django-autocomplete-light Documentation

Release 1.0.26

James Pic

March 05, 2016

Contents

CHAPTER 1

Features

This app fills all your ajax autocomplete needs:

- **global navigation** autocomplete *like on facebook*
- **autocomplete widget** for *ModelChoiceField and ModelMultipleChoiceField*
- **GenericForeignKey** fully *supported*
- **django-generic-m2m** support, yes that's a *generic M2M relation*!
- **CharField autocomplete** support, *comma separated values*, useful for tags field,
- **APIs** powered autocomplete support, proposing *results that are not (yet) in the database*
- **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,
- **optionnal** autocomplete *templating*,
- **99%** of the python logic is encapsulated in "autocomplete" 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*
- **add another** *also available outside the admin*,
- **simple** python, html and javascript, easy to hack, PEP8 compliant, tested

CHAPTER 2

README

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

CHAPTER 3

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 4

Resources

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

- Video demo graciously hosted by Youtube,
- 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 5

Demo

See test_project/README

CHAPTER 6

Full documentation

6.1 django-autocomplete-light demo

Test projects live in django-autocomplete-light root, they are:

- test_project: demonstrates basic autocomplete features
- test_remote_project: demonstrates advanced features

6.2 test_project: basic features and examples

Virtualenv is a great solution to isolate python environments. If necessary, you can install it from your package manager or the python package manager, ie.:

```
sudo easy_install virtualenv
```

6.2.1 Install last release

Install packages from PyPi and the test project from Github:

```
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
```

6.2.2 Or install the development version

Install directly from github:

```
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
```

6.2.3 Usage

- Run the server,
- Connect to */admin/*, ie. http://localhost:8000/admin/,
- Login with user "test" and password "test",
- Try the many example applications,

6.2.4 Database

A working SQLite database is shipped, but you can make your own ie.:

```
cd test_project
rm -rf db.sqlite
./manage.py syncdb --noinput
./manage.py migrate
./manage.py cities_light
```

Note that *test_project/project_specific/models.py* filters cities from certain countries.

6.3 test_remote_project: advanced features and examples

The autocomplete can suggest results from a remote API - objects that do not exist in the local database.

This project demonstrates how test_remote_project can provide autocomplete suggestions using the database from test_project.

6.3.1 Usage

In one console:

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

In another:

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

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

Then, connect to http://localhost:8001/admin/remote_autocomplete/address/add/ and the city autocomplete should propose cities from both projects.

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

6.4 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.

6.4.1 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() and **any form import** for that matter:

```
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' %}
```

Note that you should adapt the static.html template to your needs at some point, because its purpose is to work for all projects, not to be optimal for your project.

6.4.2 Quick admin integration

To enable autocomplete form widgets, you need to load:

- jQuery
- autocomplete_light/autocomplete.js
- autocomplete_light/widget.js

Optionally:

- autocomplete_light/style.css
- autocomplete_light/remote.js

A quick way to enable all this in the admin, is to replace template `admin/base_site.html`, ie.:

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

{% block extrahead %}
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javasc
    {% include 'autocomplete_light/static.html' %}
{% endblock %}
```

Create `yourapp/autocomplete_light_registry.py`, you can copy this example autocomplete *Registry*:

```
import autocomplete_light

from cities_light.models import City

autocomplete_light.register(City, search_fields=('search_names',),
    autocomplete_js_attributes={'placeholder': 'city name ..'})
```

At this point, the easiest is to use autocomplete-light's modelform_factory shortcut directly in `yourapp/admin.py`, ie.:

```
from django.contrib import admin

import autocomplete_light

from models import Address

class AddressAdmin(admin.ModelAdmin):
    form = autocomplete_light.modelform_factory(Address)

admin.site.register(Address, AddressAdmin)
```

6.4.3 Quick form integration

Example models:

```
from django.db import models
from django.core import urlresolvers


class Widget(models.Model):
    city = models.ForeignKey('cities_light.city', null=True, blank=True)
    users = models.ManyToManyField('auth.user', blank=True)

    def get_absolute_url(self):
        return urlresolvers.reverse('non_admin:widget_update', args=(self.pk,))
```

Example forms:

```
from django import forms

import autocomplete_light

from models import Widget

# in the case of this example, we could just have:
# WidgetForm = autocomplete_light.modelform_factory(Widget)
# but we'll not use this shortcut
```

```
class WidgetForm(forms.ModelForm):
    class Meta:
        widgets = autocomplete_light.get_widgets_dict(Widget)
        model = Widget
```

Example urls:

```
from django.conf.urls import patterns, url
from django.views import generic

from forms import WidgetForm
from models import Widget


urlpatterns = patterns('',
    url(r'widget/add/$', generic.CreateView.as_view(
        model=Widget, form_class=WidgetForm)),
    url(r'widget/(?P<pk>\d+)/update/$', generic.UpdateView.as_view(
        model=Widget, form_class=WidgetForm), name='widget_update'),
)
```

Note: It is not mandatory to use url namespaces.

Example template:

```
<html>
    <body>
        <form method="post" action="">
            {% csrf_token %}
            <table>
                {{ form.as_table }}
            </table>
            <input type="submit" />
        </form>

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

You can manually use autocomplete_light.ChoiceWidget or autocomplete_light.MultipleChoiceWidget for django's ModelChoiceField and ModelMultipleChoiceField respectively.

6.5 Making a global navigation autocomplete

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

There is a living example in test_project/navigation_autocomplete, which this page describes.

Note that there are many ways to implement such a feature, we're just describing a simple one.

6.5.1 A simple view

As we're just going to use autocomplete.js for this, we only need a view to render the autocomplete:

```
from django import shortcuts
from django.db.models import Q
from django.contrib.auth.models import User, Group

from cities_light.models import Country, Region, City


def navigation_autocomplete(request,
    template_name='navigation_autocomplete/autocomplete.html'):

    q = request.GET.get('q', '')
    context = {'q': q}

    queries = {}
    queries['users'] = User.objects.filter(
        Q(username__icontains=q) |
        Q(first_name__icontains=q) |
        Q(last_name__icontains=q) |
        Q(email__icontains=q)
    ).distinct()[:3]
    queries['groups'] = Group.objects.filter(name__icontains=q)[:3]
    queries['cities'] = City.objects.filter(search_names__icontains=q)[:3]
    queries['regions'] = Region.objects.filter(name_ascii__icontains=q)[:3]
    queries['countries'] = Country.objects.filter(name_ascii__icontains=q)[:3]

    context.update(queries)

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

And a trivial template (`test_project/navigation_autocomplete/templates/navigation_autocomplete/auto`

```
{% for country in countrys %}
<a style="display:block" href="{% url admin:cities_light_country_change country.pk %}">{{ country }}<
{% endfor %}
{% for region in regions %}
<a style="display:block" href="{% url admin:cities_light_region_change region.pk %}">{{ region }}</a>
{% endfor %}
{% for city in cities %}
<a style="display:block" href="{% url admin:cities_light_city_change city.pk %}">{{ city }}</a>
{% endfor %}
{% for group in groups %}
<a style="display:block" href="{% url admin:auth_group_change group.pk %}">{{ group }}</a>
{% endfor %}
{% for user in users %}
<a style="display:block" href="{% url admin:auth_user_change user.pk %}">{{ user }}</a>
{% endfor %}
```

And of course a url:

```
from django.conf.urls import patterns, url

urlpatterns = patterns('navigation_autocomplete.views',
    url(r'^$', 'navigation_autocomplete', name='navigation_autocomplete'),
)
```

6.5.2 A basic autocomplete configuration

That's a pretty basic usage of autocomplete.js (`test_project/navigation_autocomplete/templates/navigation_au`

```
<script type="text/javascript">
$(document).ready(function() {
    $('#navigation_autocomplete').yourlabsAutocomplete({
        url: '{% url navigation_autocomplete %}',
        choiceSelector: 'a',
    }).input.bind('selectChoice', function(e, choice, autocomplete) {
        document.location.href = choice.attr('href');
    });
});
</script>
```

Which works on such a simple input (`test_project/navigation_autocomplete/templates/navigation_autocomp`

```
<input type="text" name="q" id="navigation_autocomplete" style="width: 270px; font-size: 16px;" />
```

See how `admin/base_site.html` includes them:

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

{% block extrahead %}
    {# These are needed to enable autocompletes in forms #}
    <script src="{{ STATIC_URL }}jquery.js" type="text/javascript"></script>
    {% include 'autocomplete_light/static.html' %}

    {% if user.is_authenticated %}
        {# This script enables global navigation autocomplete #}
        {# refer to the docs about global navigation autocomplete for more information #}
        {% include 'navigation_autocomplete/script.html' %}
    {% endif %}
{% endblock %}

{% block branding %}
    <h1 style="display: inline-block" id="site-name">{% trans 'Autocomplete-light demo' %}</h1>

    {% if user.is_authenticated %}
        {% comment %}
        This is a simple input, used for global navigation autocomplete. It
        serves as an example, refer to the documentation to make a navigation
        autocomplete.

        FYI It leaves in test_project/navigation_autocomplete/templates
        {% endcomment %}
        {% include 'navigation_autocomplete/input.html' %}
    {% endif %}
{% endblock %}
```

6.6 Integration with forms

The purpose of this documentation is to describe every element in a chronological manner.

It is complementary with the quick documentation.

6.6.1 Django startup

Registry

Autocomplete basics

Examples

Simple model autocomplete:

```
import autocomplete_light

from cities_light.models import City

autocomplete_light.register(City, search_fields=('search_names',),
    autocomplete_js_attributes={'placeholder': 'city name ..'})
```

Slightly advanced autocomplete:

```
import autocomplete_light
from cities_light.models import Country, City
from django.contrib.auth.models import User, Group


class AutocompleteTaggableItems(autocomplete_light.AutocompleteGenericBase):
    choices = (
        User.objects.all(),
        Group.objects.all(),
        City.objects.all(),
        Country.objects.all(),
    )

    search_fields = (
        ('username', 'email'),
        ('name',),
        ('search_names',),
        ('name_ascii',),
    )


autocomplete_light.register(AutocompleteTaggableItems)
```

API

There are many autocompletes you can use, just to name a few:

- AutocompleteRestModelBase,
- AutocompleteGenericTemplate,
- AutocompleteModelTemplate,
- AutocompleteChoiceListBase ...

Each of them should have tests in autocomplete_light/tests and at least one example app in test_project/.

Forms

Note: Due to Django's issue #9321, you may have to use `autocomplete_light.FixedModelForm` instead of `django.forms.ModelForm`. Otherwise, you might see help text like 'Hold down "Control" key ...' for MultipleChoiceWidgets.

API

A more high level API is also available:

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`:

```
{% load static %}
<script type="text/javascript" src="{% static 'autocomplete_light/autocomplete.js' %}"></script>
<script type="text/javascript" src="{% static 'autocomplete_light/widget.js' %}"></script>
<script type="text/javascript" src="{% static 'autocomplete_light/addanother.js' %}"></script>
<script type="text/javascript" src="{% static 'autocomplete_light/text_widget.js' %}"></script>
<script type="text/javascript" src="{% static 'autocomplete_light/remote.js' %}"></script>
<link rel="stylesheet" type="text/css" href="{% static '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>
```

To enable autocomplete form widgets, you need to load:

- jQuery
- autocomplete_light/autocomplete.js
- autocomplete_light/widget.js

Optionally:

- autocomplete_light/style.css
- autocomplete_light/remote.js

A quick way to enable all this in the admin, is to replace template `admin/base_site.html`, ie.:

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

{% block extrahead %}
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javasc
    {% include 'autocomplete_light/static.html' %}
{% endblock %}
```

6.6.2 Widget in action

Widget definition

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

Example:

```
from django import forms
from django.contrib.auth.models import User
from cities_light.models import City
import autocomplete_light

from models import Profile


class ProfileForm(forms.ModelForm):
    user = forms.ModelChoiceField(User.objects.all(),
        widget=autocomplete_light.ChoiceWidget('UserAutocomplete'))

    cities = forms.ModelMultipleChoiceField(City.objects.all(),
        widget=autocomplete_light.MultipleChoiceWidget('CityAutocomplete'))

    class Meta:
        model = Profile
```

Widget rendering

This is what the default widget template looks like:

```
{% load i18n %}
{% load staticfiles %}
{% load autocomplete_light_tags %}
{% load url from future %}

<span class="autocomplete-light-widget {{ name }}
    {% if widget.widget_js_attributes.max_values == 1 %}single{% else %}multiple{% endif %}"
    id="{{ widget.html_id }}-wrapper"
    {{ widget.widget_js_attributes|autocomplete_light_data_attributes }}
    {{ widget.autocomplete_js_attributes|autocomplete_light_data_attributes:'autocomplete-' }}
    >

    {# a deck that should contain the list of selected options #}
    {{ choices }}
    <div id="{{ widget.html_id }}-deck" class="deck" >
        {% for choice in autocomplete.choices_for_values %}
            {{ choice|autocomplete_light_choice_html:autocomplete }}
        {% endfor %}
    </div>

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

    {# A link to add a new choice using a popup #}
    {% if autocomplete.add_another_url_name %}
    <a href="{% url autocomplete.add_another_url_name %}?_popup=1" class="autocomplete-add-another" i
            <img src="{% static 'admin/img/icon_addlink.gif' %}" width="10" height="10" alt="{% trans
        </a>
```

```
    {% endif %}

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

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

    <div style="display:none" class="choice-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 %}
        <div class="choice">
        </div>
    </div>
</span>
```

6.6.3 Javascript initialization

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

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

If you want to initialize the widget 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_widget({
        getValue: function(choice) {
            // your own logic to get the value from an html choice
            return ...;
        }
    });
});
```

Also, load *yourapp/autocomplete_light.js* in your override of autocomplete_light/static.html.

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

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

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

This is exactly what you need to use to make autocompletes that depend on each other.

Javascript cron

widget.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 widget, and adds it to the widget 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.

The reason for that is that adding and selecting an option from a select doesn't trigger the javascript change event, which is a hudge pity.

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 minimumCharacters 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 autocomplete view. That view delegates the rendering of the autocomplete box to the registered autocomplete.

Then, autocomplete.js recognizes options with a selector. By default, it is '[data-value]'. This means that any element with a data-value attribute in the autocomplete html is considered a selectable choice.

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

When an option is removed from the widget, widget.js also removes it from the hidden select.

6.7 Templating autocompletes

This documentation drives through the example app `template_autocomplete`, which is available in `test_project`.

6.7.1 API

In `autocomplete_light/autocomplete/__init__.py`, it is used as a mixin:

```
class AutocompleteModelBase(AutocompleteModel, AutocompleteBase):
    pass


class AutocompleteModelTemplate(AutocompleteModel, AutocompleteTemplate):
    pass
```

6.7.2 Example

In this case, all you have to do, is use `AutocompleteModelTemplate` instead of `AutocompleteModelBase`. For example, in `test_project/template_autocomplete/autocomplete_light_registry.py`:

```
import autocomplete_light

from models import TemplatedChoice

autocomplete_light.register(TemplatedChoice,
```

```
        autocomplete_light.AutocompleteModelTemplate,
        choice_template='template_autocomplete/templated_choice.html')
```

This example template makes choices clickable, it is `test_project/template_autocomplete/templates/template_au`

```
<div data-value="{{ choice.pk }}">
        <a href="{% url admin:template_autocomplete_templatedchoice_change choice.pk %}?_popup=1" tar
                {{ choice }}
        </a>
</div>
```

6.8 CharField autocompletes

django-tagging and derivates like django-tagging-ng provide a `TagField`, which is a `CharField` expecting comma separated tags. Behind the scenes, this field is parsed and `Tag` model instances are created and/or linked.

A stripped variant of `widget.js`, `text_widget.js`, enables autocompletion for such a field. To make it even easier, a stripped variant of `Widget`, `TextWidget`, automates configuration of `text_widget.js`.

Needless to say, `TextWidget` and `text_widget.js` have a structure that is consistent with `Widget` and `widget.js`.

It doesn't have many features for now, but feel free to participate to the project on GitHub.

As usual, a working example lives in test_project. in app `charfield_autocomplete`.

Warning: Note that this feature was added in version 1.0.16, if you have overloaded `autocomplete_light/static.html` from a previous version then you should make it load `autocomplete_light/text_widget.js` to get this new feature.

6.8.1 Example

This demonstrates a working usage of TextWidget:

```
from django import forms

import autocomplete_light

from models import Taggable


class TaggableForm(forms.ModelForm):
    class Meta:
        model = Taggable
        widgets = {
                'tags': autocomplete_light.TextWidget('TagAutocomplete'),
               }
```

FTR, using the form in the admin is still as easy:

```
from django.contrib import admin

from forms import TaggableForm
from models import Taggable
```

```
class TaggableAdmin(admin.ModelAdmin):
    form = TaggableForm
    list_display = ['name', 'tags']

admin.site.register(Taggable, TaggableAdmin)
```

So is registering an Autocomplete for Tag:

```
from tagging.models import Tag

import autocomplete_light


autocomplete_light.register(Tag)
```

6.8.2 Django-tagging

This demonstrates the models setup used for the above example, using django-taggit, which provides a normal CharField behaviour:

```
from django.db import models

from tagging.fields import TagField
import tagging


class Taggable(models.Model):
        name = models.CharField(max_length=50)
        tags = TagField(null=True, blank=True)

        def __unicode__(self):
                return self.name

tagging.register(Taggable, tag_descriptor_attr='etags')
```

6.8.3 Django-taggit

django-taggit does it slightly differently. It is supported by autocomplete_light as of 1.0.25, using the *autocomplete_light.contrib.taggit_tagfield* module.

6.9 Add another popup outside the admin

This documentation drives throught the example app `non_admin_add_another` which lives in `test_project`.

Implementing this feature is utterly simple and can be done in two steps:

- make your create view to return some script if called with `_popup=1`,
- add `add_another_url_name` attribute to your Autocomplete,

Warning: Note that this feature was added in version 1.0.21, if you have overloaded `autocomplete_light/static.html` from a previous version then you should make it load `autocomplete_light/addanother.js` to get this new feature.

6.9.1 Specifications

Consider such a model:

```
from django.db import models
from django.core import urlresolvers


class Widget(models.Model):
    name = models.CharField(max_length=100)
    widget = models.ForeignKey('self', null=True, blank=True,
        related_name='widget_fk')
    widgets = models.ManyToManyField('self', blank=True,
        related_name='widget_m2m')

    def get_absolute_url(self):
        return urlresolvers.reverse(
            'non_admin_add_another:widget_update', args=(self.pk,))

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

And we want to have add/update views outside the admin, with autocompletes for relations as well as a +/add-another button just like in the admin.

Technical details come from a blog post written by me a couple years ago, Howto: javascript popup form returning value for select like Django admin for foreign keys.

6.9.2 Create view

A create view opened via the add-another button should return such a body:

```
<script type="text/javascript">
opener.dismissAddAnotherPopup(
    window,
    "name of created model",
    "id of created model"
);
</script>
```

Note that you could also use `autocomplete_light.CreateView` which simply wraps around `django.views.generic.edit.CreateView.form_valid()` to do that, example usage:

```
from django.conf.urls import patterns, url
from django.views import generic

import autocomplete_light

from forms import WidgetForm
from models import Widget


urlpatterns = patterns('',
    url(r'widget/add/$', autocomplete_light.CreateView.as_view(
        model=Widget, form_class=WidgetForm), name='widget_create'),
    url(r'widget/(?P<pk>\d+)/update/$', generic.UpdateView.as_view(
        model=Widget, form_class=WidgetForm), name='widget_update'),
)
```

Note: It is not mandatory to use url namespaces.

6.9.3 Autocompletes

Simply register an Autocomplete for widget, with an `add_another_url_name` argument, for example:

```
from django.core import urlresolvers

import autocomplete_light

from models import Widget


autocomplete_light.register(Widget, add_another_url_name='non_admin_add_another:widget_create')
```

6.10 GenericForeignKey support

Generic relation support comes in two flavors:

- for django's generic foreign keys,
- and for django-generic-m2m's generic many to many in autocomplete_light.contrib.generic_m2m,

6.10.1 AutocompleteGeneric

Example

```
import autocomplete_light
from cities_light.models import Country, City
from django.contrib.auth.models import User, Group


class AutocompleteTaggableItems(autocomplete_light.AutocompleteGenericBase):
    choices = (
        User.objects.all(),
        Group.objects.all(),
        City.objects.all(),
        Country.objects.all(),
    )

    search_fields = (
        ('username', 'email'),
        ('name',),
        ('search_names',),
        ('name_ascii',),
    )


autocomplete_light.register(AutocompleteTaggableItems)
```

API

6.10.2 GenericModelChoiceField

Example

```
from django import forms

import autocomplete_light

from models import TaggedItem


class TaggedItemForm(autocomplete_light.GenericModelForm):
    content_object = autocomplete_light.GenericModelChoiceField(
        widget=autocomplete_light.ChoiceWidget(
            autocomplete='AutocompleteTaggableItems'))

    class Meta:
        model = TaggedItem
        exclude = ('content_type', 'object_id')
```

API

6.10.3 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, \
    GenericModelMultipleChoiceField

from models import ModelGroup


class ModelGroupForm(GenericModelForm):
    """
    Use AutocompleteTaggableItems defined in
```

```
    gfk_autocomplete.autocomplete_light_registry.
    """

    related = GenericModelMultipleChoiceField(
        widget=autocomplete_light.MultipleChoiceWidget(
            'AutocompleteTaggableItems'))

    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

6.11 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_remote_project*, where one project serves a full database of cities via an API to another.

6.11.1 Example

In test_remote_project/remote_autocomplete, of course you should not hardcode urls like that in actual projects:

```
from cities_light.contrib.autocompletes import *

import autocomplete_light


autocomplete_light.register(Country, CountryRestAutocomplete,
    source_url='http://localhost:8000/cities_light/country/')

autocomplete_light.register(Region, RegionRestAutocomplete,
    source_url='http://localhost:8000/cities_light/region/')

autocomplete_light.register(City, CityRestAutocomplete,
    source_url='http://localhost:8000/cities_light/city/')
```

Check out the documentation of RemoteCountryChannel and RemoteCityChannel for more.

6.11.2 API

6.11.3 Javascript fun

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

```
var RemoteAutocompleteWidget = {
    /*
    The default deck getValue() implementation just returns the PK from the
    choice HTML. RemoteAutocompleteWidget.getValue's implementation checks for
    a url too. If a url is found, it will post to that url and expect the pk to
    be in the response.

    This is how autocomplete-light supports proposing values that are not there
    in the database until user selection.
    */
    getValue: function(choice) {
        var value = choice.data('value');

        if (typeof(value)=='string' && isNaN(value) && value.match(/^http:/)) {
            $.ajax(this.autocompleteOptions.url, {
                async: false,
                type: 'post',
                data: {
                    'value': value,
                },
                success: function(text, jqXHR, textStatus) {
                    value = text;
                }
            });

            choice.data('value', value);
        }

        return value;
    }
}

$(document).bind('yourlabsWidgetReady', function() {
    // Instanciate decks with RemoteAutocompleteWidget as override for all widgets with
    // autocomplete 'remote'.
    $('.autocomplete-light-widget[data-bootstrap=rest_model]').live('initialize', function() {
        $(this).yourlabsWidget(RemoteAutocompleteWidget);
    });
});
```

6.12 Dependencies between autocompletes

This means that the selected value in an autocomplete widget is used to filter choices from another autocomplete widget.

This page drives through the example in test_project/dependant_autocomplete/.

6.12.1 Specifications

Consider such a model:

```
from django.db import models


class Dummy(models.Model):
    parent = models.ForeignKey('self', null=True, blank=True)
    country = models.ForeignKey('cities_light.country')
    region = models.ForeignKey('cities_light.region')

    def __unicode__(self):
        return u'%s %s' % (self.country, self.region)
```

And we want two autocompletes in the form, and make the "region" autocomplete to be filtered using the value of the "country" autocomplete.

6.12.2 Autocompletes

Register an Autocomplete for Region that is able to use 'country_id' GET parameter to filter choices:

```
import autocomplete_light

from cities_light.models import Country, Region

autocomplete_light.register(Country, search_fields=('name', 'name_ascii',),
    autocomplete_js_attributes={'placeholder': 'country name ..'})


class AutocompleteRegion(autocomplete_light.AutocompleteModelBase):
    autocomplete_js_attributes={'placeholder': 'region name ..'}

    def choices_for_request(self):
        q = self.request.GET.get('q', '')
        country_id = self.request.GET.get('country_id', None)

        choices = self.choices.all()
        if q:
            choices = choices.filter(name_ascii__icontains=q)
        if country_id:
            choices = choices.filter(country_id=country_id)

        return self.order_choices(choices)[0:self.limit_choices]

autocomplete_light.register(Region, AutocompleteRegion)
```

6.12.3 Javascript

Actually, a normal modelform is sufficient. But it was decided to use Form.Media to load the extra javascript:

```
from django import forms

import autocomplete_light

from models import Dummy
```

```
class DummyForm(forms.ModelForm):
    class Media:
        """
        We're currently using Media here, but that forced to move the
        javascript from the footer to the extrahead block ...

        So that example might change when this situation annoys someone a lot.
        """
        js = ('dependant_autocomplete.js',)

    class Meta:
        model = Dummy
        widgets = autocomplete_light.get_widgets_dict(Dummy)
```

That's the piece of javascript that ties the two autocompletes:

```
$(document).ready(function() {
    $('.autocomplete-light-widget select[name$=country]').live('change', function() {
        var countrySelectElement = $(this);
        var regionSelectElement = $('#' + $(this).attr('id').replace('country', 'region'));
        var regionWidgetElement = regionSelectElement.parents('.autocomplete-light-widget');

        // When the country select changes
        value = $(this).val();

        if (value) {
            // If value is contains something, add it to autocomplete.data
            regionWidgetElement.yourlabsWidget().autocomplete.data = {
                'country_id': value[0],
            };
        } else {
            // If value is empty, empty autocomplete.data
            regionWidgetElement.yourlabsWidget().autocomplete.data = {}
        }

        // example debug statements, that does not replace using breakbpoints and a proper debugger
        // console.log($(this), 'changed to', value);
        // console.log(regionWidgetElement, 'data is', regionWidgetElement.yourlabsWidget().autocomp
    })
});
```

6.12.4 Conclusion

Again, there are many ways to acheive this. It's just a working example you can test in the demo, you may copy it and adapt it to your needs.

6.13 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.

6.13.1 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 7

Javascript API

Work in progress:

- autocomplete.js
- widget.js
- addanother.js
- remote.js
- text_widget.js

CHAPTER 8

When things go wrong

There is a convenience view to visualize the registry, login as staff, and open the autocomplete url, for example: /autocomplete_light/.

Ensure that:

- jquery is loaded,
- `autocomplete_light/static.html` is included once, it should load `autocomplete.js`, `widget.js` and `style.css`,
- your form uses autocomplete_light widgets,
- your channels are properly defined see `/autocomplete/` if you included `autocomplete_light.urls` with prefix `/autocomplete/`.

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 9

Indices and tables

- genindex
- modindex
- search