django-autocomplete-light Documentation

Release 0.1

James Pic

March 05, 2016

Contents

CHAPTER 1

Features

This app fills all your ajax autocomplete needs:

- global navigation autocomplete like on http://betspire.com
- autocomplete widget for ModelChoiceField and ModelMultipleChoiceField
- **0 hack** required for admin integration, just use a form that uses the widget
- **no jQuery-ui** required, the autocomplete script is as simple as possible
- **all** the design of the autocompletes is encapsulated in template, unlimited design possibilities
- **99%** of the python logic is encapsulated in “channel” classes, unlimited development possibilities
- **99%** the javascript logic is encapsulated in a class, you can override any attribute or method, unlimited development possibilities
- **0 inline javascript** you can load the javascript just before </body> for best page loading performance
- **simple** python, html and javascript, easy to hack
- **less sucking** code, no funny hacks, clean api, as few code as possible, that also means this is not for pushovers

CHAPTER 2

Full documentation

2.1 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.

2.1.1 Quick install

Install the package:

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

Add to INSTALLED_APPS: 'autocomplete_light'

Add to urls:

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

Add before admin.autodiscover():

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

Add to your base template:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js" type="text/javascript">
{% load autocomplete_light_tags %}
{% autocomplete_light_static %}
```

2.1.2 Quick admin integration

Create yourapp/autocomplete_light_registry.py:

```
import autocomplete_light

from models import Author

autocomplete_light.register(Author)
```

In yourapp/admin.py:

```python
from django.contrib import admin

import autocomplete_light

from models import Book

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

2.1.3 Quick form integration

AutocompleteWidget is usable on ModelChoiceField and ModelMultipleChoiceField.

2.2 Integration with forms

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

It is complementary with the quick documentation.

2.2.1 Django startup

Registry

Autodiscovery is part of the *autocomplete_light.registry* module:

Autodiscovery is the first thing that happens as it is called early in urls.py:

autocomplete_light.registry.autodiscover fills *autocomplete_light.registry.registry*, which is an instance of ChannelRegistry:

Channels

As you can see, registration creates a channel if it is only passed a model. You'll want to make your own channel class, this is what a channel looks like:

Forms

To save you some boilerplate, a couple of helpers are provided:

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">
{% load autocomplete_light_tags %}
{% autocomplete_light_static %}
```

That said, 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>
```

2.2.2 Widget in action

Widget definition

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

Widget rendering

This is what the default widget template looks like:

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

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

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

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

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

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

    {# a hidden div that serves as template for the 'remove from deck' button #}
    <div style="display:none" class="remove">
```

```
        {# This will be appended to results on the deck, it's the remove button #}
        X
    </div>

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

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

Javascript initialization

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

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

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

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

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

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

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

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

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

Javascript cron

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

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

Javascript events

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

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

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

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

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

2.3 Making a global navigation autocomplete

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

2.3.1 Create the view

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

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

Only two things matter:

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

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

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

Example

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

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

from art.models import Artist, Artwork

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

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

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

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

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

And in project_specific/autocomplete.html:

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

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

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

{% if not options %}
    <li><em>{% trans 'No options' %}</em></li>
    <li><a href="{% url 'haystack_search' %}?q={{ q|urlencode }}">{% blocktrans %}Search for {{ q }}
{% endif %}
```

```
</ul>
```

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

As for the url, it looks like this:

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

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

2.3.2 Create the input

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

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

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

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

2.3.3 Loading the script

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

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

2.3.4 Script usage

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

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

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

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

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

        // hide the autocomplete
        autocomplete.hide();

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

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

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

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

CHAPTER 3

When things go wrong

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

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

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

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

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

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

Also, don't hesitate to do pull requests !

Indices and tables

- genindex
- modindex
- search