## N



## P



## R



## V