django-autocomplete-light Documentation

Release 0.7

James Pic

June 05, 2012

CONTENTS

CHAPTER

ONE

FEATURES

This app fills all your ajax autocomplete needs:

- **global navigation** autocomplete like on http://betspire.com
- **autocomplete widget** for *ModelChoiceField and ModelMultipleChoiceField*
- **autocompletes that depend on each other**, *working example* provided
- **GenericForeignKey** fully *supported*
- **django-generic-m2m** support, yes that's a *generic M2M relation*!
- **APIs** powered autocomplete support, proposing *results that are not (yet) in the database*
- **very few code** was required to implement this kind of features , this app lets you satisfy the craziest autocomplete ideas your users might want, in a maintainable and sane way,
- **0 hack** required for *admin integration*, just use a form that uses the widget. It works exactly the same in the admin and in your pages.
- **no jQuery-ui** required, the autocomplete script is as simple as possible,
- **all** the design of the autocompletes is encapsulated in template, *unlimited design possibilities*
- **99%** of the python logic is encapsulated in "channel" classes, *unlimited server side development possibilities*
- **99%** the javascript logic is encapsulated in an object, you can override any attribute or method, *unlimited client side development possibilities*
- **0 inline javascript** you can load the javascript just before </body> for best page loading performance, *wherever you want*
- **simple** python, html and javascript, easy to hack, PEP8 compliant
- **less sucking** code, no funny hacks, clean api, as few code as possible, that also means this is *not for pushovers*

CHAPTER

TWO

README

This is a simple alternative to django-ajax-selects.

CHAPTER

THREE

REQUIREMENTS

- Python 2.7
- jQuery 1.7+
- Django 1.4+ (at least for autocomplete_light.forms helpers)
- django.contrib.staticfiles or you're on your own

CHAPTER

FOUR

RESOURCES

You could subscribe to the mailing list ask questions or just be informed of package updates.

- Mailing list graciously hosted by Google
- Git graciously hosted by GitHub,
- Documentation graciously hosted by RTFD,
- Package graciously hosted by PyPi,
- Continuous integration graciously hosted by Travis-ci

CHAPTER

FIVE

DEMO

See test_project/README

CHAPTER

SIX

FULL DOCUMENTATION

6.1 django-autocomplete-light demo

The test_project lives in the test_project subdirectory of django-autocomplete-light's repository.

6.1.1 Install

We're going to use virtualenv, so that we don't pollute your system when installing dependencies. If you don't already have virtualenv, you can install it either via your package manager, either via python's package manager with something like:

```
sudo easy_install virtualenv
```

Install last release:

```
rm -rf django-autocomplete-light autocomplete_light_env/

virtualenv autocomplete_light_env
source autocomplete_light_env/bin/activate
git clone https://jpic@github.com/yourlabs/django-autocomplete-light.git
cd django-autocomplete-light/test_project
pip install -r requirements.txt
./manage.py runserver
```

Install development versions, if you want to contribute hehehe:

```
AUTOCOMPLETE_LIGHT_VERSION="master"
CITIES_LIGHT_VERSION="master"

rm -rf autocomplete_light_env/

virtualenv autocomplete_light_env
source autocomplete_light_env/bin/activate
pip install -e git+git://github.com/yourlabs/django-cities-light.git@$CITIES_LIGHT_VERSION#egg=cities
pip install -e git+git://github.com/yourlabs/django-autocomplete-light.git@$AUTOCOMPLETE_LIGHT_VERSIO
cd autocomplete_light_env/src/autocomplete-light/test_project
pip install -r requirements.txt
./manage.py runserver
```

Login with user "test" and password "test".

If you want to redo the database, but make sure you read README first:

```
rm db.sqlite
./manage.py syncdb
./manage.py cities_light
```

6.1.2 Try basic features

Once you have the test_project server running (see INSTALL if you don't), open the first contact.

You will see two addresses:

- one at Paris, France
- one at Paris, United States

The reason for that is that there are several cities in the world with the name "Paris". This is the reason why the double autocomplete widget is interresting: it filters the cities based on the selected country.

Note that only cities from France, USA and Belgium are in the demo database.

Note that you can test autocompletes for generic foreign keys in this project too.

6.1.3 Try advanced features

Assuming you installed the test_project, all you need in addition is to install requirements for this project:

```
cd autocomplete_light_env/src/autocomplete-light/test_api_project
pip install -r requirements.txt
```

Then, refer to README.rst in this folder.

This project demonstrates how the autocomplete can suggest results from a remote API - and thus which don't have a pk in the local database.

In one console:

```
cd test_project
./manage.py runserver
```

In another:

```
cd test_api_project
./manage.py runserver 127.0.0.1:8001
```

In http://localhost:8001/admin, you should be able to test:

- compatibility with django-admintools-bootstrap
- generic fk autocomplete
- generic m2m autocomplete
- remote api autocomplete (cities/countries are suggested and imported from test_project)
- autocompletes in inlines, dual widget, etc, etc ...

If you're not going to use localhost:8000 for test_project, then you should update source urls in *test_api_project/test_api_project/autocomplete_light_registry.py*.

Now, note that there are no or few countries in test_api_project database.

Again, test_project's database only includes countries France, Belgium and America so there's no need to try the other one unless you know what you're doing.

Also note that, city and country autocomplete work the same. The reason for that is that test_api_project uses City and Country remote channel to add results to the autocomplete that are not in the local database.

6.2 Quick start

The purpose of this documentation is to get you started as fast as possible, because your time matters and you probably have other things to worry about.

6.2.1 Quick install

Install the package:

```
pip install django-autocomplete-light
# or the development version
pip install -e git+git://github.com/yourlabs/django-autocomplete-light.git#egg=django-autocomplete-li
```

Add to INSTALLED_APPS: 'autocomplete_light'

Add to urls:

```
url(r'autocomplete/', include('autocomplete_light.urls')),
```

Add before admin.autodiscover():

```
import autocomplete_light
autocomplete_light.autodiscover()
```

At this point, we're going to assume that you have django.contrib.staticfiles working. This means that static files are automatically served with runserver, and that you have to run collectstatic when using another server (fastcgi, uwsgi, and whatnot). If you don't use django.contrib.staticfiles, then you're on your own to manage staticfiles. This is an example of how you could load the javascript:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javascript">
{% include 'autocomplete_light/static.html' %}
```

6.2.2 Quick admin integration

For AutocompleteWidget to be enabled in the admin, you should create your own `admin/base_site.html` template as demonstrated in `test_project/templates/admin/base_site.html`:

```
{% extends "admin/base.html" %}
{% load i18n %}

{% block footer %}
    {{ block.super }}

    <script src="{{ STATIC_URL }}jquery.js" type="text/javascript"></script>
    {% include 'autocomplete_light/static.html' %}
    {% comment %}
    Load additionnal script or style dependencies here. For instance, the
    double country/city autocomplete widget requires the countrycity deck
    bootstrap so we'll load it. But you don't need this one if you don't use
    the countrycity widget of the cities_light app.
    {% endcomment %}
```

```
    <script src="{{ STATIC_URL }}cities_light/autocomplete_light.js" type="text/javascript"></script>
{% endblock %}
```

Create `yourapp/autocomplete_light_registry.py`, assuming "Author" has a "full_name" CharField:

```
import autocomplete_light

from models import Author

autocomplete_light.register(Author, search_field='full_name')
```

See more about the channel registry in *Registry*.

But still, the default implementation of query_filter() is pretty trivial, you might want to customize how it will filter the queryset. See more about customizing channels in *Channels basics*.

Anyway, finish by setting `BookAdmin.form` in yourapp/admin.py:

```
from django.contrib import admin

import autocomplete_light

from models import Book

class BookAdmin(admin.ModelAdmin):
    # use an autocomplete for Author
    form = autocomplete_light.modelform_factory(Book)
admin.site.register(Book, BookAdmin)
```

6.2.3 Quick form integration

AutocompleteWidget is usable on ModelChoiceField and ModelMultipleChoiceField.

class `autocomplete_light.widgets.`**`AutocompleteWidget`**(*channel_name*, **args*, ***kwargs*)

Widget suitable for ModelChoiceField and ModelMultipleChoiceField.

Example usage:

```
from django import forms

import autocomplete_light

from models import Author

class AuthorsForm(forms.Form):
    lead_author = forms.ModelChoiceField(Author.objects.all(), widget=
        autocomplete_light.AutocompleteWidget(
            'AuthorChannel', max_items=1))

    contributors = forms.ModelMultipleChoiceField(Author.objects.all(),
        widget=autocomplete_light.AutocompleteWidget('AuthorChannel'))
```

AutocompleteWidget constructor decorates SelectMultiple constructor

Arguments: channel_name – the name of the channel that this widget should use.

Keyword arguments are passed to javascript via data attributes of the autocomplete wrapper element:

max_items The number of items that this autocomplete allows. If set to 0, then it allows any number of selected items like a multiple select, well suited for ManyToMany relations or any kind of ModelMultipleChoiceField. If set to 3 for example, then it will only allow 3 selected items. It should be set to 1 if the widget is for a ModelChoiceField or ForeignKey, in that case it would be like a normal select. Default is 0.

min_characters The minimum number of characters before the autocomplete box shows up. If set to 2 for example, then the autocomplete box will show up when the input receives the second character, for example 'ae'. If set to 0, then the autocomplete box will show up as soon as the input is focused, even if it's empty, behaving like a normal select. Default is 0.

bootstrap The name of the bootstrap kind. By default, deck.js will only initialize decks for wrappers that have data-bootstrap="normal". If you want to implement your own bootstrapping logic in javascript, then you set bootstrap to anything that is not "normal". By default, its value is copied from channel.bootstrap.

placeholder The initial value of the autocomplete input field. It can be something like 'type your search here'. By default, it is copied from channel.placeholder.

payload A dict of data that will be exported to JSON, and parsed into the Deck instance in javascript. It allows to pass variables from Python to Javascript.

6.3 Making a global navigation autocomplete

This guide demonstrates how to make a global navigation autocomplete like on http://betspire.com.

6.3.1 Create the view

The global navigation autocomplete is generated by a normal view, with a normal template.

Then, you can just test it by openning /your/autocomplete/url/?q=someString

Only two things matter:

- you should be able to define a selector for your options. For example, your autocomplete template could contain a list of divs with class "option", and your selector would be '.option'.
- each option should contain an url of course, to redirect the user when he selects a option

Actually, it's not totally true, you could do however you want, but that's a simple way i've found.

Once this works, you can follow to the next step. For your inspiration, you may also read the following example.

Example

Personnaly, I like to have an app called 'project_specific' where I can put my project-specific, non-reusable, code. So in project_specific/autocomplete.py of a project I have this:

```
from django import shortcuts
from django.db.models import Q

from art.models import Artist, Artwork

def autocomplete(request,
    template_name='project_specific/autocomplete.html', extra_context=None):
    q = request.GET['q'] # crash if q is not in the url
    context = {
        'q': q,
    }
```

```
    queries = {}
    queries['artworks'] = Artwork.objects.filter(
        name__icontains=q).distinct()[:3]
    queries['artists'] = Artist.objects.filter(
        Q(first_name__icontains=q)|Q(last_name__icontains=q)|Q(name__icontains=q)
        ).distinct()[:3]
    # more ...

    # install queries into the context
    context.update(queries)

    # mix options
    options = 0
    for query in queries.values():
        options += len(query)
    context['options'] = options

    return shortcuts.render(request, template_name, context)
```

And in project_specific/autocomplete.html:

```
{% load i18n %}
{% load thumbnail %}
{% load url from future %}
{% load humanize %}

<ul>
{% if artworks %}
    <li><em>{% trans 'Artworks' %}</em></li>
    {% for artwork in artworks %}
        <li class="artwork">
            <a href="{{ artwork.get_absolute_url }}">
                {% if artwork.first_image %}
                    <img src="{% thumbnail artwork.first_image 16x16 %}" style="vertical-align: middl
                {% endif %}
                {{ artwork }}
            </a>
        </li>
    {% endfor %}
{% endif %}
{% if artists %}
    <li><em>{% trans 'Artists' %}</em></li>
    {% for artist in artists %}
        <li class="artist">
            <a href="{{ artist.get_absolute_url }}">
                {% if artist.image %}
                    <img src="{% thumbnail artist.image 16x16 %}" style="vertical-align: middle" />
                {% endif %}

                {{ artist }}
            </a>
        </li>
    {% endfor %}
{% endif %}
{# more ...}

{% if not options %}
    <li><em>{% trans 'No options' %}</em></li>
```

```
    <li><a href="{% url 'haystack_search' %}?q={{ q|urlencode }}">{% blocktrans %}Search for {{ q }}
{% endif %}

</ul>
```

In this template, my option selector is simply 'li:has(a)'. So every <a> tag that is in an li with an a tag will be considered as a valid option by the autocomplete.

As for the url, it looks like this:

```
url(
    r'^autocomplete/$',
    views.autocomplete,
    name='project_specific_autocomplete',
),
```

So, nothing really special here ... and that's what I like with this autocomplete. You can use the presentation you want as long as you have a selector for your options.

6.3.2 Create the input

Nothing magical here, just add an HTML input to your base template, for example:

```
<input type="text" name="q" id="main_autocomplete" />
```

Of course, if you have haystack or any kind of search, you could use it as well, it doesn't matter:

```
<form action="{% url haystack_search %}" method="get">
    {{ search_form.q }}
</form>
```

6.3.3 Loading the script

If you haven't done it already, load jQuery and the yourlabs_autocomplete extension, for example:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javascript">
<script src="{{ STATIC_URL }}autocomplete_light/autocomplete.js" type="text/javascript"></script>
```

6.3.4 Script usage

The last thing we need to do is to connect the autocomplete script with the input and the autocomplete view. Something like this would work:

```
<script type="text/javascript">
$(document).ready(function() {
    $('input#main_autocomplete').yourlabs_autocomplete({
        url: '{% url project_specific_autocomplete %}',
        zindex: 99999,
        id: 'main_autocomplete',
        iterablesSelector: 'li:has(a)',
        defaultValue: "{% trans 'Search : an artwork, an artist, a user, a contact...' %}",
    });
});
</script>
```

There are other options. If these don't work very well for you, you should read autocomplete.js. It's not a fat bloated script like jQueryUi autocomplete with tons of dependencies, so it shouldn't be that hard to figure it out.

The other thing you want to do, is bind an event to the event yourlabs_autocomplete.selectOption, that is fired when the user selects an option by clicking on it for example:

```
<script type="text/javascript">
$(document).ready(function() {
    $('#search_bloc input[name=q]').bind('yourlabs_autocomplete.selectOption', function(e, option) {
        var autocomplete = $(this).yourlabs_autocomplete();

        // hide the autocomplete
        autocomplete.hide();

        // change the input's value to 'loading page: some page'
        autocomplete.el.val('{% trans 'loading page' %}: ' + $.trim(option.text()));

        // find the url of the option
        link = $(option).find('a:first');

        // if the link looks good
        if (link.length && link.attr('href') != undefined) {
            // open the link
            window.location.href = link.attr('href');
            return false;
        } else {
            // that should only happen during development !!
            alert('sorry, i dunno what to do with your selection!!');
        }
    });
});
</script>
```

That's all folks ! Enjoy your fine global navigation autocomplete. Personnaly I think there should be one in the header of every project, it is just **so** convenient for the user. And if nicely designed, it is very 'web 2.0' whatever it means hahah.

6.4 Integration with forms

The purpose of this documentation is to describe every element in a chronological manner. Because you want to know everything about this app and hack like crazy.

It is complementary with the quick documentation.

6.4.1 Django startup

Registry

The registry module provides tools to maintain a registry of channels.

The first thing that should happen when django starts is registration of channels. It should happen first, because channels are required for autocomplete widgets. And autocomplete widgets are required for forms. And forms are required for ModelAdmin.

It looks like this:

- in `yourapp/autocomplete_light_registry.py`, register your channels with `autocomplete_light.register()`,
- in `urls.py`, do `autocomplete_light.autodiscover()` **before** `admin.autodiscover()`.

ChannelRegistry Subclass of Python's dict type with registration/unregistration methods.

registry Instance of ChannelRegistry.

register Proxy registry.register.

autodiscover Find channels and fill registry.

class `autocomplete_light.registry.`**`ChannelRegistry`**
Dict with some shortcuts to handle a registry of channels.

`channel_for_model`(*model*)
Return the channel class for a given model.

`register`(**args*, ***kwargs*)
Proxy registry.register_model_channel() or registry.register_channel() if there is no apparent model for the channel.

Example usages:

```
# Will create and register SomeModelChannel, if SomeChannel.model
# is None (which is the case by default):
autocomplete_light.register(SomeModel)

# Same but using SomeChannel as base:
autocomplete_light.register(SomeModel, SomeChannel)

# Register a channel without model, ensure that SomeChannel.model
# is None (which is the default):
autocomplete_light.register(SomeChannel)

# As of 0.5, you may also pass attributes*, ie.:
autocomplete_light.register(SomeModel, search_field='search_names',
    result_template='somemodel_result.html')
```

You may pass attributes via kwargs, only if the registry creates a type:

- if no channel class is passed,
- or if the channel class has no model attribute,
- and if the channel classs is not generic

`register_channel`(*channel*)
Register a channel without model, like a generic channel.

`register_model_channel`(*model*, *channel=None*, *channel_name=None*, ***kwargs*)
Add a model to the registry, optionnaly with a given channel class.

model The model class to register.

channel The channel class to register the model with, default to ChannelBase.

channel_name Register channel under channel_name, default is ModelNameChannel.

kwargs Extra attributes to set to the channel class, if created by this method.

Three cases are possible:

- specify model class and ModelNameChannel will be generated extending ChannelBase, with attribute model=model

- specify a model and a channel class that does not have a model attribute, and a ModelNameChannel will be generated, with attribute model=model
- specify a channel class with a model attribute, and the channel is directly registered

To keep things simple, the name of a channel is it's class name, which is usually generated. In case of conflicts, you may override the default channel name with the channel_name keyword argument.

unregister(*name*)
Unregister a channel.

autocomplete_light.registry.**register**(**args*, ***kwargs*)
Proxy registry.register

autocomplete_light.registry.**autodiscover**()
Check all apps in INSTALLED_APPS for stuff related to autocomplete_light.

For each app, autodiscover imports app.autocomplete_light_registry if available, resulting in execution of register() statements in that module, filling registry.

Consider a standard app called 'cities_light' with such a structure:

```
cities_light/
    __init__.py
    models.py
    urls.py
    views.py
    autocomplete_light_registry.py
```

With such a autocomplete_light_registry.py:

```
from models import City, Country
import autocomplete_light
autocomplete_light.register(City)
autocomplete_light.register(Country)
```

When autodiscover() imports cities_light.autocomplete_light_registry, both CityChannel and CountryChannel will be registered. For details on how these channel classes are generated, read the documentation of ChannelRegistry.register.

Channels basics

Example

django-cities-light ships the working example.

API

The channel.base module provides a channel class which you can extend to make your own channel. It also serves as default channel class.

class autocomplete_light.channel.base.**ChannelBase**
A basic implementation of a channel, which should fit most use cases.

Attributes:

model The model class this channel serves. If None, a new class will be created in registry.register, and the model attribute will be set in that subclass. So you probably don't need to worry about it, just know that it's there for you to use.

result_template The template to use in result_as_html method, to render a single autocomplete suggestion. By default, it is autocomplete_light/channelname/result.html or autocomplete_light/result.html.

autocomplete_template The template to use in render_autocomplete method, to render the autocomplete box. By default, it is autocomplete_light/channelname/autocomplete.html or autocomplete_light/autocomplete.html.

search_field The name of the field that the default implementation of query_filter uses. Default is 'name'.

limit_results The number of results that this channel should return. For example, if query_filter returns 50 results and that limit_results is 20, then the first 20 of 50 results will be rendered. Default is 20.

bootstrap The name of the bootstrap kind. By default, deck.js will only initialize decks for wrappers that have data-bootstrap="normal". If you want to implement your own bootstrapping logic in javascript, then you set bootstrap to anything that is not "normal". Default is 'normal'.

placeholder The initial text in the autocomplete text input.

Set result_template and autocomplete_template if necessary.

are_valid(*values*)
Return True if the values are valid.

By default, expect values to be a list of object ids, return True if all the ids are found in the queryset.

as_dict()
Return a dict of variables for this channel, it is used by javascript.

get_absolute_url()
Return the absolute url for this channel, using autocomplete_light_channel url

get_queryset()
Return a queryset for the channel model.

get_results(*values=None*)
Return an iterable of result to display in the autocomplete box.

By default, it will:

- call self.get_queryset(),
- call values_filter() if values is not None,
- call query_filter() if self.request is set,
- call order_results(),
- return a slice from offset 0 to self.limit_results.

init_for_request(*request*, **args*, ***kwargs*)
Set self.request, self.args and self.kwargs, useful in query_filter.

order_results(*results*)
Return the result list after ordering.

By default, it expects results to be a queryset and order it by search_field.

query_filter(*results*)
Filter results using the request.

By default this will expect results to be a queryset, and will filter it with self.search_field + '__icontains'=self.request['q'].

render_autocomplete()
Render the autocomplete suggestion box.

By default, render self.autocomplete_template with the channel in the context.

result_as_html (*result*, *extra_context=None*)
Return the html representation of a result for display in the deck and autocomplete box.

By default, render result_template with channel and result in the context.

result_as_value (*result*)
Return the value that should be set to the widget field for a result.

By default, return result.pk.

values_filter (*results*, *values*)
Filter results based on a list of values.

By default this will expect values to be an iterable of model ids, and results to be a queryset. Thus, it will return a queryset where pks are in values.

Forms

Example

A simple example from test_project:

```
from django import forms

import autocomplete_light
from cities_light.models import City
from cities_light.contrib.autocomplete_light_widgets import \
    CityAutocompleteWidget

from models import Address
from generic_form_example import TaggedItemForm


class AddressForm(forms.ModelForm):
    city = forms.ModelChoiceField(City.objects.all(),
        widget=CityAutocompleteWidget('CityChannel', max_items=1))

    class Meta:
        model = Address
        widgets = autocomplete_light.get_widgets_dict(Address,
            autocomplete_exclude='city')
```

API

A couple of helper functions to help enabling AutocompleteWidget in ModelForms.

autocomplete_light.forms.**get_widgets_dict** (*model*, *autocomplete_exclude=None*)
Return a dict of field_name: widget_instance for model that is compatible with Django.

autocomplete_exclude the list of model field names to ignore

Inspect the model's field and many to many fields, calls registry.channel_for_model to get the channel for the related model. If a channel is returned, then an AutocompleteWidget will be spawned using this channel.

The dict is usable by ModelForm.Meta.widgets. In django 1.4, with modelform_factory too.

autocomplete_light.forms.**modelform_factory**(*model*, *autocomplete_exclude=None*, ***kwargs*)

Wraps around Django's django_modelform_factory, using get_widgets_dict.

Basically, it will use the dict returned by get_widgets_dict in order and pass it to django's modelform_factory, and return the resulting modelform.

Page rendering

It is important to load jQuery first, and then autocomplete_light and application specific javascript, it can look like this:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javascript">
{% include 'autocomplete_light/static.html' %}
```

However, `autocomplete_light/static.html` also includes "remote.js" which is required only by remote channels. If you don't need it, you could either load the static dependencies directly in your template, or override `autocomplete_light/static.html`:

```
<script type="text/javascript" src="{{ STATIC_URL }}autocomplete_light/autocomplete.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}autocomplete_light/deck.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}autocomplete_light/remote.js"></script>
<link rel="stylesheet" type="text/css" href="{{ STATIC_URL }}autocomplete_light/style.css" />
```

Or, if you only want to make a global navigation autocomplete, you only need:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javascript">
<script src="{{ STATIC_URL }}autocomplete_light/autocomplete.js" type="text/javascript"></script>
```

For AutocompleteWidget to be enabled in the admin, you should create your own `admin/base_site.html` template as demonstrated in `test_project/templates/admin/base_site.html`:

```
{% extends "admin/base.html" %}
{% load i18n %}

{% block footer %}
    {{ block.super }}

    <script src="{{ STATIC_URL }}jquery.js" type="text/javascript"></script>
    {% include 'autocomplete_light/static.html' %}
    {% comment %}
    Load additionnal script or style dependencies here. For instance, the
    double country/city autocomplete widget requires the countrycity deck
    bootstrap so we'll load it. But you don't need this one if you don't use
    the countrycity widget of the cities_light app.
    {% endcomment %}
    <script src="{{ STATIC_URL }}cities_light/autocomplete_light.js" type="text/javascript"></script>
{% endblock %}
```

6.4.2 Widget in action

Widget definition

The first thing that happens is the definition of an AutocompleteWidget in a form.

class autocomplete_light.widgets.**AutocompleteWidget**(*channel_name*, **args*, ***kwargs*)

Widget suitable for ModelChoiceField and ModelMultipleChoiceField.

Example usage:

```
from django import forms

import autocomplete_light

from models import Author

class AuthorsForm(forms.Form):
    lead_author = forms.ModelChoiceField(Author.objects.all(), widget=
        autocomplete_light.AutocompleteWidget(
            'AuthorChannel', max_items=1))

    contributors = forms.ModelMultipleChoiceField(Author.objects.all(),
        widget=autocomplete_light.AutocompleteWidget('AuthorChannel'))
```

AutocompleteWidget constructor decorates SelectMultiple constructor

Arguments: channel_name – the name of the channel that this widget should use.

Keyword arguments are passed to javascript via data attributes of the autocomplete wrapper element:

max_items The number of items that this autocomplete allows. If set to 0, then it allows any number of selected items like a multiple select, well suited for ManyToMany relations or any kind of ModelMultipleChoiceField. If set to 3 for example, then it will only allow 3 selected items. It should be set to 1 if the widget is for a ModelChoiceField or ForeignKey, in that case it would be like a normal select. Default is 0.

min_characters The minimum number of characters before the autocomplete box shows up. If set to 2 for example, then the autocomplete box will show up when the input receives the second character, for example 'ae'. If set to 0, then the autocomplete box will show up as soon as the input is focused, even if it's empty, behaving like a normal select. Default is 0.

bootstrap The name of the bootstrap kind. By default, deck.js will only initialize decks for wrappers that have data-bootstrap="normal". If you want to implement your own bootstrapping logic in javascript, then you set bootstrap to anything that is not "normal". By default, its value is copied from channel.bootstrap.

placeholder The initial value of the autocomplete input field. It can be something like 'type your search here'. By default, it is copied from channel.placeholder.

payload A dict of data that will be exported to JSON, and parsed into the Deck instance in javascript. It allows to pass variables from Python to Javascript.

render(*name*, *value*, *attrs=None*)
Render the autocomplete widget.

It will try two templates, like django admin: - autocomplete_light/channelname/widget.html - autocomplete_light/widget.html

Note that it will not pass 'value' to the template, because 'value' might be a list of model ids in the case of ModelMultipleChoiceField, or a model id in the case of ModelChoiceField. To keep things simple, it will just pass a list, 'values', to the template context.

value_from_datadict(*data*, *files*, *name*)
Route to Select if max_items is 1, else route to SelectMultiple

Widget rendering

This is what the default widget template looks like:

```
{% load i18n %}
{% load autocomplete_light_tags %}

{% comment %}
The outer element is called the 'widget wrapper'. It contains some data
attributes to communicate between Python and JavaScript. And of course, it
wraps around everything the widget needs.
{% endcomment %}
<span class="autocomplete_light_widget {{ name }}" id="{{ widget.html_id }}_wrapper" data-bootstrap="

    {# a deck that should contain the list of selected options #}
    <ul id="{{ html_id }}_deck" class="deck" >
        {% for result in results %}
            {{ result|autocomplete_light_result_as_html:channel }}
        {% endfor %}
    </ul>

    {# a text input, that is the 'autocomplete input' #}
    <input type="text" class="autocomplete" name="{{ name }}_autocomplete" id="{{ widget.html_id }}_t

    {# a hidden select, that contains the actual selected values #}
    <select style="display:none" class="valueSelect" name="{{ name }}" id="{{ widget.html_id }}" {% i
        {% for value in values %}
            <option value="{{ value }}" selected="selected">{{ value }}</option>
        {% endfor %}
    </select>

    {# a hidden textarea that contains some json about the widget #}
    <textarea class="json_payload" style="display:none">
        {{ json_payload }}
    </textarea>

    {# a hidden div that serves as template for the 'remove from deck' button #}
    <div style="display:none" class="remove">
        {# This will be appended to results on the deck, it's the remove button #}
        X
    </div>

    <ul style="display:none" class="add_template">
        {% comment %}
        the contained element will be used to render options that are added to the select
        via javascript, for example in django admin with the + sign

        The text of the option will be inserted in the html of this tag
        {% endcomment %}
        <li class="result">
        </li>
    </ul>
</span>
```

6.4.3 Javascript initialization

deck.js initializes all widgets that have bootstrap='normal' (the default), as you can see:

```
$('.autocomplete_light_widget[data-bootstrap=normal]').each(function() {
    $(this).yourlabs_deck();
});
```

If you want to initialize the deck yourself, set the widget or channel bootstrap to something else, say 'yourinit'. Then, add to *yourapp/static/yourapp/autocomplete_light.js* something like:

```
$('.autocomplete_light_widget[data-bootstrap=yourinit]').each(function() {
    $(this).yourlabs_deck({
        getValue: function(result) {
            // your own logic to get the value from an html result
            return ...;
        }
    });
});
```

yourapp/static/yourapp/autocomplete_light.js will be automatically collected by by autodiscover, and the script tag generated by *{% autocomplete_light_static %}*.

In django-cities-light source, you can see a more interresting example where two autocompletes depend on each other.

You should take a look at the code of autocomplete.js and deck.js, as it lets you override everything.

One interresting note is that the plugins (yourlabs_autocomplete and yourlabs_deck) hold a registry. Which means that:

- calling someElement.yourlabs_deck() will instanciate a deck with the passed overrides
- calling someElement.yourlabs_deck() again will return the deck instance for someElement

Javascript cron

deck.js includes a javascript function that is executed every two seconds. It checks each widget's hidden select for a value that is not in the deck, and adds it to the deck if any.

This is useful for example, when an item was added to the hidden select via the '+' button in django admin. But if you create items yourself in javascript and add them to the select it would work too.

Javascript events

When the autocomplete input is focused, autocomplete.js checks if there are enought caracters in the input to display an autocomplete box. If minCharacters is 0, then it would open even if the input is empty, like a normal select box.

If the autocomplete box is empty, it will fetch the channel view. The channel view will delegate the rendering of the autocomplete box to the actual channel. So that you can override anything you want directly in the channel.

class `autocomplete_light.views.`**`ChannelView`**(***kwargs*)

Simple view that routes the request to the appropriate channel.

Constructor. Called in the URLconf; can contain helpful extra keyword arguments, and other things.

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

Return an HttpResponse with the return value of channel.render_autocomplete().

This view is called by the autocomplete script, it is expected to return the rendered autocomplete box contents.

To do so, it gets the channel class from the registry, given the url keyword argument channel, that should be the channel name.

Then, it instanciates the channel with no argument as usual, and calls channel.init_for_request, passing all arguments it recieved.

Finnaly, it makes an HttpResponse with the result of channel.render_autocomplete(). The javascript will use that to fill the autocomplete suggestion box.

post(*request*, **args*, ***kwargs*)
Just proxy channel.post().

This is the key to communication between the channel and the widget in javascript. You can use it to create results and such.

ChannelBase.**render_autocomplete**()
Render the autocomplete suggestion box.

By default, render self.autocomplete_template with the channel in the context.

ChannelBase.**result_as_html**(*result*, *extra_context=None*)
Return the html representation of a result for display in the deck and autocomplete box.

By default, render result_template with channel and result in the context.

Then, autocomplete.js recognizes options with a selector. By default, it is '.result'. This means that any element with the '.result' class in the autocomplete box is considered as an option.

When an option is selected, deck.js calls it's method getValue() and adds this value to the hidden select. Also, it will copy the result html to the deck.

When an option is removed from the deck, deck.js also removes it from the hidden select. This is the default HTML template for the autocomplete:

```
{% load autocomplete_light_tags %}

<ul>
{% for result in channel.get_results %}
    {{ result|autocomplete_light_result_as_html:channel }}
{% endfor %}
</ul>
```

This is the default HTML template for results:

```
<li class="result" data-value="{{ value|safe }}">
    {{ result }} {{ extra_html|safe }}
</li>
```

6.5 GenericForeignKey support

Generic foreign keys are supported since 0.4.

6.5.1 GenericChannelBase

Example

```
import autocomplete_light

from models import Contact, Address

class MyGenericChannel(autocomplete_light.GenericChannelBase):
    def get_querysets(self):
        return {
            Contact: Contact.objects.all(),
            Address: Address.objects.all(),
        }
```

```
    def order_results(self, results):
        if results.model == Address:
            return results.order_by('street')
        elif results.model == Contact:
            return results.order_by('name')

    def query_filter(self, results):
        q = self.request.GET.get('q', None)

        if q:
            if results.model == Address:
                results = results.filter(street__icontains=q)
            elif results.model == Contact:
                results = results.filter(name__icontains=q)

        return results

autocomplete_light.register(MyGenericChannel)
```

API

class `autocomplete_light.channel.generic.`**`GenericChannelBase`**

Wraps around multiple querysets, from multiple model classes, rather than just one.

This is also interresting as it overrides **all** the default model logic from ChannelBase. Hell, you could even copy it and make your CSVChannelBase, a channel that uses a CSV file as backend. But only if you're really bored or for a milion dollars.

Set result_template and autocomplete_template if necessary.

`are_valid`(*values*)

Return True if it can find all the models refered by values.

`get_results`(*values=None*)

Return results for each queryset returned by get_querysets().

Note that it limits each queryset's to self.limit_result. If you want a maximum of 12 suggestions and have a total of 4 querysets, then self.limit_results should be set to 3.

`order_results`(*results*)

Return results, **without** doing any ordering.

In most cases, you would not have to override this method as querysets should be ordered by default, based on model.Meta.ordering.

`result_as_value`(*result*)

Rely on GenericForeignKeyField to return a string containing the content type id and object id of the result.

Because this channel is made for that field, and to avoid code duplication.

`values_filter`(*results*, *values*)

Filter out any result from results that is not refered to by values.

6.5.2 GenericForeignKeyField

Example

```
import autocomplete_light

from models import TaggedItem


class TaggedItemForm(autocomplete_light.GenericModelForm):
    content_object = autocomplete_light.GenericForeignKeyField(
        widget=autocomplete_light.AutocompleteWidget(
            'MyGenericChannel', max_items=1))

    class Meta:
        model = TaggedItem
        widgets = autocomplete_light.get_widgets_dict(TaggedItem)
        exclude = (
            'content_type',
            'object_id',
        )
```

API

class autocomplete_light.generic.**GenericModelForm**(**args*, ***kwargs*)

This simple subclass of ModelForm fixes a couple of issues with django's ModelForm.

- treat virtual fields like GenericForeignKey as normal fields, Django should already do that but it doesn't,
- when setting a GenericForeignKey value, also set the object id and content type id fields, again Django could probably afford to do that.

What ModelForm does, but also add virtual field values to self.initial.

save(*commit=True*)

What ModelForm does, but also set GFK.ct_field and GFK.fk_field if such a virtual field has a value.

This should probably be done in the GFK field itself, but it's here for convenience until Django fixes that.

class autocomplete_light.generic.**GenericForeignKeyField**(*required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=None*, *error_messages=None*, *show_hidden_initial=False*, *validators=[]*, *localize=False*)

Simple form field that converts strings to models.

prepare_value(*value*)

Given a model instance as value, with content type id of 3 and pk of 5, return such a string '3-5'.

to_python(*value*)

Given a string like '3-5', return the model of content type id 3 and pk 5.

6.5.3 GenericManyToMany

Example

Example model with `related`:

```
from django.db import models
from django.db.models import signals
from django.contrib.contenttypes import generic

from genericm2m.models import RelatedObjectsDescriptor

class ModelGroup(models.Model):
    name = models.CharField(max_length=100)

    related = RelatedObjectsDescriptor()

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

Example `generic_m2m.GenericModelForm` usage:

```
import autocomplete_light
from autocomplete_light.contrib.generic_m2m import GenericModelForm, \
    GenericManyToMany

from models import ModelGroup


class ModelGroupForm(GenericModelForm):
    related = GenericManyToMany(
        widget=autocomplete_light.AutocompleteWidget('MyGenericChannel'))

    class Meta:
        model = ModelGroup
```

Example `ModelAdmin`:

```
from django.contrib import admin

from models import ModelGroup
from forms import ModelGroupForm


class ModelGroupAdmin(admin.ModelAdmin):
    form = ModelGroupForm
admin.site.register(ModelGroup, ModelGroupAdmin)
```

API

`autocomplete_light.contrib.generic_m2m` couples django-autocomplete-light with django-generic-m2m.

Generic many to many are supported since 0.5. It depends on django-generic-m2m external apps. Follow django-generic-m2m installation documentation, but at the time of writing it barely consists of adding the `genericm2m` to `INSTALLED_APPS`, and adding a field to models that should have a generic m2m relation. So, kudos to the maintainers of django-generic-m2m, fantastic app, use it for generic many to many relations.

See examples in `test_project/generic_m2m_example`.

class `autocomplete_light.contrib.generic_m2m.`**GenericManyToMany**(*required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=None*, *error_messages=None*, *show_hidden_initial=False*, *validators=[]*, *localize=False*)

Simple form field that converts strings to models.

class `autocomplete_light.contrib.generic_m2m.`**GenericModelForm**(**args*, ***kwargs*)

Extension of autocomplete_light.GenericModelForm, that handles genericm2m's RelatedObjectsDescriptor.

Add related objects to initial for each generic m2m field.

generic_m2m_fields()

Yield name, field for each RelatedObjectsDescriptor of the model of this ModelForm.

save(*commit=True*)

Sorry guys, but we have to force commit=True and call save_m2m() right after.

The reason for that is that Django 1.4 kind of left over cases where we wanted to override save_m2m: it enforces its own, which does not care of generic_m2m of course.

save_m2m()

Save selected generic m2m relations.

6.6 Proposing results from a remote API

This documentation is optionnal, but it is complementary with all other documentation. It aims advanced users.

Consider a social network about music. In order to propose all songs in the world in its autocomplete, it should either:

- have a database with all songs of the world,
- use a simple REST API to query a database with all songs world

The purpose of this documentation is to describe every elements involved. Note that a living demonstration is available in *test_api_project*, where one project serves a full database of cities via an API to another.

6.6.1 Example

In test_api_project, of course you should not hardcode urls like that in actual projects:

```
import autocomplete_light

from cities_light.contrib.autocomplete_light_restframework import RemoteCountryChannel, RemoteCityCha
from cities_light.models import City, Country

autocomplete_light.register(Country, RemoteCountryChannel,
    source_url = 'http://localhost:8000/cities_light/country/')
autocomplete_light.register(City, RemoteCityChannel,
    source_url = 'http://localhost:8000/cities_light/city/')
```

Check out the documentation of *RemoteCountryChannel and RemoteCityChannel* for more.

6.6.2 API

class autocomplete_light.channel.remote.**RemoteChannelBase**

Uses an API to propose suggestions from an HTTP API, tested with djangorestframework.

model_for_source_url A **very important function to override!** take an API URL and return the corresponding model class. This is API specific, there is a complete example in cities_light.contrib.

source_url The full URL to the list API. For example, to a djangorestframework list view.

An example implementation usage is demonstrated in the django-cities-light contrib folder.

Autocomplete box display chronology:

- autocomplete.js requests autocomplete box to display for an input,
- get_results() fetches some extra results via get_remote_results(),
- get_remote_results() calls source_url and returns a list of models,
- the remote results are rendered after the local results in widget.html. It includes some JSON in a hidden textarea, like the API's url for each result.

Remote result selection chronology:

- deck.js calls remoteGetValue() instead of the default getValue(),
- remoteGetValue() posts the json from the result to ChannelView,
- ChannelView.post() does its job of proxying RemoteChannelBase.post(),
- RemoteChannelBase.post() returns an http response which body is just the pk of the result in the local database, using self.fetch_result(),
- self.fetch_result() passes the API url of the result and recursively saves the remote models into the local database, returning the id of the newly created object.

Set result_template and autocomplete_template if necessary.

fetch(*url*)

Given an url to a remote object, return the corresponding model from the local database.

The default implementation expects url to respond with a JSON dict of the attributes of an object.

For relation attributes, it expect the value to be another url that will respond with a JSON dict of the attributes of the related object.

It calls model_for_source_url() to find which model class corresponds to which url. This allows fetch() to be recursive.

fetch_result(*result*)

Take a result's dict representation, return it's local pk which might have been just created.

If your channel works with 0 to 1 API call, consider overriding this method. If your channel is susceptible of using several different API calls, consider overriding fetch().

get_remote_results(*max*)

Parses JSON from the API, return model instances.

The JSON should contain a list of dicts. Each dict should contain the attributes of an object. Relation attributes should be represented by their url in the API, which is set to model._source_url.

get_results(*values=None*)

Returns a list of results from both the local database and the API if in the context of a request.

Using self.limit_results and the number of local results, adds results from get_remote_results().

get_source_url(*limit*)
Return an API url for the current autocomplete request.

By default, return self.source_url with the data dict returned by get_source_url_data().

get_source_url_data(*limit*)
Given a limit of items, return a dict of data to send to the API.

By default, it passes current request GET arguments, along with format: 'json' and the limit.

model_for_source_url(*url*)
Take an URL from the API this remote channel is supposed to work with, return the model class to use for that url.

It is only needed for the default implementation of fetch(), because it has to follow relations recursively.

post(*request*, **args*, ***kwargs*)
Take POST variable 'result', install it in the local database, return the newly created id.

The HTTP response has status code 201 Created.

result_as_dict(*result*)
Return the result pk or _source_url.

result_as_value(*result*)
Return the result pk or source url.

6.6.3 Javascript fun

Channels with *bootstrap='remote'* get a deck using *RemoteChannelDeck's getValue()* rather than the default *getValue()* function.

```
var RemoteChannelDeck = {
    // The default deck getValue() implementation just returns the PK from the
    // result HTML. RemoteChannelDeck's implementation checks for a textarea
    // that would contain a JSON dict in the result's HTML. If the dict has a
    // 'value' key, then return this value. Otherwise, make a blocking ajax
    // request: POST the json dict to the channel url. It expects that the
    // response will contain the value.
    getValue: function(result) {
        data = $.parseJSON(result.find('textarea').html());

        if (data.value) return data.value;

        var value = false;
        $.ajax(this.payload.channel.url, {
            async: false,
            type: 'post',
            data: {
                'result': result.find('textarea').html(),
            },
            success: function(text, jqXHR, textStatus) {
                value = text;
            }
        });

        return value;
    }
}
```

```
$(document).ready(function() {
    // Instanciate decks with RemoteChannelDeck as override for all widgets with
    // channel 'remote'.
    $('.autocomplete_light_widget[data-bootstrap=remote]').each(function() {
        $(this).yourlabs_deck(RemoteChannelDeck);
    });
});
```

6.7 Django 1.3 support workarounds

The app is was developed for Django 1.4. However, there are workarounds to get it to work with Django 1.3 too. This document attemps to provide an exhaustive list of notes that should be taken in account when using the app with django-autocomplete-light.

6.7.1 modelform_factory

The provided autocomplete_light.modelform_factory relies on Django 1.4's modelform_factory that accepts a 'widgets' dict.

Django 1.3 does not allow such an argument. You may however define your form as such:

```
class AuthorForm(forms.ModelForm):
    class Meta:
        model = Author
        widgets = autocomplete_light.get_widgets_dict(Author)
```

CHAPTER

SEVEN

JAVASCRIPT API

Work in progress:

- autocomplete.js
- deck.js

CHAPTER

EIGHT

WHEN THINGS GO WRONG

If you don't know how to debug, you should learn to use:

Firebug javascript debugger Open the script tab, select a script, click on the left of the code to place a breakpoint

Ipdb python debugger Install ipdb with pip, and place in your python code: import ipdb; ipdb.set_trace()

If you are able to do that, then you are a professional, enjoy autocomplete_light !!!

If you need help, open an issue on the github issues page.

But make sure you've read how to report bugs effectively and how to ask smart questions.

Also, don't hesitate to do pull requests !

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

a