django-autocomplete-light Documentation

Release 1.4.9

James Pic

January 25, 2017

Contents

django-autocomplete-light's purpose is to enable autocompletes quickly and properly in a django project: it is the fruit of **years of R&D**. It was designed for Django so that every part overridable or reusable independently. It is stable, tested, documented and fully supported: it tries to be a good neighbour in Django ecosystem.

CHAPTER 1

Features

Features include:

- charfield, foreign key, many to many autocomplete widgets,
- generic foreign key, generic many to many autocomplete widgets,
- remote API backed-autocompletes,
- django template engine support for autocompletes, enabling you to include images etc ...
- 100% overridable HTML, CSS, Python and Javascript: there is no variable hidden far down in the scope anywhere.
- add-another popup supported outside the admin too.
- keyboard is supported with enter, tab and arrows by default.

Each feature has a live example and is fully documented. It is also designed and documented so that you create your own awesome features too.

CHAPTER 2

Resources

Resources include:

- **Documentation** graciously hosted by RTFD
- Live demo graciously hosted by PythonAnywhere,
- Video demo graciously hosted by Youtube,
- Mailing list graciously hosted by Google
- Git graciously hosted by GitHub,
- Package graciously hosted by PyPi,
- Continuous integration graciously hosted by Travis-ci

CHAPTER 3

Demo

You can run test projects for a local demo in a temporary virtualenv.

3.1 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
```

3.1.1 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
```

3.1.2 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
```

3.1.3 Usage

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

3.1.4 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.

3.2 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.

3.2.1 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*.

CHAPTER 4

Install

Click on any instruction step for details.

4.1 Install the `django-autocomplete-light` package with pip

Install the stable release, preferably in a virtualenv:

```
pip install django-autocomplete-light
```

Or the development version:

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

4.2 Append `'autocomplete_light'` to `settings.INSTALLED_APPS`

Enable templates and static files by adding `autocomplete_light` to settings.INSTALLED_APPS which is editable in `settings.py`, it can look like this:

```
INSTALLED_APPS = [
    # [...] your list of app packages is here, add this:
    'autocomplete_light',
]
```

4.3 Call `autocomplete_light.autodiscover()` *before* `admin.autodiscover()`

In `urls.py`, call `autocomplete_light.autodiscover()` before `admin.autodiscover()`, it can look like this:

```
import autocomplete_light
# import every app/autocomplete_light_registry.py
autocomplete_light.autodiscover()

import admin
admin.autodiscover()
```

4.4 Include `autocomplete_light.urls`

Install the autocomplete view and staff debug view in `urls.py` using the include function, it can look like this:

```
# Django 1.5:
from django.conf.urls import patterns, url, include

# In Django 1.4:
# from django.conf.urls.default import patterns, url, include

urlpatterns = patterns('',
    # [...] your url patterns are here
    url(r'^autocomplete/', include('autocomplete_light.urls')),
)
```

4.5 Ensure understanding of `django.contrib.staticfiles`

Ensure that you understand django-staticfiles, if you don't try this article or refer to official howto and topic.

4.6 Include `autocomplete_light/static.html` after loading jquery.js (>=1.7)

Load the javascript scripts after loading `jquery.js`, it can look like this:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.js" type="text/javascript"></sc
{% include 'autocomplete_light/static.html' %}
```

4.7 Optionaly include it in `admin/base_site.html` too

For admin support, override `admin/base_site.html`. It could look like this:

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

{% block extrahead %}
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.js" type="text/javascript">
    {% include 'autocomplete_light/static.html' %}
{% endblock %}
```

Note: There is **nothing** magic in how the javascript loads. This means that you can use django-compressor or anything.

If you didn't click any, and this is your first install: bravo !

CHAPTER 5

Upgrade

Run `pip install -U django-autocomplete-light`. Check the CHANGELOG for BC (Backward Compatibility) breaks. There should be none for minor version upgrades ie. from 1.1.3 to 1.1.22.

CHAPTER 6

Tutorial

Learn the concepts by doing useful things.

6.1 Enable an autocomplete in admin forms in two steps: high level API concepts

6.1.1 `autocomplete_light.register()` shortcut to generate and register Autocomplete classes

Register an Autocomplete for your model in `your_app/autocomplete_light_registry.py`, it can look like this:

```
import autocomplete_light
from models import Person

# This will generate a PersonAutocomplete class
autocomplete_light.register(Person,
    # Just like in ModelAdmin.search_fields
    search_fields=['^first_name', 'last_name'],
    # This will actually html attribute data-placeholder which will set
    # javascript attribute widget.autocomplete.placeholder.
    autocomplete_js_attributes={'placeholder': 'Other model name ?',},
)
```

Because `PersonAutocomplete` is registered, `AutocompleteView.get()` can proxy `PersonAutocomplete.autocomplete_html()`. This means that openning `/autocomplete/PersonAutocomplete/` will call `AutocompleteView.get()` which will in turn call `PersonAutocomplete.autocomplete_html()`.

Also `AutocompleteView.post()` would proxy `PersonAutocomplete.post()` if it was defined. It could be useful to build your own features like on-the-fly object creation using *Javascript method overrides* like the *remote autocomplete*.

Warning: Note that this would make **all** `Person` public. Fine tuning security is explained later in this tutorial in section *Overriding the queryset of a model autocomplete to secure an Autocomplete*.

`autocomplete_light.register()` works by passing the extra keyword arguments like `search_fields` to the Python `type()` function. This means that extra keyword arguments will be used as class attributes of the generated class. An equivalent version of the above code would be:

```
class PersonAutocomplete(autocomplete_light.AutocompleteModelBase):
    search_fields = ['^first_name', 'last_name']
    autocomplete_js_attributes={'placeholder': 'Other model name ?',}
    model = Person
autocomplete_light.register(PersonAutocomplete)
```

Note: If you wanted, you could override the default `AutocompleteModelBase` used by `autocomplete_light.register()` to generate `Autocomplete` classes.

It could look like this (in urls.py):

```
autocomplete_light.registry.autocomplete_model_base = YourAutocompleteModelBase
autocomplete_light.autodiscover()
```

6.1.2 `modelform_factory()` shortcut to generate ModelForms in the admin

Make the admin `Order` form that uses `PersonAutocomplete`, in `your_app/admin.py`:

```
from django.contrib import admin
import autocomplete_light
from models import Order

class OrderAdmin(admin.ModelAdmin):
    # This will generate a ModelForm
    form = autocomplete_light.modelform_factory(Order)
admin.site.register(Order)
```

There are other ways to generate forms, depending on your needs. If you just wanted to replace selects in the admin then autocomplete_light's job is done by now !

6.2 Making Autocomplete classes

6.2.1 Create a basic list-backed autocomplete class

Class attributes are thread safe because `autocomplete_light.register()` always create a class copy. So, registering a custom Autocomplete class for your model in `your_app/autocomplete_light_registry.py` could look like this:

```
import autocomplete_light

class OsAutocomplete(autocomplete_light.AutocompleteListBase):
    choices = ['Linux', 'BSD', 'Minix']

autocomplete_light.register(OsAutocomplete)
```

6.2.2 Using a template to render the autocomplete

You could use `AutocompleteListTemplate` instead:

```
import autocomplete_light

class OsAutocomplete(autocomplete_light.AutocompleteListTemplate):
    choices = ['Linux', 'BSD', 'Minix']
    autocomplete_template = 'your_autocomplete_box.html'

autocomplete_light.register(OsAutocomplete)
```

Note: In reality, AutocompleteListBase inherits from both AutocompleteList and AutocompleteBase, and AutocompleteListTemplate inherits from both AutocompleteList and AutocompleteTemplate. It is the same for the other Autocomplete: AutocompleteModel + AutocompleteTemplate = AutocompleteModelTemplate and so on.

6.2.3 Create a basic model autocomplete class

Registering a custom Autocomplete class for your model in `your_app/autocomplete_light_registry.py` can look like this:

```
import autocomplete_light

from models import Person

class PersonAutocomplete(autocomplete_light.AutocompleteModelBase):
    search_fields = ['^first_name', 'last_name']
autocomplete_light.register(Person, PersonAutocomplete)
```

Note: An equivalent of this example would be:

```
autocomplete_light.register(Person,
    search_fields=['^first_name', 'last_name'])
```

6.2.4 Overriding the queryset of a model autocomplete to secure an Autocomplete

You can override any method of the Autocomplete class. Filtering choices based on the request user could look like this:

```
import autocomplete_light

from models import Person

class PersonAutocomplete(autocomplete_light.AutocompleteModelBase):
    search_fields = ['^first_name', 'last_name'])

    def choices_for_request(self):
        choices = super(PersonAutocomplete, self).choices_for_request()

        if not self.request.user.is_staff:
            choices = choices.filter(private=False)

        return choices

autocomplete_light.register(Person, PersonAutocomplete)
```

6.2.5 Registering the same Autocomplete class for several autocompletes

This code registers an autocomplete with name 'ContactAutocomplete':

```
autocomplete_light.register(ContactAutocomplete)
```

To register two autocompletes with the same class, pass in a name argument:

```
autocomplete_light.register(ContactAutocomplete, name='Person',
    choices=Person.objects.filter(is_company=False))
autocomplete_light.register(ContactAutocomplete, name='Company',
    choices=Person.objects.filter(is_company=True))
```

6.3 Your own form classes

6.3.1 Working around Django bug #9321: *Hold down "Control" ...*

If any autocomplete widget renders with a message like 'Hold down "Control" to select multiple items at once', it is because of Django bug #9321. A trivial fix is to use `autocomplete_light.FixedModelForm`.

`FixedModelForm` inherits from `django.forms.ModelForm` and only takes care or removing this message. It remains compatible and can be used as a drop-in replacement for ''ModelForm'.'

Of course, `FixedModelForm` is **not** required, but might prove helpful.

6.3.2 Override a default relation select in `ModelForm.Meta.widgets`

You can override the default relation select as such:

```
from django import forms

import autocomplete_light

from models import Order, Person

class OrderForm(forms.ModelForm):
    class Meta:
        model = Order
        widgets = autocomplete_light.get_widgets_dict(Order)
```

6.3.3 Or in a `ModelChoiceField` or similar

Now use `PersonAutocomplete` in a `ChoiceWidget` ie. for a `ForeignKey`, it can look like this:

```
from django import forms

import autocomplete_light

from models import Order, Person

class OrderForm(forms.ModelForm):
    person = forms.ModelChoiceField(Person.objects.all(),
        widget=autocomplete_light.ChoiceWidget('PersonAutocomplete'))
```

```
    class Meta:
        model = Order
```

6.3.4 Using your own form in a `ModelAdmin`

You can use this form in the admin too, it can look like this:

```
from django.contrib import admin

from forms import OrderForm
from models import Order

class OrderAdmin(admin.ModelAdmin):
    form = OrderForm
admin.site.register(Order, OrderAdmin)
```

Note: Ok, this has nothing to do with `django-autocomplete-light` because it is plain Django, but still it might be useful to someone.

6.3.5 Using autocomplete widgets in non model-forms

There are 3 kinds of widgets:

- `autocomplete_light.ChoiceWidget` has a hidden `<select>` which works for `django.forms.ChoiceField`,
- `autocomplete_light.MultipleChoiceWidget` has a hidden `<select multiple="multiple">` which works for `django.forms.MultipleChoiceField`,
- `autocomplete_light.TextWidget` just enables an autocomplete on its `<input>` and works for `django.forms.CharField`.

For example:

```
# Using widgets directly in any kind of form.
class NonModelForm(forms.Form):
    user = forms.ModelChoiceField(User.objects.all(),
        widget=autocomplete_light.ChoiceWidget('UserAutocomplete'))

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

    tags = forms.CharField(
        widget=autocomplete_light.TextWidget('TagAutocomplete'))
```

6.3.6 Overriding a JS option in Python

Javascript widget options can be set in Python via the `widget_js_attributes` keyword argument. And javascript autocomplete options can be set in Python via the `autocomplete_js_attributes`.

Those can be set either on an Autocomplete class, either using the `register()` shortcut, either via the Widget constructor.

Per Autocomplete class

```
class AutocompleteYourModel(autocomplete_light.AutocompleteModelTemplate):
    template_name = 'your_app/your_special_choice_template.html'

    autocomplete_js_attributes = {
        # This will actually data-autocomplete-minimum-characters which
        # will set widget.autocomplete.minimumCharacters.
        'minimum_characters': 4,
    }

    widget_js_attributes = {
        # That will set data-max-values which will set widget.maxValues
        'max_values': 6,
    }
```

Per registered Autocomplete

```
autocomplete_light.register(City,
    # Those have priority over the class attributes
    autocomplete_js_attributes={
        'minimum_characters': 0,
        'placeholder': 'City name ?',
    }
    widget_js_attributes = {
        'max_values': 6,
    }
)
```

Per widget

```
class SomeForm(forms.Form):
    cities = forms.ModelMultipleChoiceField(City.objects.all(),
        widget=autocomplete_light.MultipleChoiceWidget('CityAutocomplete',
            # Those attributes have priority over the Autocomplete ones.
            autocomplete_js_attributes={'minimum_characters': 0,
                                        'placeholder': 'Choose 3 cities ...'},
            widget_js_attributes={'max_values': 3}))
```

6.4 Javascript API concepts

django-autocomplete-light provides consistent JS plugins. A concept that you understand for one plugin is likely to be appliable for others.

6.4.1 Using `$.yourlabsAutocomplete` to create a navigation autocomplete

If your website has a lot of data, it might be useful to add a search input somewhere in the design. For example, there is a search input in Facebook's header. You will also notice that the search input in Facebook provides an autocomplete which allows to directly navigate to a particular object's detail page. This allows a visitor to jump to a particular page with very few effort.

Our autocomplete script is designed to support this kind of autocomplete. It can be enabled on an input field and query the server for a rendered autocomplete with anything like images and nifty design. Just create a view that renders just a list of links based on `request.GET.q`.

Then you can use it to make a global navigation autocomplete using `autocomplete.js` directly. It can look like this:

```
// Make a javascript Autocomplete object and set it up
var autocomplete = $('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
});
```

So when the user clicks on a link of the autocomplete box which is generated by your view: it is like if he clicked on a normal link.

Note: This is because `autocomplete.js` is simple and stupid, it can't even generate an autocomplete box HTML ! But on the other hand you can use any server side caching or templates that you want ... So maybe it's a good thing ?

6.4.2 Using the `choiceSelector` option to enable keyboard navigation

Because the script doesn't know what HTML the server returns, it is nice to tell it how to recognize choices in the autocomplete box HTML:

```
$('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
    choiceSelector: 'a',
});
```

This will allow to use the keyboard arrows up/down to navigate between choices.

6.4.3 Using the `selectChoice` event to enable keyboard choice selection

`autocomplete.js` doesn't do anything but trigger `selectChoice` on the input when a choice is selected either with mouse **or keyboard**, let's enable some action:

```
$('#yourInput').bind('selectChoice', function(e, choice, autocomplete) {
    window.location.href = choice.attr('href');
});
```

Note: Well, not only doesn't autocomplete.js generate the autocomplete box HTML, but it can't even do anything uppon choice selection ! What a stupid script. On the other hand it does allow to plug in radically different behaviours (ie. ModelChoiceWidget, TextWidget, ...) so maybe it's a good thing.

6.4.4 Combining the above to make a navigation autocomplete for mouse and keyboard

You've learned that you can have a fully functional navigation autocomplete like on Facebook with just this:

```
$('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
    choiceSelector: 'a',
```

```
}).bind('selectChoice', function(e, choice, autocomplete) {
    window.location.href = choice.attr('href');
});
```

6.4.5 Override autocomplete JS options in JS

The array passed to the plugin function will actually be used to $.extend the autocomplete instance, so you can override any option, ie:

```
$('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
    // Hide after 200ms of mouseout
    hideAfter: 200,
    // Choices are elements with data-url attribute in the autocomplete
    choiceSelector: '[data-url]',
    // Show the autocomplete after only 1 character in the input.
    minimumCharacters: 1,
    // Override the placeholder attribute in the input:
    placeholder: '{% trans 'Type your search here ...' %}',
    // Append the autocomplete HTML somewhere else:
    appendAutocomplete: $('#yourElement'),
    // Override zindex:
    autocompleteZIndex: 1000,
});
```

Note: The pattern is the same for all plugins provided by django-autocomplete-light.

6.4.6 Override autocomplete JS methods

Overriding methods works the same, ie:

```
$('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
    choiceSelector: '[data-url]',
    getQuery: function() {
        return this.input.val() + '&search_all=' + $('#searchAll').val();
    },
    hasChanged: function() {
        return true; // disable cache
    },
});
```

Note: The pattern is the same for all plugins provided by django-autocomplete-light.

6.4.7 Overload autocomplete JS methods

Use call to call a parent method. This example automatically selects the choice if there is only one:

```
$(document).ready(function() {
    var autocomplete = $('#id_city_text').yourlabsAutocomplete();
```

```
    autocomplete.show = function(html) {
        yourlabs.Autocomplete.prototype.show.call(this, html)
        var choices = this.box.find(this.choiceSelector);

        if (choices.length == 1) {
            this.input.trigger('selectChoice', [choices, this]);
        }
    }
});
```

6.4.8 Get an existing autocomplete object and chain autocompletes

You can use the jQuery plugin `yourlabsAutocomplete()` to get an existing autocomplete object. Which makes chaining autocompletes with other form fields as easy as:

```
$('#country').change(function() {
    $('#yourInput').yourlabsAutocomplete().data = {
        'country': $(this).val();
    }
});
```

6.4.9 Overriding widget JS methods

The widget js plugin will only bootstrap widgets which have `data-bootstrap="normal"`. Which means that you should first name your new bootstrapping method to ensure that the default behaviour doesn't get in the way.

```
autocomplete_light.register(City,
    widget_js_attributes={'bootstrap': 'your-custom-bootstrap'})
```

Note: You could do this at various level, by setting the `bootstrap` argument on a widget instance, via `register()` or directly on an autocomplete class. See Overriding JS options in Python for details.

Now, you can instanciate the widget yourself like this:

```
$(document).bind('yourlabsWidgetReady', function() {
    $('.your.autocomplete-light-widget[data-bootstrap=your-custom-bootstrap]').live('initialize', fu
        $(this).yourlabsWidget({
            // Override options passed to $.yourlabsAutocomplete() from here
            autocompleteOptions: {
                url: '{% url "your_autocomplete_url" %}',
                // Override any autocomplete option in this array if you want
                choiceSelector: '[data-id]',
            },
            // Override some widget options, allow 3 choices:
            maxValues: 3,
            // or method:
            getValue: function(choice) {
                // This is the method that returns the value to use for the
                // hidden select option based on the HTML of the selected
                // choice.
                //
                // This is where you could make a non-async post request to
                // this.autocomplete.url for example. The default is:
                return choice.data('id')
```

```
            },
        })
    });
});
```

You can use the remote autocomplete as an example.

Note: You could of course call `$.yourlabsWidget()` directly, but using the `yourlabsWidgetReady` event takes advantage of the built-in DOMNodeInserted event: your widgets will also work with dynamically created widgets (ie. admin inlines).

You should now fully understand the concepts and be able to do literally what you want.

CHAPTER 7

Topics

Using just the concepts you've learned in the tutorial, we've built-in several really cool things, backed by live examples.

7.1 Templating autocompletes

This documentation drives through the example app `test_project/template_autocomplete`.

You can use `AutocompleteTemplate` as a mixin just like us:

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

You could also directly inherit from `AutocompleteModelTemplate`.

Anyway, this enable two new attributes: `choice_template` and `autocomplete_template`

7.1.1 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" ta
                {{ choice }}
        </a>
</div>
```

7.1.2 Alternative

FTR, here's another way to do it, assuming your models have a *get_absolute_update_url* method defined:

```
class AutocompleteEditableModelBase(autocomplete_light.AutocompleteModelBase):
    choice_html_format = u'''
        <span class="div" data-value="%s">%s</span>
        <a href="%s" title="%s"><img src="%s%s" /></a>
    '''

    def choice_html(self, choice):
        """
        Return a choice formated according to self.choice_html_format.
        """
        return self.choice_html_format % (
            self.choice_value(choice), self.choice_label(choice),
            choice.get_absolute_update_url(), _(u'Update'),
            settings.STATIC_URL, 'admin/img/icon_changelink.gif')

autocomplete_light.register(AppCategory, AutocompleteEditableModelBase,
    add_another_url_name='appstore_appcategory_create')

autocomplete_light.register(AppFeature, AutocompleteEditableModelBase,
    add_another_url_name='appstore_appfeature_create')
```

7.2 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.

7.2.1 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

```
{% load url from future %}

{% for country in countrys %}
<a class="div choice" href="{% url 'admin:cities_light_country_change' country.pk %}">{{ country }}<
{% endfor %}
{% for region in regions %}
<a class="div choice" href="{% url 'admin:cities_light_region_change' region.pk %}">{{ region }}</a>
{% endfor %}
{% for city in cities %}
<a class="div choice" href="{% url 'admin:cities_light_city_change' city.pk %}">{{ city }}</a>
{% endfor %}
{% for group in groups %}
<a class="div choice" href="{% url 'admin:auth_group_change' group.pk %}">{{ group }}</a>
{% endfor %}
{% for user in users %}
<a class="div choice" 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'),
)
```

7.2.2 A basic autocomplete configuration

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

```
{% load url from future %}

<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;" />
<style type="text/css">
    /* cancel out django default for now, or choices are white on white */
    #header a.choice:link, #header a.choice:visited {
```

```
        color: black;
    }
</style>
```

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 %}
```

7.3 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.

7.3.1 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 TaggableInline(admin.TabularInline):
    form = TaggableForm
    model = Taggable


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

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

So is registering an Autocomplete for Tag:

```
from tagging.models import Tag

import autocomplete_light


autocomplete_light.register(Tag)
```

7.3.2 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)
    parent = models.ForeignKey('self', null=True, blank=True)

    def __unicode__(self):
        return self.name

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

7.3.3 Django-taggit

For django-taggit, you need autocomplete_light.contrib.taggit_tagfield.

7.4 AutocompleteGeneric, for GenericForeignKey or GenericManyToMany

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,

7.4.1 AutocompleteGeneric

Example using `AutocompleteGeneric` as shown in `test_project/gfk_autocomplete/autocomplete_light_regi`

```
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)
```

7.4.2 GenericModelForm and GenericModelChoiceField

Example using `GenericModelForm` and `GenericModelChoiceField` as shown in `test_project/gfk_autocomplete/forms.py`:

```
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',
            autocomplete_js_attributes={'minimum_characters': 0}))

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

7.4.3 GenericManyToMany

Example

Consider this example model with a generic many-to-many relation descriptor `related` as in `test_project/generic_m2m_autocomplete/models.py`:

```
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 `GenericModelForm` and `GenericModelMultipleChoiceField` usage as per `test_project/generic_m2m_autocomplete/forms.py`:

```
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
```

The form defined above can directly be using in the admin:

```
from django.contrib import admin

from models import ModelGroup
from forms import ModelGroupForm


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

7.5 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/.

7.5.1 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.

7.5.2 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)
```

7.5.3 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() {
    $('body').on('change', '.autocomplete-light-widget select[name$=country]', 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
    })
});
```

7.5.4 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.

7.6 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.

7.6.1 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.

7.6.2 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.

7.6.3 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')
```

7.7 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.

7.7.1 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.

7.7.2 API

7.7.3 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(/^https?:/)) {
            $.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'.
    $('body').on('initialize', '.autocomplete-light-widget[data-bootstrap=rest_model]', function() {
        $(this).yourlabsWidget(RemoteAutocompleteWidget);
    });
});
```

7.8 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.

7.8.1 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)
```

7.9 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 !

7.10 Voodoo black magic

This cookbook is a work in progress. Please report any error or things that could be explained better ! And make pull requests heh ...

7.10.1 High level Basics

Various cooking recipes `your_app/autocomplete_light_registry.py`:

```
# This actually creates a thread safe subclass of AutocompleteModelBase.
autocomplete_light.register(SomeModel)

# If NewModel.get_absolute_url or get_absolute_update_url is defined, this
# will look more fancy
autocomplete_light.register(NewModel,
    autocomplete_light.AutocompleteModelTemplate)

# Extra **kwargs are used as class properties in the subclass.
autocomplete_light.register(SomeModel,
    # SomeModel is already registered, re-register with custom name
    name='AutocompleteSomeModelNew',
    # Filter the queryset
    choices=SomeModel.objects.filter(new=True))

# It is possible to override javascript options from Python.
autocomplete_light.register(OtherModel,
    autocomplete_js_attributes={
        # This will actually data-minimum-characters which
        # will set widget.autocomplete.minimumCharacters.
        'minimum_characters': 0,
        'placeholder': 'Other model name ?',
    }
)

# But you can make your subclass yourself and override methods.
class YourModelAutocomplete(autocomplete_light.AutocompleteModelTemplate):
    template_name = 'your_app/your_special_choice_template.html'

    autocomplete_js_attributes = {
```

```
        'minimum_characters': 4,
    }

    widget_js_attributes = {
        # That will set data-max-values which will set widget.maxValues
        'max_values': 6,
    }

    def choices_for_request(self):
        """ Return choices for a particular request """
        return super(YourModelAutocomplete, self).choices_for_request(
            ).exclude(extra=self.request.GET['extra'])

# Just pass the class to register and it'll subclass it to be thread safe.
autocomplete_light.register(YourModel, YourModelAutocomplete)

# This will subclass the subclass, using extra kwargs as class attributes.
autocomplete_light.register(YourModel, YourModelAutocomplete,
    # Again, registering another autocomplete for the same model, requires
    # registration under a different name
    name='YourModelOtherAutocomplete',
    # Extra **kwargs passed to register have priority.
    choice_template='your_app/other_template.html')
```

Various cooking recipes for `your_app/forms.py`:

```
# Use as much registered autocompletes as possible.
SomeModelForm = autocomplete_light.modelform_factory(SomeModel,
    exclude=('some_field'))

# Same with a custom modelform, using Meta.get_widgets_dict().
class CustomModelForm(forms.ModelForm):
    some_extra_field = forms.CharField()

    class Meta:
        model = SomeModel
        widgets = autocomplete_light.get_widgets_dict(SomeModel)

# Using widgets directly in any kind of form.
class NonModelForm(forms.Form):
    user = forms.ModelChoiceField(User.objects.all(),
        widget=autocomplete_light.ChoiceWidget('UserAutocomplete'))

    cities = forms.ModelMultipleChoiceField(City.objects.all(),
        widget=autocomplete_light.MultipleChoiceWidget('CityAutocomplete',
            # Those attributes have priority over the Autocomplete ones.
            autocomplete_js_attributes={'minimum_characters': 0,
                                        'placeholder': 'Choose 3 cities ...'},
            widget_js_attributes={'max_values': 3}))

    tags = autocomplete_light.TextWidget('TagAutocomplete')
```

7.10.2 Low level basics

This is something you probably won't need in the mean time. But it can turn out to be useful so here it is.

Various cooking recipes for `autocomplete.js`, useful if you want to use it manually for example to make a

navigation autocomplete like facebook:

```
// Use default options, element id attribute and url options are required:
var autocomplete = $('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}'
});

// Because the jQuery plugin uses a registry, you can get the autocomplete
// instance again by calling yourlabsAutocomplete() again, and patch it:
$('#country').change(function() {
    $('#yourInput').yourlabsAutocomplete().data = {
        'country': $(this).val();
    }
});
// And that's actually how to do chained autocompletes.

// The array passed to the plugin will actually be used to $.extend the
// autocomplete instance, so you can override any option:
$('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
    // Hide after 200ms of mouseout
    hideAfter: 200,
    // Choices are elements with data-url attribute in the autocomplete
    choiceSelector: '[data-url]',
    // Show the autocomplete after only 1 character in the input.
    minimumCharacters: 1,
    // Override the placeholder attribute in the input:
    placeholder: '{% trans 'Type your search here ...' %}',
    // Append the autocomplete HTML somewhere else:
    appendAutocomplete: $('#yourElement'),
    // Override zindex:
    autocompleteZIndex: 1000,
});

// Or any method:
$('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
    choiceSelector: '[data-url]',
    getQuery: function() {
        return this.input.val() + '&search_all=' + $('#searchAll').val();
    },
    hasChanged: function() {
        return true; // disable cache
    },
});

// autocomplete.js doesn't do anything but trigger selectChoice when
// an option is selected, let's enable some action:
$('#yourInput').bind('selectChoice', function(e, choice, autocomplete) {
    window.location.href = choice.attr('href');
});

// For a simple navigation autocomplete, it could look like:
$('#yourInput').yourlabsAutocomplete({
    url: '{% url "your_autocomplete_url" %}',
    choiceSelector: 'a',
}).bind('selectChoice', function(e, choice, autocomplete) {
    window.location.href = choice.attr('href');
});
```

Using *widget.js* is pretty much the same:

```
$('#yourWidget').yourlabsWidget({
    autocompleteOptions: {
        url: '{% url "your_autocomplete_url" %}',
        // Override any autocomplete option in this array if you want
        choiceSelector: '[data-id]',
    },
    // Override some widget options, allow 3 choices:
    maxValues: 3,
    // or method:
    getValue: function(choice) {
        return choice.data('id'),
    },
});

// Supporting dynamically added widgets (ie. inlines) is
// possible by using "solid initialization":
$(document).bind('yourlabsWidgetReady', function() {
    $('.your.autocomplete-light-widget[data-bootstrap=your-custom-bootstrap]').live('initialize', fu
        $(this).yourlabsWidget({
            // your options ...
        })
    });
});
// This method takes advantage of the default DOMNodeInserted observer
// served by widget.js
```

There are some differences with *autocomplete.js*:

- widget expect a certain HTML structure by default,
- widget options can be overridden from HTML too,
- widget can be instanciated automatically via the default bootstrap

Hence the widget.js HTML cookbook:

```
<!--
- class=autocomplete-light-widget: get picked up by widget.js defaults,
- data-bootstrap=normal: Rely on automatic bootstrap because
  if don't need to override any method, but you could change
  that and make your own bootstrap, enabling you to make
  chained autocomplete, create options, whatever ...
- data-max-values: override a widget option
- data-minimum-characters: override an autocomplete option,
-->
<span
    class="autocomplete-light-widget"
    data-bootstrap="normal"
    data-max-values="3"
    data-minimum-characters="0"
>

    <!-- Expected structure: have an input -->
    <input type="text" id="some-unique-id" />

    <!--
    Default expected structure: have a .deck element to append selected
    choices too:
    -->
```

```
    <span class="deck">
        <!-- Suppose a choice was already selected: -->
        <span class="choice" data-value="1234">Option #1234</span>
    </span>

    <!--
    Default expected structure: have a multiple select.value-select:
    -->
    <select style="display:none" class="value-select" name="your_input" multiple="multiple">
        <!-- If option 1234 was already selected: -->
        <option value="1234">Option #1234</option>
    </select>

    <!--
    Default expected structure: a .remove element that will be appended to
    choices, and that will de-select them on click:
    -->
    <span style="display:none" class="remove">Remove this choice</span>

    <!--
    Finally, supporting new options to be created directly in the select in
    javascript (ie. add another) is possible with a .choice-template. Of
    course, you can't take this very far, since all you have is the new
    option's value and html.
    -->
    <span style="display:none" class="choice-template">
        <span class="choice">
        </span>
    </span>
</span>
```

Read everything about the registry and widgets.

CHAPTER 8

Integration with external apps

8.1 Support for django-generic-m2m

See GenericManyToMany documentation.

8.2 Support for django-hvad

8.3 Support for 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.

CHAPTER 9

FAQ

9.1 Why not use Widget.Media ?

In the early versions (0.1) of django-autocomplete-light, we had widgets defining the Media class like this:

```
class AutocompleteWidget(forms.SelectMultiple):
    class Media:
        js = ('autocomplete_light/autocomplete.js',)
```

This caused a problem if you want to load jquery and autocomplete.js globally **anyway** and **anywhere** in the admin to have a global navigation autocomplete: it would double load the scripts.

Also, this didn't work well with django-compressor and other cool ways of deploying the JS.

So, in the next version, I added a dependency management system. Which sucked and was removed right away to finally keep it simple and stupid as we have it today.

9.2 How to ask for help ?

The best way to ask for help is:

- fork the repo,
- add a simple way to reproduce your problem in a new app of test_project, try to keep it minimal,
- open an issue on github and mention your fork.

Really, it takes quite some time for me to clean pasted code and put up an example app it would be really cool if you could help me with that !

If you don't want to do the fork and the reproduce case, then you should better ask on StackOverflow and you might be lucky (just tag your question with django-autocomplete-light to ensure that I find it).

CHAPTER 10

API reference

10.1 Python API reference

10.1.1 Registry

10.1.2 Autocompletes

AutocompleteInterface

AutocompleteBase

AutocompleteTemplate

Other Autocompletes

AutocompleteModel

AutocompleteGeneric

10.1.3 `import autocomplete_light`

See everything available in `autocomplete_light/__init__.py`:

```
"""
Provide tools to enable nice autocompletes in your Django project.
"""
from .registry import AutocompleteRegistry, registry, register, autodiscover
from .autocomplete import *
from .widgets import ChoiceWidget, MultipleChoiceWidget, TextWidget
from .forms import get_widgets_dict, modelform_factory, FixedModelForm
from .generic import GenericModelForm, GenericModelChoiceField
from .views import CreateView
from .exceptions import AutocompleteNotRegistered

ModelForm = FixedModelForm
```

10.1.4 Widgets

WidgetBase

ChoiceWidget

MultipleChoiceWidget

TextWidget

10.1.5 Form shortcuts

10.1.6 Views

RegistryView

AutocompleteView

CreateView

Generic foreign key support

10.1.7 `autocomplete_light.contrib.generic_m2m`

10.1.8 `autocomplete_light.contrib.taggit_tagfield`

10.2 Javascript API reference

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

CHAPTER 11

Indices and tables

- genindex
- modindex
- search