django-autocomplete-light Documentation

Release 2.3.3

James Pic

Mar 28, 2018

Contents

1 Projects upgrading to Django 1.9 3

2 Features 5

3 Resources 7

4 Live Demo 9

5 Installation 13

6 Tutorial 17

7 Reference and design documentation 21

8 Topics 37

9 FAQ 47

10 API: find hidden gems 53

11 Upgrade 67

12 Documentation that has not yet been ported to v2 73

13 Indices and tables 81

Python Module Index 83

django-autocomplete-light's purpose is to enable autocompletes quickly and properly in a django project: it is the fruit of half a decade of R&D and thousands of contributions. 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

Projects upgrading to Django 1.9

DAL has been ready for Django 1.9 since April 2015 thanks to @blueyed & @jpic. **HOWEVER** due to the app loading refactor in 1.9 you should apply the following:

```
find . -name '*.py' | xargs perl -i -pe 's/import autocomplete_light/from␣
↪autocomplete_light import shortcuts as autocomplete_light/'
```

See the test_project running on Django 1.9 and its new cool admin theme: http://dal-yourlabs.rhcloud.com/admin (test:test).

CHAPTER 2

Features

Features include:

- charfield, foreign key, many to many autocomplete widgets,
- generic foreign key, generic many to many autocomplete widgets,
- 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.
- Django 1.7 to 1.10, PyPy, Python 2 and 3, PostgreSQL, SQLite, MySQL

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 3

Resources

Resources include:

- **Documentation** graciously hosted by RTFD
- Live demo graciously hosted by RedHat, thanks to PythonAnywhere for hosting it in the past,
- 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
- **Online paid support** provided via HackHands,

CHAPTER 4

Live Demo

While you can use the live demo hosted by RedHat on OpenShift, you can run test projects for a local demo in a temporary virtualenv.

4.1 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
```

4.1.1 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 --recursive https://jpic@github.com/yourlabs/django-autocomplete-light.git
pip install django-autocomplete-light
cd django-autocomplete-light/test_project
pip install -r requirements.txt
./manage.py runserver
```

4.1.2 Or install the development version

Install directly from github:

```
rm -rf autocomplete_light_env/

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

4.1.3 Usage

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

4.1.4 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.

4.2 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.

4.2.1 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 5

Installation

Advanced Django users are likely to install it in no time by following this step-list. Click on a step for details.

5.1 Install the `django-autocomplete-light>=2.0.0pre` 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.git
↪#egg=autocomplete_light
```

5.2 Append `'autocomplete_light'` to `settings.INSTALLED_APPS` before `django.contrib.admin`

Enable templates and static files by adding `autocomplete_light` to settings.INSTALLED_APPS which is editable in `settings.py`. For example:

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

5.3 Django < 1.7 support: call `autocomplete_light.autodiscover()` *before* `admin.autodiscover()`

In `urls.py`, call `autocomplete_light.autodiscover()` before `admin.autodiscover()` **and before any import of a form with autocompletes**. It might look like this:

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

import admin
admin.autodiscover()
```

Also, if you have `yourapp.views` which imports a form that has autocomplete, say `SomeForm`, this will work:

```
import autocomplete_light.shortcuts as al
al.autodiscover()

from yourapp.views import SomeCreateView
```

But this won't:

```
from yourapp.views import SomeCreateView

import autocomplete_light.shortcuts as al
al.autodiscover()
```

That is because auto-discovery of autocomplete classes should happen before definition of forms using autocompletes.

5.4 Include `autocomplete_light.urls`

Install the autocomplete view and staff debug view in `urls.py` using the include function. Example:

```
# Django 1.4 onwards:
from django.conf.urls import patterns, url, include

# 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')),
)
```

5.5 Ensure you understand `django.contrib.staticfiles`

If you're just trying this out using the Django runserver, that will take care of staticfiles for you - but for production, you'll need to understand django-staticfiles to get everything working properly. If you don't, here's a good article about staticfiles or refer to the official Django howto and Django topic.

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

Load the javascript scripts after loading `jquery.js`, for example by doing:

```
{% load static %}
<script src="{% static 'admin/js/vendor/jquery/jquery.min.js' %}" type="text/
↪javascript"></script>
{% include 'autocomplete_light/static.html' %}
```

5.7 Optionally include it in `admin/base_site.html` too

For admin support, override `admin/base_site.html`. For example:

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

{% block extrahead %}
    {{ block.super }}
    <script src="{% static 'admin/js/vendor/jquery/jquery.min.js' %}" type="text/
↪javascript"></script>
    {% 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 6

Tutorial

Enabling autocompletes inside and outside of the admin has become piece of cake.

6.1 Quick start: adding simple autocompletes

This tutorial guides you through the process of enabling autocomplete for a simple form. We'll enable autocompletion for `Person` on the form of an `Order` model, i.e. we want to get autocompletion for `Person` objects when we type the person's first or last name.

For this to work, we have to register an autocomplete for the `Person` model. The autocomplete tells autocomplete-light how to render your autocompletion and what model fields to search in (like `first_name`, `last_name`, ...).

When we have defined how a `Person` autocompletion should look like, we have to enable it in the `Order` form. This is done by modifying the `Order` form so that autocomplete-light's special form fields are used instead of the ones built into Django. autocomplete-light provides a handy wrapper around Django's modelforms which makes this a very easy thing to do.

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:

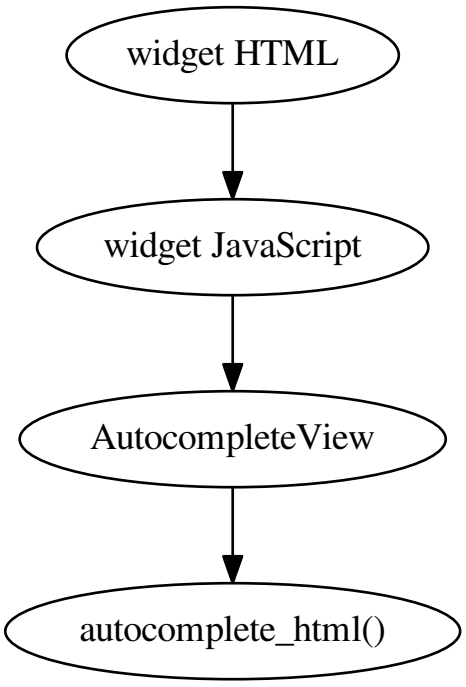
```
import autocomplete_light.shortcuts as al
from models import Person

# This will generate a PersonAutocomplete class.
al.register(Person,
    # Just like in ModelAdmin.search_fields.
    search_fields=['^first_name', 'last_name'],
    attrs={
        # This will set the input placeholder attribute:
```

```
        'placeholder': 'Other model name ?',
        # This will set the yourlabs.Autocomplete.minimumCharacters
        # options, the naming conversion is handled by jQuery.
        'data-autocomplete-minimum-characters': 1,
    },
    # This will set the data-widget-maximum-values attribute on the
    # widget container element, and will be set to
    # yourlabs.Widget.maximumValues (jQuery handles the naming
    # conversion).
    widget_attrs={
        'data-widget-maximum-values': 4,
        # Enable modern-style widget !
        'class': 'modern-style',
    },
)
```

Note: If using **Django >= 1.7**, you might as well do *register()* calls directly in your `AppConfig.ready()` as demonstrated in the example app: `autocomplete_light.example_apps.app_config_without_registry_file`.

AutocompleteView.get() can proxy *PersonAutocomplete.autocomplete_html()* because `PersonAutocomplete` is registered. 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() generates an Autocomplete class, passing the extra keyword arguments like *AutocompleteModel.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']
    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 your project's `urls.py`):

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

Refer to the *Autocomplete classes* documentation for details, it is the first chapter of the *the reference documentation*.

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

First, ensure that scripts are *installed in the admin base template*.

Then, enabling autocompletes in the admin is as simple as overriding `ModelAdmin.form` in `your_app/admin.py`. You can use the `modelform_factory()` shortcut as such:

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

Refer to the *Form, fields and widgets* documentation for other ways of making forms, it is the second chapter of the *the reference documentation*.

6.1.3 `autocomplete_light.ModelForm` to generate Autocomplete fields, the DRY way

First, ensure that *scripts are properly installed in your template*.

Then, you can use `autocomplete_light.ModelForm` to replace automatic `Select` and `SelectMultiple` widgets which renders `<select>` HTML inputs by autocompletion widgets:

```
class OrderModelForm(autocomplete_light.ModelForm):
    class Meta:
        model = Order
```

Note that the first Autocomplete class registered for a model becomes the default Autocomplete for that model. If you have registered several Autocomplete classes for a given model, you probably want to use a different Autocomplete class depending on the form using `Meta.autocomplete_names`:

```
class OrderModelForm(autocomplete_light.ModelForm):
    class Meta:
        autocomplete_names = {'company': 'PublicCompanyAutocomplete'}
        model = Order
```

`autocomplete_light.ModelForm` respects `Meta.fields` and `Meta.exclude`. However, you can enable or disable `autocomplete_light.ModelForm`'s behaviour in the same fashion with `Meta.autocomplete_fields` and `Meta.autocomplete_exclude`:

```
class OrderModelForm(autocomplete_light.ModelForm):
    class Meta:
        model = Order
        # only enable autocompletes on 'person' and 'product' fields
        autocomplete_fields = ('person', 'product')

class PersonModelForm(autocomplete_light.ModelForm):
    class Meta:
        model = Order
        # do not make 'category' an autocomplete field
        autocomplete_exclude = ('category',)
```

Also, it will automatically enable autocompletes on generic foreign keys and generic many to many relations if you have at least one generic Autocomplete class register (typically an `AutocompleteGenericBase`).

For more documentation, continue reading *the reference documentation*.

CHAPTER 7

Reference and design documentation

If you need anything more than just enabling autocompletes in the admin, then you should understand django-autocomplete-light's architecture. Because you can override any part of it.

The architecture is based on 3 main parts which you can override to build insanely creative features as many users already did.

7.1 Autocomplete classes

Note: This chapter assumes that you have been through the entire *Quick start: adding simple autocompletes*.

7.1.1 Design documentation

Any class which implements `AutocompleteInterface` is guaranteed to work because it provides the methods that are expected by the view which serves autocomplete contents from ajax, and the methods that are expected by the form field for validation and by the form widget for rendering.

However, implementing those methods directly would result in duplicate code, hence `AutocompleteBase`. It contains all necessary rendering logic but lacks any business-logic, which means that it is not connected to any data.

If you wanted to make an Autocomplete class that implements business-logic based on a python list, you would end up with `AutocompleteList`.

As you need both business-logic and rendering logic for an Autocomplete class to be completely usable, you would mix both `AutocompleteBase` and `AutocompleteList` resulting in `AutocompleteListBase`:

If you wanted to re-use your python list business logic with a template based rendering logic, you would mix `AutocompleteList` and `AutocompleteTemplate`, resulting in `AutocompleteListTemplate`:

So far, you should understand that rendering and business logic are not coupled in autocomplete classes: you can make your own business logic (ie. using redis, haystack ...) and re-use an existing rendering logic (ie. `AutocompleteBase` or `AutocompleteTemplate`) and vice-versa.

For an exhaustive list of Autocomplete classes, refer to *the Autocomplete API doc*.

One last thing: Autocomplete classes should be *registered* so that the view can serve them and that form fields and widget be able to re-use them. The registry itself is rather simple and filled with good intentions, refer to *Registry API* documentation.

7.1.2 Examples

Create a basic list-backed autocomplete class

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

```
import autocomplete_light.shortcuts as al

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

al.register(OsAutocomplete)
```

First, we imported `autocomplete_light`'s module. It should contain everything you need.

Then, we subclassed `autocomplete_light.AutocompleteListBase`, setting a list of OSes in the `choices` attribute.

Finally, we registered the Autocomplete class. It will be registered with the class name by default.

Note: Another way of achieving the above using the *register* shortcut could be:

```
autocomplete_light.register(autocomplete_light.AutocompleteListBase,
    name='OsAutocomplete', choices=['Linux', 'BSD', 'Minix'])
```

Using a template to render the autocomplete

You could use `AutocompleteListTemplate` instead of `AutocompleteListBase`:

```
import autocomplete_light.shortcuts as al

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

al.register(OsAutocomplete)
```

Inheriting from `AutocompleteListTemplate` instead of `AutocompleteListBase` like as show in the **previous** example enables two optionnal options:

- *`autocomplete_template`* which we have customized, if we hadn't then *`AutocompleteTemplate.choice_html()`* would have fallen back on the parent *`AutocompleteBase.choice_html()`*,
- *`choice_template`* which we haven't set, so *`AutocompleteTemplate.choice_html()`* will fall back on the parent *`AutocompleteBase.choice_html()`*,

See *Design documentation* for details.

Note: Another way of achieving the above could be:

```
autocomplete_light.register(autocomplete_light.AutocompleteListTemplate,
    name='OsAutocomplete', choices=['Linux', 'BSD', 'Minix'],
    autocomplete_template='your_autocomplete_box.html')
```

Creating a basic model autocomplete class

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

```
from models import Person

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

In the same fashion, you could have used `AutocompleteModelTemplate` instead of `AutocompleteModelBase`. You can see that the inheritance diagram follows the same pattern:

Note: An equivalent of this example would be:

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

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:

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

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

        return super(PersonAutocomplete, self).choices_for_request()

# we have specified PersonAutocomplete.model, so we don't have to repeat
# the model class as argument for register()
autocomplete_light.register(PersonAutocomplete)
```

It is very important to note here, that *clean()* **will** raise a `ValidationError` if a model is selected in a `ModelChoiceField` or `ModelMultipleChoiceField`

Note: Use at your own discretion, as this can cause problems when a choice is no longer part of the queryset, just like with django's Select widget.

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))
```

7.2 Form, fields and widgets

Note: This chapter assumes that you have been through *Quick start: adding simple autocompletes* and *Autocomplete classes*.

7.2.1 Design documentation

This app provides optionnal helpers to make forms:

- `autocomplete_light.modelform_factory` which wraps around django's modelform_factory but uses the heroic `autocomplete_light.ModelForm`.
- `autocomplete_light.ModelForm`: the heroic ModelForm which ties all our loosely coupled tools together:
  - `SelectMultipleHelpTextRemovalMixin`, which removes the "Hold down control or command to select more than one" help text on autocomplete widgets (fixing Django ticket #9321),
  - `VirtualFieldHandlingMixin` which enables support for generic foreign keys,
  - `GenericM2MRelatedObjectDescriptorHandlingMixin` which enables support for generic many to many, if django-genericm2m is installed,
  - `ModelFormMetaclass` which enables `FormfieldCallback` to replace the default form field creator replacing <*select*> with autocompletes for relations and creates generic foreign key and generic many to many fields.

You probably already know that Django has form-fields for validation and each form-field has a widget for rendering logic.

`autocomplete_light.FieldBase` makes a form field field rely on an Autocomplete class for initial choices and validation (hail DRY configuration !), it is used as a mixin to make some simple field classes:

- *`autocomplete_light.ChoiceField`*,
- *`autocomplete_light.MultipleChoiceField`*,
- *`autocomplete_light.ModelChoiceField`*,
- *`autocomplete_light.ModelMultipleChoiceField`*,
- *`autocomplete_light.GenericModelChoiceField`*, and
- *`autocomplete_light.GenericModelMultipleChoiceField`*.

In the very same fashion, *autcomplete_light.WidgetBase* renders a template which should contain:

- a hidden <select> field containing real field values,
- a visible <input> field which has the autocomplete,
- a **deck** which contains the list of selected values,
- add-another optionnal link, because add-another works outside the admin,
- a hidden choice template, which is copied when a choice is created on the fly (ie. with add-another).

It is used as a mixin to make some simple widget classes:

- *autocomplete_light.ChoiceWidget*,
- *autocomplete_light.MultipleChoiceWidget*,
- *autocomplete_light.TextWidget*.

7.2.2 Examples

This basic example demonstrates how to use an autocomplete form field in a form:

```
class YourForm(forms.Form):
    os = autocomplete_light.ChoiceField('OsAutocomplete')
```

Using autocomplete_light.ModelForm

Consider such a model which have every kind of relations that are supported out of the box:

```
class FullModel(models.Model):
    name = models.CharField(max_length=200)

    oto = models.OneToOneField('self', related_name='reverse_oto')
    fk = models.ForeignKey('self', related_name='reverse_fk')
    mtm = models.ManyToManyField('self', related_name='reverse_mtm')

    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    gfk = generic.GenericForeignKey("content_type", "object_id")

    # that's generic many to many as per django-generic-m2m
    gmtm = RelatedObjectsDescriptor()
```

Assuming that you have registered an Autocomplete for FullModel **and** a generic Autocomplete, then autocomplete_light.ModelForm will contain 5 autocompletion fields by default: *oto*, *fk*, *mtm*, *gfk* and *gmtm*.

```
class FullModelModelForm(autocomplete_light.ModelForm):
    class Meta:
        model = FullModel
        # add for django 1.6:
        fields = '__all__'
```

autocomplete_light.ModelForm gives autocompletion super powers to django.forms.ModelForm. To disable the fk input for example:

```
class FullModelModelForm(autocomplete_light.ModelForm):
    class Meta:
        model = FullModel
        exclude = ['fk']
```

Or, to just get the default <select> widget for the fk field:

```
class FullModelModelForm(autocomplete_light.ModelForm):
    class Meta:
        model = FullModel
        autocomplete_exclude = ['fk']
```

In the same fashion, you can use Meta.fields and Meta.autocomplete_fields. To the difference that they all understand generic foreign key names and generic relation names in addition to regular model fields.

Not using autocomplete_light.ModelForm

Instead of using our autocomplete_light.ModelForm, you could create such a ModelForm using our mixins:

```
class YourModelForm(autocomplete_light.SelectMultipleHelpTextRemovalMixin,
        autocomplete_light.VirtualFieldHandlingMixin,
        autocomplete_light.GenericM2MRelatedObjectDescriptorHandlingMixin,
        forms.ModelForm):
    pass
```

This way, you get a fully working ModelForm which does **not** handle any field generation. You **can** use form fields directly though, which is demonstrated in the next example.

Using form fields directly

You might want to use form fields directly for any reason:

- you don't want to or can't extend autocomplete_light.ModelForm,
- you want to override a field, ie. if you have several Autocomplete classes registered for a model or for generic relations and you want to specify it,
- you want to override any option like placeholder, help_text and so on.

Considering the model of the above example, this is how you could do it:

```
class FullModelModelForm(autocomplete_light.ModelForm):
    # Demonstrate how to use a form field directly
    oto = autocomplete_light.ModelChoiceField('FullModelAutocomplete')
    fk = autocomplete_light.ModelChoiceField('FullModelAutocomplete')
    m2m = autocomplete_light.ModelMultipleChoiceField('FullModelAutocomplete')
    # It will use the default generic Autocomplete class by default
    gfk = autocomplete_light.GenericModelChoiceField()
    gmtm = autocomplete_light.GenericModelMultipleChoiceField()

    class Meta:
        model = FullModel
        # django 1.6:
        fields = '__all__'
```

As you see, it's as easy as 1-2-3, but keep in mind that this can break DRY: *Model field's help_text and verbose_name are lost when overriding the widget*.

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.

7.3 Scripts: the javascript side of autocompletes

Note: This chapter assumes that you have been through *Quick start: adding simple autocompletes* and *Autocomplete classes* and *Form, fields and widgets*.

7.3.1 Design documentation

Before installing your own autocomplete scripts, you should know about the humble provided scripts:

- `autocomplete.js` provides `yourlabs.Autocomplete` via the `$.yourlabsAutocomplete()` jQuery extension: add an autocomplete box to a text input, it can be used on its own to create a navigation autocomplete like facebook and all the cool kids out there.
- `widget.js` provides `yourlabs.Widget` via the `$.yourlabsWidget()` jQuery extension: combine an text input with an autocomplete box with a django form field which is represented by a hidden `<select>`.
- `addanother.js` enables adding options to a `<select>` via a popup from outside the admin, code mostly comes from Django admin BTW,
- `remote.js` provides `yourlabs.RemoteAutocompleteWidget`, which extends `yourlabs.Widget` and overrides `yourlabs.Widget.getValue` to create choices on-the-fly.
- `text_widget.js` provides `yourlabs.TextWidget`, used to manage the value of a text input that has an autocomplete box.

Why a new autocomplete script you might ask ? What makes this script unique is that it relies on the server to render the contents of the autocomplete-box. This means that you can fully design it like you want, including new HTML tags like `<img>`, using template tags like `{% url %}`, and so on.

If you want to change something on the javascript side, chances are that you will be better off overriding a method like `yourlabs.RemoteAutocompleteWidget` instead of installing your own script right away.

What you need to know is that:

- widgets don't render any inline javascript: the have HTML attributes that will tell the scripts how to instanciate objects with `$.yourlabsWidget()`, `$.yourlabsTextWidget()` and so on. This allows to support dynamically inserted widgets ie. with a dynamic formsets inside or outside of django admin.

- the particular attribute that is watched for is `data-bootstrap`. If an HTML element with class `.autocomplete-light-widget` is found or created with `data-bootstrap="normal"` then `widget.js` will call `$.yourlabsWidget`.
- if you customize `data-bootstrap`, `widget.js` will not do anything and you are free to implement your script, either by extending a provided a class, either using a third-party script, either completely from scratch.

7.3.2 Examples

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

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.

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

`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:

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

Refer to *Making a global navigation autocomplete* for complete help on making a navigation autocomplete.

Overriding a JS option in Python

Javascript widget and autocomplete objects options can be overidden via HTML data attributes:

- `yourlabs.Autocomplete` will use any `data-autocomplete-*` attribute **on the input tag**,

- `yourlabs.Widget` will use any `data-widget-*` attribute **on the widget container**.

Those can be set in Python either with `register()`, as Autocomplete class attributes or as widget attributes. See next examples for details.

Per registered Autocomplete

These options can be set with the `register()` shortcut:

```
autocomplete_light.register(Person,
    attrs={
        'placeholder': 'foo',
        'data-autocomplete-minimum-characters': 0
    },
    widget_attrs={'data-widget-maximum-values': 4}
)
```

Per Autocomplete class

Or equivalently on a Python Autocomplete class:

```
class YourAutocomplete(autocomplete_light.AutocompleteModelBase):
    model = Person
    attrs={
        'placeholder': 'foo',
        'data-autocomplete-minimum-characters': 0
    },
    widget_attrs={'data-widget-maximum-values': 4}
```

Per widget

Or via the Python widget class:

```
autocomplete_light.ChoiceWidget('FooAutocomplete',
    attrs={
        'placeholder': 'foo',
        'data-autocomplete-minimum-characters': 0
    }
    widget_attrs={'data-widget-maximum-values': 4}
)
```

NOTE the difference of the option name here. It is `attrs` to match django and not `attrs`. Note that `Autocomplete.attrs` might be renamed to `Autocomplete.attrs` before v2 hits RC.

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.

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.

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]);
        }
    }
});
```

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();
    }
});
```

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_attrs={'data-widget-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', function() {
        $(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).

7.4 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 ...

7.4.1 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,
    attrs={
        # This will actually data-autocomplete-minimum-characters which
        # will set widget.autocomplete.minimumCharacters.
        'data-autocomplete-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'

    attrs = {
        'data-mininum-minimum-characters': 4,
        'placeholder': 'choose your model',
    }

    widget_attrs = {
        # That will set widget.maximumValues, naming conversion is done by
        # jQuery.data()
        'data-widget-maximum-values': 6,
        'class': 'your-custom-class',
    }

    def choices_for_request(self):
```

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

# 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 autocomplete_light.ModelForm
class CustomModelForm(autocomplete_light.ModelForm):
    # autocomplete_light.ModelForm will set up the fields for you
    some_extra_field = forms.CharField()

    class Meta:
        model = SomeModel

# Using form fields directly in any kind of form
class NonModelForm(forms.Form):
    user = autocomplete_light.ModelChoiceField('UserAutocomplete')

    cities = autocomplete_light.ModelMultipleChoiceField('CityAutocomplete',
        widget=autocomplete_light.MultipleChoiceWidget('CityAutocomplete',
            # Those attributes have priority over the Autocomplete ones.
            attrs={'data-autocomplete-minimum-characters': 0,
                   'placeholder': 'Choose 3 cities ...'},
            widget_attrs={'data-widget-maximum-values': 3}))

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

7.4.2 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',
}).input.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:
    maximumValues: 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', function() {
        $(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,
- any data-widget-* attribute will override yourlabs.Widget js option,
- data-widget-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-widget-maximum-values: override a widget option maximumValues, note
  that the naming conversion is done by jQuery.data().
-->
<span
    class="autocomplete-light-widget"
    data-widget-bootstrap="normal"
    data-widget-maximum-values="3"
>

    <!--
    Expected structure: have an input, it can set override default
    autocomplete options with data-autocomplete-* attributes, naming
    conversion is done by jQuery.data().
    -->
    <input
        type="text"
        data-autocomplete-minimum-characters="0"
        data-autocomplete-url="/foo"
    />

    <!--
```

```
    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

Topics

Using just the concepts you've learned in the reference, here are some of the things you can do.

8.1 Styling autocompletes

A complete autocomplete widget has three parts you can style individually:

- the autocomplete widget, rendered on the form,
- the autocomplete box, fetched by ajax,
- choices presented by both the autocomplete box and *widget deck*.

Note that a choice HTML element is copied from the autocomplete box into the deck uppon selection. It is then appended a "remove" element, that will remove the choice uppon click.

8.1.1 Styling choices

By default, choices are rendered by the *choice_html()* method. The result of this method will be used in the autocomplete box as well as in the *widget deck*. There are three easy ways to customize it:

- overriding `AutocompleteBase.choice_html_format`,
- overriding *AutocompleteBase.choice_html()*,
- or even with a template specified in *AutocompleteTemplate.choice_template*

Overriding `AutocompleteBase.choice_html_format`

The easiest and most limited way to change how a choice is rendered is to override the `AutocompleteBase.choice_html_format` attribute.

For example:

```
class OsAutocomplete(autocomplete_light.AutocompleteListBase):
    choices = ['Linux', 'BSD', 'Minix']
    choice_html_format = u'<span class="block os" data-value="%s">%s</span>'
```

This will add the class `os` to choices.

Overriding `AutocompleteBase.choice_html()`

Overriding *`AutocompleteBase.choice_html()`* enables changing the way choices are rendered.

For example:

```
class PersonAutocomplete(autocomplete_light.AutocompleteModelBase):
    choice_html_format = u'''
        <span class="block" data-value="%s"><img src="%s" /> %s</span>
    '''

    def choice_html(self, choice):
        return self.choice_html_format % (self.choice_value(choice),
            choice.profile_image.url, self.choice_label(choice))
```

Overriding `AutocompleteTemplate.choice_template`

Perhaps the coolest way to style choices is to use a template. Just set *`AutocompleteTemplate.choice_template`*. It is used by *`AutocompleteTemplate.choice_html`*:

```
class PersonAutocomplete(autocomplete_light.AutocompleteModelTemplate):
    choice_template = 'person_choice.html'
```

Now, all you have to do is create a `person_choice.html` template. Consider this elaborated example with image and links to the detail page and admin change form:

```
{% load i18n %}
{% load thumbnail %}

<span class="block person" data-value="{{ choice.pk }}">
    <img src="{% thumbnail choice.profile_image.url 50x50 crop %}" />

    <a href="{{ choice.get_absolute_url }}">
        {{ choice.first_name }} {{ choice.last_name }}
    </a>

    <a href="{% url 'admin:persons_person_change' choice.pk %}">
        {% trans 'Edit person' %}
    </a>

    {% if choice.company %}
    <a href="{{ choice.get_absolute_url }}">
        {{ choice.company }}
    </a>
    {% endif %}
</span>
```

First, the template loads the `i18n` template tags library which enables the `{% trans %}` template tag, useful for internationalization.

Then, it defines the tag, this element is valid anywhere even if your autocomplete widget is rendered in a <table>. However, this element has the block class which makes it display: block for space. Also, it adds the person class to enable specific CSS stylings. Finally it defines the data-value attribute. Note that **the "data-value" is critical** because it is what tells autocomplete.js that this element is a choice, and it also tells widget.js that the value is {{ choice.pk }} (which will be rendered before widget.js gets its hands on it of course).

8.1.2 Styling autocomplete boxes

By default, the autocomplete box is rendered by the *autocomplete_html()* method. The result of this method will be used to render the autocomplete box. There are many ways to customize it:

- overriding AutocompleteBase.autocomplete_html_format,
- overriding *AutocompleteBase.autocomplete_html()*,
- or even with a template specified in *AutocompleteTemplate.autocomplete_template* if using *AutocompleteTemplate* for rendering logic.

Overriding AutocompleteBase.autocomplete_html_format

The easiest and most limited way to change how a autocomplete is rendered is to override the AutocompleteBase.autocomplete_html_format attribute.

For example:

```
class OsAutocomplete(autocomplete_light.AutocompleteListBase):
    autocompletes = ['Linux', 'BSD', 'Minix']
    autocomplete_html_format = u'<span class="autocomplete-os">%s</span>'
```

This will add the autocomplete-os class to the autocomplete box.

Overriding AutocompleteBase.autocomplete_html

Overriding *AutocompleteBase.autocomplete_html()* enables changing the way autocompletes are rendered.

For example:

```
class PersonAutocomplete(autocomplete_light.AutocompleteModelBase):
    autocomplete_html_format = u'''
        <span class="autocomplete-os">
            <span class="count">%s Persons matching your query</span>
            %s
        </span>
    '''

    def autocomplete_html(self):
        html = ''.join(
            [self.choice_html(c) for c in self.choices_for_request()])

        if not html:
            html = self.empty_html_format % _('no matches found').capitalize()

        count = len(self.choices_for_request())
        return self.autocomplete_html_format % (count, html)
```

This will add a choice counter at the top of the autocomplete.

Overriding `AutocompleteTemplate.autocomplete_template`

Perhaps the coolest way to style an autocomplete box is to use a template. Just set *`AutocompleteTemplate.autocomplete_template`*. It is used by *`AutocompleteTemplate.autocomplete_html`*:

```
class PersonAutocomplete(autocomplete_light.AutocompleteModelTemplate):
    autocomplete_template = 'person_autocomplete.html'
```

Now, all you have to do is create a `person_autocomplete.html` template. Consider this elaborated example with user-friendly translated messages:

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

{% if choices %}
    <h2>{% trans 'Please select a person' %}</h2>
    {% for choice in choices %}
        {{ choice|autocomplete_light_choice_html:autocomplete }}
    {% endfor %}
{% else %}
    <h2>{% trans 'No matching person found' %}</h2>
    <p>
        {% blocktrans %}Sometimes, persons have not filled their name,
        maybe try to search based on email addresses ?{% endblocktrans %}
    </p>
{% endif %}
```

First, it loads Django's i18n template tags for translation. Then, it loads autocomplete-light's tags.

If there are any choices, it will display the list of choices, rendered by `choice_html()` through the `autocomplete_light_choice_html` template filter as such: `{{ choice|autocomplete_light_choice_html:autocomplete }}`.

If no choice is found, then it will display a user friendly suggestion.

8.1.3 Styling widgets

Widgets are rendered by the `render()` method. By default, it renders *autocomplete_light/widget.html*. While you can override the widget template globally, there are two ways to override the widget template name on a per-case basis:

- *`WidgetBase.widget_template`*,
- `AutocompleteBase.widget_template`,

Using another template instead of a global override allows to extend the default widget template and override only the parts you need.

If you're not sure what is in a widget template, please review *part 2 of reference documentation about widget templates*.

Also, note that the widget is styled with CSS, you can override or extend any definition of `autocomplete_light/style.css`.

AutocompleteModelTemplate

By default, `AutocompleteModelTemplate` sets `choice_template` to `autocomplete_light/model_template/choice.html`. It adds a "view absolute url" link as well as an "update form url" link based on `YourModel.get_absolute_url()` and `YourModel.get_absolute_update_url()` with such a template:

```
{% load i18n l10n %}
{% load static %}

{% spaceless %}
<span class="block" data-value="{{ choice.pk|unlocalize }}">
    {{ choice }}
    {% with choice.get_absolute_url as url %}
        {% if url %}
        <a href="{{ url }}" target="_blank" class="choice-detail">&rarr;</a>
        {% endif %}
    {% endwith %}

    {% with choice.get_absolute_update_url  as url %}
        {% if url %}
        <a href="{{ url }}" target="_blank" class="choice-update">
            <img src="{% static 'admin/img/icon_changelink.gif' %}" />
        </a>
        {% endif %}
    {% endwith %}
</span>
{% endspaceless %}
```

It does not play well in all projects, so it was not set as default. But you can inherit from it:

```
class YourAutocomplete(autocomplete_light.AutocompleteModelTemplate):
    model = YourModel
autocomplete_light.register(YourAutocomplete)
```

Or let the *`register()`* shortcut use it:

```
autocomplete_light.register(YourModel,
    autocomplete_light.AutocompleteModelTemplate)
```

Or set it as default with *`AutocompleteRegistry.autocomplete_model_base`* and used it as such:

```
autocomplete_light.register(YourModel)
```

8.2 Making a global navigation autocomplete

This guide demonstrates how to make a global navigation autocomplete like on Facebook.

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

8.2.1 A simple view

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

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

from autocomplete_light.example_apps.music.models import Artist, Genre

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

    q = request.GET.get('q', '')

    queries = {}

    queries['artists'] = Artist.objects.filter(
        Q(name__icontains=q) |
        Q(genre__name__icontains=q)
    ).distinct()[:6]

    queries['genres'] = Genre.objects.filter(name__icontains=q)[:6]

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

Along with a trivial template for `navigation_autocomplete/autocomplete.html` would work:

```
<span class="separator">Artists</span>
{% for artist in artists %}
<a class="block choice" href="{{ artist.get_absolute_url }}">{{ artist }}</a>
{% endfor %}

<span class="separator">Genres</span>
{% for genre in genre %}
<a class="block choice" href="{{ genre.get_absolute_url }}">{{ genre }}</a>
{% endfor %}
```

8.2.2 A basic autocomplete configuration

That's a pretty basic usage of *autocomplete.js*, concepts are detailed in *Using $.yourlabsAutocomplete to create a navigation autocomplete*, this is what it looks like:

```
// Change #yourInput by a selector that matches the input you want to use
// for the navigation autocomplete.
$('#yourInput').yourlabsAutocomplete({
    // Url of the view you just created
    url: '{% url "your_autocomplete_url" %}',

    // With keyboard, we should iterate around <a> tags in the autocomplete
    choiceSelector: 'a',
}).input.bind('selectChoice', function(e, choice, autocomplete) {
    // When a choice is selected, open it. Note: this is not needed for
    // mouse click on the links of course, but this is necessary for keyboard
    // selection.
    window.location.href = choice.attr('href');
});
```

8.3 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 autocomplete_light/example_apps/dependant_autocomplete/.

8.3.1 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 '%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.

8.3.2 Autocompletes

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

```
import autocomplete_light.shortcuts as 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)
```

8.3.3 Javascript

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

```
import autocomplete_light.shortcuts as autocomplete_light
from django import forms

from .models import Dummy


class DummyForm(autocomplete_light.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
        exclude = []
```

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 but can hel
        // console.log($(this), 'changed to', value);
        // console.log(regionWidgetElement, 'data is', regionWidgetElement.
↪yourlabsWidget().autocomplete.data)
    })
});
```

8.3.4 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.

8.4 Generic relations

First, you need to register an autocomplete class for autocompletes on generic relations.

The easiest is to inherit from `AutocompleteGenericBase` or `AutocompleteGenericTemplate`. The main logic is contained in *`AutocompleteGeneric`* which is extended by both the Base and Template versions.

Generic relation support comes in two flavors:

- for django's generic foreign keys,
- and for django-generic-m2m's generic many to many.

autocomplete_light.ModelForm will setup the fields:

- *`autocomplete_light.GenericModelChoiceField`*, and
- *`autocomplete_light.GenericModelMultipleChoiceField`*.

Those fields will use the default generic autocomplete class, which is the last one you register as generic. If you want to use several generic autocomplete classes, then you should setup the fields yourself to specify the autocomplete name as such:

```
class YourModelForm(autocomplete_light.ModelForm):
    # if your GenericForeignKey name is "generic_fk":
    generic_fk = autocomplete_light.GenericModelChoiceField('YourAutocomplete1')

    # if your RelatedObjectsDescriptor is "generic_m2m":
    generic_m2m = autocomplete_light.GenericModelMultipleChoiceField(
↪'YourAutocomplete2')
```

But please note that you will *loose some DRY* by doing that, as stated in the faq.

8.4.1 Example using `AutocompleteGenericBase`

This example demonstrates how to setup a generic autocomplete with 4 models:

```
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)
```

8.5 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

FAQ

9.1 Can't see admin add-another + button when overriding a `ModelChoiceField`

It's common for users to report that the +/add-another button disappears when using a `ModelForm` with an overriden `ModelChoiceField`. This is actually a Django issue.

As a workaround, 2.0.6 allows using `Autocomplete.add_another_url_name` in the admin, ie.:

```
autocomplete_light.register(YourModel, add_another_url_name='admin:yourapp_yourmodel_
↪add')
```

While using *Add another popup outside the admin* has been supported for years, support for using it as a workaround in Django admin is currently experimental. You can obviously imagine the problem if such an Autocomplete is used for a user which has no access to the admin: the plus button will fail.

Your input is very welcome on this matter.

9.2 RemovedInDjango18Warning: Creating a ModelForm without either the 'fields' attribute or the 'exclude' attribute is deprecated - form YourForm needs updating

It's very common for users who are not actively following Django 1.7 development and Django security matters (even though they should!) to report the following warning as a problem with django-autocomplete-light:

```
autocomplete_light/forms.py:266: RemovedInDjango18Warning: Creating a ModelForm␣
↪without either the 'fields' attribute or the 'exclude' attribute is deprecated -␣
↪form YourForm needs updating
```

This is a **new security feature from Django**, and has **nothing to do with django-autocomplete-light**. As the message clearly states, it is **deprecated to create a ModelForm without either the 'fields' attribute or the 'exclude' attribute**.

Solution: pass `fields = '__all__'`.

See Django documentation on "Selecting the fields to use" for details.

9.3 How to run tests

You should not try to `test autocomplete_light` from your own project because tests depend on example apps to be present in `INSTALLED_APPS`. You may use the provided `test_project` which is prepared to run all testst.

Install a version from git, ie:

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

From there you have two choices:

- either go in `env/src/autocomplete_light/test_project` and run `./manage.py test autocomplete_light`,
- either go in `env/src/autocomplete_light/` and run `tox` after installing it from pip.

If you're trying to run a buildbot then you can use `test.sh` and use that buildbot configuration to enable CI on the 28 supported configurations:

```
def make_build(python, django, genericm2m, taggit):
    name = 'py%s-dj%s' % (python, django)

    if genericm2m != '0':
        name += '-genericm2m'
    if taggit != '0':
        name += '-taggit'

    slavenames = ['example-slave']
    if python == '2.7':
        slavenames.append('gina')

    factory = BuildFactory()
    # check out the source
    factory.addStep(Git(repourl='https://github.com/yourlabs/django-autocomplete-
↪light.git', mode='incremental'))
    # run the tests (note that this will require that 'trial' is installed)
    factory.addStep(ShellCommand(command=["./test.sh"], timeout=3600))

    c['builders'].append(
        BuilderConfig(name=name,
            slavenames=slavenames,
            factory=factory,
            env={
                'DJANGO_VERSION': django,
                'PYTHON_VERSION': python,
                'DJANGO_GENERIC_M2M': genericm2m,
                'DJANGO_TAGGIT': taggit,
            }
        )
```

```
    )

    c['schedulers'].append(SingleBranchScheduler(
                          name="all-%s" % name,
                          change_filter=filter.ChangeFilter(branch='v2'),
                          treeStableTimer=None,
                          builderNames=[name]))
    c['schedulers'].append(ForceScheduler(
                          name="force-%s" % name,
                          builderNames=[name]))


c['builders'] = []
djangos = ['1.4', '1.5', '1.6']
pythons = ['2.7', '3.3']

for python in pythons:
    for django in djangos:
        if python == '3.3' and django == '1.4':
            continue

        for genericm2m in ['0','1']:
            for taggit in ['0','1']:
                make_build(python, django, genericm2m, taggit)
```

9.4 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 wanted to load jQuery and autocomplete.js globally **anyway** and **anywhere** in the admin to have a global navigation autocomplete: it would load the scripts twice.

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.

9.5 Model field's `help_text` and `verbose_name` are lost when overriding the widget

This has nothing to do with django-autocomplete-light, but still it's a FAQ so here goes.

When Django's ModelForm creates a form field for a model field, it copies `models.Field.verbose_name` to `forms.Field.label` and `models.Field.help_text` to `forms.Field.help_text`, as uses `models.Field.blank` to create `forms.Field.required`.

For example:

```
class Person(models.Model):
    name = models.CharField(
        max_length=100,
        blank=True,
        verbose_name='Person name',
        help_text='Please fill in the complete person name'
    )

class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
```

Thanks to Django's DRY system, this is equivalent to:

```
class PersonForm(forms.ModelForm):
    name = forms.CharField(
        max_length=100,
        required=False,
        label='Person name',
        help_text='Please fill in the complete person name'
    )

    class Meta:
        model = Person
```

But you will loose that logic as soon as you decide to override Django's generated form field with your own. So if you do this:

```
class PersonForm(forms.ModelForm):
    name = forms.CharField(widget=YourWidget)

    class Meta:
        model = Person
```

Then you loose Django's DRY system, because **you** instanciate the name form field, so Django leaves it as is.

If you want to override the widget of a form field and you **don't** want to override the form field, then you should refer to Django's documentation on overriding the default fields which means you should use `Meta.widgets`, ie.:

```
class PersonForm(forms.ModelForm):
    class Meta:
        model = Person
        widgets = {'name': YourWidget}
```

Again, this has nothing to do with django-autocomplete-light.

9.6 Fields bound on values which are not in the queryset anymore raise a ValidationError

This is not specific to django-autocomplete-light, but still it's a FAQ so here goes.

Django **specifies in its unit tests** that a `ModelChoiceField` and `ModelMultipleChoiceField` should raise a `ValidationError` if a value is not part of the `queryset` passed to the field constructor.

This is the relevant part of Django's specification:

```
# Delete a Category object *after* the ModelChoiceField has already been
# instantiated. This proves clean() checks the database during clean() rather
# than caching it at time of instantiation.
Category.objects.get(url='5th').delete()
with self.assertRaises(ValidationError):
    f.clean(c5.id)

# [...]

# Delete a Category object *after* the ModelMultipleChoiceField has already been
# instantiated. This proves clean() checks the database during clean() rather
# than caching it at time of instantiation.
Category.objects.get(url='6th').delete()
with self.assertRaises(ValidationError):
    f.clean([c6.id])
```

django-autocomplete-light behaves exactly the same way. If an item is removed from the queryset, then its value will be dropped from the field values on display of the form. Trying to save that value again will raise a ValidationError will be raised, just like if the item wasn't there at all.

But don't take my word for it, try the `security_test` app of the `test_project`, it provides:

- an admin to control which items are in and out of the queryset,
- an update view with a django select
- another update view with an autocomplete instead

9.7 How to override a JS method ?

Refer to *Override autocomplete JS methods*.

9.8 How to work around Django bug #9321: *Hold down "Control" . . .* ?

Just use the `autocomplete_light.ModelForm` or inherit from both `SelectMultipleHelpTextRemovalMixin` and `django.forms.ModelForm`.

9.9 How to report a bug effectively ?

Read How to Report Bugs Effectively and open an issue on django-autocomplete-light's issue tracker on GitHub.

9.10 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: find hidden gems

10.1 Registry API

class autocomplete_light.registry.**AutocompleteRegistry**(*autocomplete_model_base=None*)

AutocompleteRegistry is a dict of `AutocompleteName: AutocompleteClass` with some shortcuts to handle a registry of autocompletes.

autocomplete_model_base

The default model autocomplete class to use when registering a Model without Autocomplete class. Default is `AutocompleteModelBase`. You can override it just before calling autodiscover() in urls.py as such:

```
import autocomplete_light.shortcuts as al
al.registry.autocomplete_model_base = al.AutocompleteModelTemplate
al.autodiscover()
```

You can pass a custom base autocomplete which will be set to *autocomplete_model_base* when instanciating an AutocompleteRegistry.

autocomplete_for_generic()

Return the default generic autocomplete.

autocomplete_for_model(*model*)

Return the default autocomplete class for a given model or None.

classmethod extract_args(**args*)

Takes any arguments like a model and an autocomplete, or just one of those, in any order, and return a model and autocomplete.

register(**args*, ***kwargs*)

Register an autocomplete.

Two unordered arguments are accepted, at least one should be passed:

- a model if not a generic autocomplete,
- an autocomplete class if necessary, else one will be generated.

'name' is also an acceptable keyword argument, that can be used to override the default autocomplete name which is the class name by default, which could cause name conflicts in some rare cases.

In addition, keyword arguments will be set as class attributes.

For thread safety reasons, a copy of the autocomplete class is stored in the registry.

unregister(*name*)
Unregister a autocomplete given a name.

autocomplete_light.registry.**register**(**args*, ***kwargs*)
Proxy method *AutocompleteRegistry.register()* of the registry module level instance.

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

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

Consider a standard app called cities_light with such a structure:

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

Where autocomplete_light_registry.py contains something like:

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

When autodiscover() imports cities_light.autocomplete_light_registry, both CityAutocomplete and CountryAutocomplete will be registered. See *AutocompleteRegistry.register()* for details on how these autocomplete classes are generated.

10.2 Autocomplete class API

10.2.1 AutocompleteInterface

class autocomplete_light.autocomplete.base.**AutocompleteInterface**(*request=None*, *values=None*)
An autocomplete proposes "choices". A choice has a "value". When the user selects a "choice", then it is converted to a "value".

AutocompleteInterface is the minimum to implement in a custom Autocomplete class usable by the widget and the view. It has two attributes:

values
A list of values which *validate_values()* and *choices_for_values()* should use.

request
A request object which *autocomplete_html()* should use.

It is recommended that you inherit from *AutocompleteBase* instead when making your own classes because it has taken some design decisions favorising a DRY implementation of *AutocompleteInterface*.

Instanciate an Autocomplete with a given `request` and `values` arguments. `values` will be casted to list if necessary and both will be assigned to instance attributes *request* and *values* respectively.

autocomplete_html()
Return the HTML autocomplete that should be displayed under the text input. *request* can be used, if set.

choices_for_values()
Return the list of choices corresponding to *values*.

get_absolute_url()
Return the absolute url for this autocomplete, using autocomplete_light_autocomplete url.

validate_values()
Return True if *values* are all valid.

10.2.2 Rendering logic Autocomplete mixins

AutocompleteBase

class autocomplete_light.autocomplete.base.**AutocompleteBase**(*request=None*, *values=None*)
A basic implementation of AutocompleteInterface that renders HTML and should fit most cases. It only needs overload of *choices_for_request()* and *choices_for_values()* which is the business-logic.

choice_html_format
HTML string used to format a python choice in HTML by *choice_html()*. It is formated with two positionnal parameters: the value and the html representation, respectively generated by *choice_value()* and *choice_label()*. Default is:

```
<span data-value="%s">%s</span>
```

empty_html_format
HTML string used to format the message "no matches found" if no choices match the current request. It takes a parameter for the translated message. Default is:

```
<span class="block"><em>%s</em></span>
```

autocomplete_html_format
HTML string used to format the list of HTML choices. It takes a positionnal parameter which contains the list of HTML choices which come from *choice_html()*. Default is:

```
%s
```

add_another_url_name
Name of the url to add another choice via a javascript popup. If empty then no "add another" link will appear.

add_another_url_kwargs
Keyword arguments to use when reversing the add another url.

widget_template
A special attribute used only by the widget. If it is set, the widget will use that instead of the default `autocomplete_light/widget.html`.

autocomplete_html()
Simple rendering of the autocomplete.

It will append the result of *choice_html()* for each choice returned by *choices_for_request()*, and wrap that in *autocomplete_html_format*.

choice_html(*choice*)
Format a choice using *choice_html_format*.

choice_label(*choice*)
Return the human-readable representation of a choice. This simple implementation returns the textual representation.

choice_value(*choice*)
Return the value of a choice. This simple implementation returns the textual representation.

choices_for_request()
Return the list of choices that are available. Uses `request` if set, this method is used by *autocomplete_html()*.

get_add_another_url()
Return the url to use when adding another element

validate_values()
This basic implementation returns True if all *values* are in *choices_for_values()*.

AutocompleteTemplate

class autocomplete_light.autocomplete.template.**AutocompleteTemplate**(*request=None*, *values=None*)
This extension of *AutocompleteBase* supports two new attributes:

choice_template
Name of the template to use to render a choice in the autocomplete. If none is specified, then *AutocompleteBase* will render the choice.

autocomplete_template
Name of the template to use to render the autocomplete. Again, fall back on *AutocompleteBase* if this is None.

autocomplete_html()
Render *autocomplete_template* with base context and `{{ choices }}`. If *autocomplete_template* is None then fall back on *base.AutocompleteBase.autocomplete_html()*.

choice_html(*choice*)
Render *choice_template* with base context and `{{ choice }}`. If *choice_template* is None then fall back on *base.AutocompleteBase.choice_html()*.

get_base_context()
Return a dict to use as base context for all templates.

It contains:

- `{{ request }}` if available,
- `{{ autocomplete }}` the "self" instance.

render_template_context(*template*, *extra_context=None*)
Render `template` with base context and `extra_context`.

10.2.3 Business logic Autocomplete mixins

AutocompleteList

class autocomplete_light.autocomplete.list.**AutocompleteList**

Simple Autocomplete implementation which expects *choices* to be a list of string choices.

choices
List of string choices.

limit_choices
The maximum of items to suggest from *choices*.

order_by
order_choices() will use this against *choices* as an argument sorted().

It was mainly used as a starter for me when doing test-driven development and to ensure that the Autocomplete pattern would be concretely simple and yet powerful.

choices_for_request()
Return any *choices* that contains the search string. It is case insensitive and ignores spaces.

choices_for_values()
Return any *choices* that is in values.

order_choices(*choices*)
Run sorted() against choices and *order_by*.

AutocompleteChoiceList

class autocomplete_light.autocomplete.choice_list.**AutocompleteChoiceList**

Simple *AutocompleteList* implementation which expects *choices* to be a list of tuple choices in the fashion of django.db.models.Field.choices.

choices
List of choice tuples (value, label) like django.db.models.Field.choices. Example:

```
choices = (
    ('v', 'Video'),
    ('p', 'Paper'),
)
```

limit_choices
The maximum of items to suggest from *choices*.

order_by
order_choices() will use this against *choices* as an argument sorted().

choice_label(*choice*)
Return item 1 of the choice tuple.

choice_value(*choice*)
Return item 0 of the choice tuple.

choices_for_request()
Return any *choices* tuple that contains the search string. It is case insensitive and ignores spaces.

choices_for_values()
Return any *choices* that is in values.

AutocompleteModel

class autocomplete_light.autocomplete.model.**AutocompleteModel**
Autocomplete which considers choices as a queryset.

choices
A queryset.

limit_choices
Maximum number of choices to display.

search_fields
Fields to search in, configurable like on `django.contrib.admin.ModelAdmin.search_fields`

split_words
If True, AutocompleteModel splits the search query into words and returns all objects that contain each of the words, case insensitive, where each word must be in at least one of search_fields. This mimics the mechanism of django's `django.contrib.admin.ModelAdmin.search_fields`.

If 'or', AutocompleteModel does the same but returns all objects that contain **any** of the words.

order_by
If set, it will be used to order choices in the deck. It can be a single field name or an iterable (ie. list, tuple). However, if AutocompleteModel is instanciated with a list of values, it'll reproduce the ordering of values.

choice_label(*choice*)
Return the textual representation of the choice by default.

choice_value(*choice*)
Return the pk of the choice by default.

choices_for_request()
Return a queryset based on *choices* using options *split_words*, *search_fields* and *limit_choices*.

choices_for_values()
Return ordered choices which pk are in *values*.

order_choices(*choices*)
Order choices using *order_by* option if it is set.

validate_values()
Return True if all values where found in *choices*.

AutocompleteGeneric

class autocomplete_light.autocomplete.generic.**AutocompleteGeneric**
AutocompleteModel extension which considers choices as a **list of querysets**, and composes a choice value with both the content type pk and the actual model pk.

choices
A list of querysets. Example:

```
choices = (
    User.objects.all(),
    Group.objects.all(),
)
```

search_fields
A list of lists of fields to search in, configurable like on ModelAdmin.search_fields. The first list of fields will be used for the first queryset in choices and so on. Example:

```
search_fields = (
    ('email', '^name'),  # Used for User.objects.all()
    ('name',)            # User for Group.objects.all()
)
```

AutocompleteGeneric inherits from *model.AutocompleteModel* and supports *limit_choices* and *split_words* exactly like AutocompleteModel.

However *order_by* is not supported (yet) in AutocompleteGeneric.

choice_value(*choice*)
Rely on GenericModelChoiceField to return a string containing the content type id and object id of the result.

choices_for_request()
Return a list of choices from every queryset in *choices*.

choices_for_values()
Values which are not found in any querysets of *choices* are ignored.

validate_values()
Ensure that every choice is part of a queryset in *choices*.

10.2.4 Autocomplete classes with both rendering and business logic

10.2.5 Views

class autocomplete_light.views.**AutocompleteView**(***kwargs*)
Simple view that routes the request to the appropriate autocomplete.

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

get(*request*, **args*, ***kwargs*)
Return an HttpResponse with the return value of autocomplete.autocomplete_html().

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

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

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

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

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

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

class autocomplete_light.views.**CreateView**(***kwargs*)
Simple wrapper for generic.CreateView, that responds to _popup.

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

form_valid(*form*)
If request.GET._popup, return some javascript.

10.3 Form, fields and widgets API

10.3.1 Widgets

WidgetBase

class autocomplete_light.widgets.**WidgetBase**(*autocomplete=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, *registry=None*, *widget_template=None*, *widget_attrs=None*)

Base widget for autocompletes.

attrs
HTML `<input />` attributes, such as class, placeholder, etc ... Note that any `data-autocomplete-*` attribute will be parsed as an option for `yourlabs.Autocomplete` js object. For example:

attrs={ 'placeholder': 'foo', 'data-autocomplete-minimum-characters': 0 'class': 'bar',

}

Will render like::

<input placeholder="foo" data-autocomplete-minimum-characters="0" class="autocomplete bar"

/>

Which will set by the way `yourlabs.Autocomplete.minimumCharacters` option - the naming conversion is handled by jQuery.

widget_attrs
HTML widget container attributes. Note that any `data-widget-*` attribute will be parsed as an option for `yourlabs.Widget` js object. For example:

```
widget_attrs={
    'data-widget-maximum-values': 6,
    'class': 'country-autocomplete',
}
```

Will render like:

```
<span
    id="country-wrapper"
    data-widget-maximum-values="6"
    class="country-autocomplete autcomplete-light-widget"
/>
```

Which will set by the way `yourlabs.Widget.maximumValues` - note that the naming conversion is handled by jQuery.

widget_js_attributes
DEPRECATED in favor of :py:attr::*widget_attrs*.

A dict of options that will override the default widget options. For example:

```
widget_js_attributes = {'max_values': 8}
```

The above code will set this HTML attribute:

```
data-max-values="8"
```

Which will override the default javascript widget maxValues option (which is 0).

It is important to understand naming conventions which are sparse unfortunately:

- python: lower case with underscores ie. `max_values`,
- HTML attributes: lower case with dashes ie. `data-max-values`,
- javascript: camel case, ie. `maxValues`.

The python to HTML name conversion is done by the autocomplete_light_data_attributes template filter.

The HTML to javascript name conversion is done by the jquery plugin.

autocomplete_js_attributes
DEPRECATED in favor of :py:attr::*attrs*.

A dict of options like for *widget_js_attributes*. However, note that HTML attributes will be prefixed by `data-autocomplete-` instead of just `data-`. This allows the jQuery plugins to make the distinction between attributes for the autocomplete instance and attributes for the widget instance.

extra_context
Extra context dict to pass to the template.

widget_template
Template to use to render the widget. Default is `autocomplete_light/widget.html`.

ChoiceWidget

class autocomplete_light.widgets.**ChoiceWidget**(*autocomplete=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, *registry=None*, *widget_template=None*, *widget_attrs=None*, **args*, ***kwargs*)

Widget that provides an autocomplete for zero to one choice.

MultipleChoiceWidget

class autocomplete_light.widgets.**MultipleChoiceWidget**(*autocomplete=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, *registry=None*, *widget_template=None*, *widget_attrs=None*, **args*, ***kwargs*)

Widget that provides an autocomplete for zero to n choices.

TextWidget

class autocomplete_light.widgets.**TextWidget**(*autocomplete=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, *registry=None*, *widget_template=None*, *widget_attrs=None*, **args*, ***kwargs*)

Widget that just adds an autocomplete to fill a text input.

Note that it only renders an `<input>`, so attrs and widget_attrs are merged together.

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

Proxy Django's TextInput.render()

10.3.2 Fields

FieldBase

class autocomplete_light.fields.**FieldBase**(*autocomplete=None*, *registry=None*, *widget=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, **args*, ***kwargs*)

ChoiceField

class autocomplete_light.fields.**ChoiceField**(*autocomplete=None*, *registry=None*, *widget=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, **args*, ***kwargs*)

widget

alias of `ChoiceWidget`

MultipleChoiceField

class autocomplete_light.fields.**MultipleChoiceField**(*autocomplete=None*, *registry=None*, *widget=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, **args*, ***kwargs*)

widget

alias of `MultipleChoiceWidget`

ModelChoiceField

class autocomplete_light.fields.**ModelChoiceField**(*autocomplete=None*, *registry=None*, *widget=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, **args*, ***kwargs*)

widget
alias of `ChoiceWidget`

ModelMultipleChoiceField

class autocomplete_light.fields.**ModelMultipleChoiceField**(*autocomplete=None*, *registry=None*, *widget=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, **args*, ***kwargs*)

widget
alias of `MultipleChoiceWidget`

GenericModelChoiceField

class autocomplete_light.fields.**GenericModelChoiceField**(*autocomplete=None*, *registry=None*, *widget=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, **args*, ***kwargs*)

Simple form field that converts strings to models.

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

to_python(*value*)
Given a string like '3-5', return the model of content type id 3 and pk 5.

widget
alias of `ChoiceWidget`

GenericModelMultipleChoiceField

class autocomplete_light.fields.**GenericModelMultipleChoiceField**(*autocomplete=None*, *registry=None*, *widget=None*, *widget_js_attributes=None*, *autocomplete_js_attributes=None*, *extra_context=None*, **args*, ***kwargs*)

Simple form field that converts strings to models.

widget
alias of `MultipleChoiceWidget`

10.3.3 Form stuff

modelform_factory

ModelForm

ModelFormMetaclass

SelectMultipleHelpTextRemovalMixin

VirtualFieldHandlingMixin

GenericM2MRelatedObjectDescriptorHandlingMixin

FormfieldCallback

ModelFormMetaclass

10.4 Script API

10.4.1 `autocomplete.js`

The autocomplete box script, see autocomplete.js API documentation.

10.4.2 `widget.js`

The script that ties the autocomplete box script and the hidden `<select>` used by django, see widget.js API documentation.

10.4.3 `text_widget.js`

The script that ties the autocomplete box script with a text input, see text_widget.js API documentation.

10.4.4 addanother.js

The script that enables adding options to a <select> outside the admin, see addanother.js API documentation.

10.4.5 remote.js

The script that overrides a method from widget.js to create choices on the fly, see remote.js API documentation.

CHAPTER 11

Upgrade

Any change is documented in the changelog, so upgrading from a version to another is always documented there. Usualy, upgrade from pip with a command like `pip install -U django-autocomplete-light`. Check the CHANGELOG for BC (Backward Compatibility) breaks. There should is no backward compatibility for minor version upgrades ie. from 1.1.3 to 1.1.22, but there *might* be some minor BC breaks for middle upgrades ie. 1.2.0 to 1.3.0.

11.1 v1 to v2

There are major changes between v1 and v2, upgrading has been extensively documented:

11.1.1 Upgrading from django-autocomplete-light v1 to v2

Please enjoy this v1 to v2 upgrade instructions to upgrade from DAL 1.x to 2.x (documented with love !).

Quick upgrade

- the Autocomplete class design hasn't changed at all.
- `yourlabsWidget()` doesn't parses `data-*` options the same,
- the django/form python code has been re-organised ie. `get_widgets_dict()` is gone and `autocomplete_light.ModelForm` wraps around all features by default.
- use `autocomplete_light.ModelForm` instead of `autocomplete_light.GenericModelForm` - generic foreign keys and django-generic-m2m are supported by default if installed.

Detailed upgrade

You should not use widget directly anymore

We used to have things like this:

```
class YourForm(autocomplete_light.GenericModelForm):
    user = forms.ModelChoiceField(User.objects.all(),
        widget=autocomplete_light.ChoiceWidget('UserAutocomplete'))

    related = GenericModelChoiceField(
            widget=autocomplete_light.ChoiceWidget(
                autocomplete='AutocompleteTaggableItems',
                autocomplete_js_attributes={'minimum_characters': 0}))

    class Meta:
        model = YourModel
```

This caused several problems:

- broke a DRY principle: if you have defined a `user` foreign key **and** registered an Autocomplete for the model in question, `User`, then you should not have to repeat this.
- broke the DRY principle since you had to set choices on both the ModelChoiceField and the Autocomplete - `UserAutocomplete` in this example.
- also, validation was done in the widget's `render()` function, mostly for security reasons. Validation is not done in the widget anymore, instead it is done in *autocomplete_light.fields*.

What should the above code be like ? Well it depends, it could just be:

```
class YourForm(autocomplete_light.ModelForm):
    class Meta:
        model = YourModel
```

If you have registered an Autocomplete for the model that the `user` ForeignKey is for, then `autocomplete_light.ModelForm` will pick it up automatically.

Assuming you have registered a generic autocomplete, `autocomplete_light.ModelForm` will pick it up automatically.

If you want Django's default behavior back (using a `<select>` tag), then you could tell `autocomplete_light.ModelForm` to not be "autocomplete-aware" for `user` as such:

```
class YourForm(autocomplete_light.ModelForm):
    class Meta:
        model = YourModel
        autocomplete_exclude = ('user',)
```

autocomplete_light.ModelChoiceField and *autocomplete_light.GenericModelChoiceField*:

```
class YourForm(autocomplete_light.ModelForm):
    user = autocomplete_light.ModelChoiceField('UserAutocomplete')
    related = autocomplete_light.GenericModelChoiceField('AutocompleteTaggableItems')

    class Meta:
        model = YourModel
        autocomplete_exclude = ('user',)
```

You can still override widgets the same way as before, but you should consider the *DRY breaking* implications (which are not specific to django-autocomplete-light, but Django's design in general).

Specification of the Autocomplete class to use

New rules are:

- if an Autocomplete class was registered for a model then it becomes the default Autocomplete class for autocompletion on that model,
- other Autocomplete classes registered for a model will not be used by default

You can still define the Autocomplete class you want in the field definition:

```
class FooForm(autocomplete_light.ModelForm):
    bar = autocomplete_light.ModelChoiceField('SpecialBarAutocomplete')

    class Meta:
        model = Foo
```

But this will break some *break django DRY logic*. Instead, this won't break DRY:

```
class FooForm(autocomplete_light.ModelForm):
    class Meta:
        model = Foo
        autocomplete_names = {'bar': 'SpecialBarAutocomplete'}
```

Python class re-organisation

Form classes like `FixedModelform` or `GenericModelForm` were renamed. But if you can, just inherit from `autocomplete_light.ModelForm` instead.

Generic field classes were moved from `autocomplete_light.contrib.generic_m2m` into `autocomplete_light.fields`: just import *`autocomplete_light.GenericModelChoiceField`* and `autocomplete_light.GenericModelMultipleChoiceField <autocomplete_light.fields.GenericModelMultipleChoiceField`.

Deprecation of `autocomplete_js_attributes` and `widget_js_attributes`

In the past, we used `autocomplete_js_attributes` and `widget_js_attributes`. Those are deprecated and HTML `data` attributes should be used instead.

For example:

```
class PersonAutocomplete(autocomplete_light.AutocompleteModelBase):
    model = Person
    autocomplete_js_attributes = {
        'minimum_characters': 0,
        'placeholder': 'foo',
    }
    widget_js_attributes = {'max_values': 3}
```

Should now be:

```
class PersonAutocomplete(autocomplete_light.AutocompleteModelBase):
    model = Person
    attrs = {
        'data-autcomplete-minimum-characters': 0,
        'placeholder': 'foo',
    }
    widget_attrs = {'data-widget-maximum-values': 3}
```

As you probably understand already magic inside `autocomplete_js_attributes` and `widget_js_attributes` is gone, we're just setting plain simple HTML attributes now with `attrs`.

Also notice the other two differences which are detailed below:

- `max-values` was renamed to `maximum-values` (see below)
- `data-autocomplete-placeholder` is gone in favor of HTML5 `placeholder` attribute (see below)

`max-values` was renamed to `maximum-values`

For consistency with one of my naming conventions which is: no abbreviations.

`data-autocomplete-placeholder` is gone in favor of HTML5 `placeholder` attribute

It made no sense to keep `data-autocomplete-placeholder` since we now have the HTML5 `placeholder` attribute.

Widget template changes

This is a side effect from the deprecation of `autocomplete_js_attributes` and `widget_js_attributes`.

This:

```
<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-' }}
    >
```

Is now:

```
<span id="{{ widget.html_id }}-wrapper" {{ widget_attrs }} >
```

Script changes

`.yourlabsWidget()` used to parse `data-*` attributes:

- `data-foo-bar` used to set the JS attribute `yourlabs.Widget.fooBar`,
- `data-autocomplete-foo-bar` used to set the JS attribute `yourlabs.Widget.autocomplete.fooBar`.

Now:

- `.yourlabsWidget()` parses `data-widget-*` attributes and,
- `.yourlabsAutocomplete()` parses `data-autocomplete-*` **on the ‘‘<input />‘‘** !

So this:

```
<span class="autocomplete-light-widget" data-autocomplete-foo-bar="2" data-foo-bar="3
↪">
    <input .. />
```

Becomes:

```
<span class="autocomplete-light-widget" data-widget-foo-bar="3">
    <input data-autocomplete-foo-bar="2" ... />
```

`.choiceDetail` and `.choiceUpdate` were renamed to `.choice-detail` and `.choice-update`

This makes the CSS class names standard.

CHAPTER 12

Documentation that has not yet been ported to v2

12.1 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.

12.1.1 Example

This demonstrates a working usage of TextWidget:

```
import autocomplete_light
from django import forms
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:

```
import autocomplete_light
from tagging.models import Tag

autocomplete_light.register(Tag)
```

12.1.2 Django-tagging

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

```
import tagging
from django.db import models
from tagging.fields import TagField


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')
```

12.1.3 Django-taggit

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

12.2 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.

12.2.1 Specifications

Consider such a model:

```
from __future__ import unicode_literals

from django.core import urlresolvers
from django.db import models
from django.utils.encoding import python_2_unicode_compatible


@python_2_unicode_compatible
class NonAdminAddAnotherModel(models.Model):
    name = models.CharField(max_length=100)
    widgets = models.ManyToManyField('self', blank=True)

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

    def __str__(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.

12.2.2 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:

```
import autocomplete_light.shortcuts as al

from autocomplete_light.compat import url, urls
from django.views import generic

from .forms import NonAdminAddAnotherModelForm
from .models import NonAdminAddAnotherModel

urlpatterns = urls([
    url(r'$', al.CreateView.as_view(
        model=NonAdminAddAnotherModel, form_class=NonAdminAddAnotherModelForm),
        name='non_admin_add_another_model_create'),
    url(r'(?P<pk>\d+)/$', generic.UpdateView.as_view(
        model=NonAdminAddAnotherModel, form_class=NonAdminAddAnotherModelForm),
        name='non_admin_add_another_model_update'),
])
```

Note: It is not mandatory to use url namespaces.

12.2.3 Autocompletes

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

```
import autocomplete_light.shortcuts as autocomplete_light

from .models import NonAdminAddAnotherModel

autocomplete_light.register(NonAdminAddAnotherModel,
    add_another_url_name='non_admin_add_another_model_create')
```

12.3 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.

12.3.1 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.

12.3.2 API

class autocomplete_light.autocomplete.rest_model.**AutocompleteRestModel**

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

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

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

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

download_choice(*choice*)
Take a choice's dict representation, return it's local pk which might have been just created.

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

get_remote_choices(*max*)
Parses JSON from the API, return model instances.

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

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

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

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

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

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

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

By default, it will return the model of self.choices.

12.3.3 Javascript fun

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

```
if (window.yourlabs === undefined) window.yourlabs = {};

yourlabs.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(yourlabs.RemoteAutocompleteWidget);
    });
});
```

12.4 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.

12.4.1 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)
```

12.5 Support for django-generic-m2m

See *GenericManyToMany documentation*.

12.6 Support for django-hvad

12.7 Support for django-taggit

django-taggit does it slightly differently. It is supported by autocomplete_light as of 1.0.25. First you need to register the taggit Tag class and for each form you need to set the TaggitWidget.

First register the tag:

```
from taggit.models import Tag
import autocomplete_light.shortcuts as al
al.register(Tag)
```

Every form which should have the autocomplete taggit widget should look like:

```
from autocomplete_light.contrib.taggit_field import TaggitField, TaggitWidget

class AppEditForm(forms.ModelForm):
   tags = TaggitField(widget=TaggitWidget('TagAutocomplete'))
   class Meta:
      model = App
```

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

autocomplete_light.autocomplete, 59
autocomplete_light.autocomplete.generic, 58
autocomplete_light.autocomplete.model, 58
autocomplete_light.autocomplete.template, 56
autocomplete_light.fields, 62
autocomplete_light.registry, 53
autocomplete_light.views, 59
autocomplete_light.widgets, 60

Index

A

add_another_url_kwargs (autocomplete_light.autocomplete.base.AutocompleteBase attribute), 55
add_another_url_name (autocomplete_light.autocomplete.base.AutocompleteBase attribute), 55
attrs (autocomplete_light.widgets.WidgetBase attribute), 60
autocomplete_for_generic() (autocomplete_light.registry.AutocompleteRegistry method), 53
autocomplete_for_model() (autocomplete_light.registry.AutocompleteRegistry method), 53
autocomplete_html() (autocomplete_light.autocomplete.base.AutocompleteBase method), 55
autocomplete_html() (autocomplete_light.autocomplete.base.AutocompleteInterface method), 55
autocomplete_html() (autocomplete_light.autocomplete.template.AutocompleteTemplate method), 56
autocomplete_html_format (autocomplete_light.autocomplete.base.AutocompleteBase attribute), 55
autocomplete_js_attributes (autocomplete_light.widgets.WidgetBase attribute), 61
autocomplete_light.autocomplete (module), 59
autocomplete_light.autocomplete.generic (module), 58
autocomplete_light.autocomplete.model (module), 58
autocomplete_light.autocomplete.template (module), 56
autocomplete_light.fields (module), 62
autocomplete_light.registry (module), 53
autocomplete_light.views (module), 59
autocomplete_light.widgets (module), 60
autocomplete_model_base (autocomplete_light.registry.AutocompleteRegistry attribute), 53
autocomplete_template (autocomplete_light.autocomplete.template.AutocompleteTemplate attribute), 56
AutocompleteBase (class in autocomplete_light.autocomplete.base), 55
AutocompleteChoiceList (class in autocomplete_light.autocomplete.choice_list), 57
AutocompleteGeneric (class in autocomplete_light.autocomplete.generic), 58
AutocompleteInterface (class in autocomplete_light.autocomplete.base), 54
AutocompleteList (class in autocomplete_light.autocomplete.list), 57
AutocompleteModel (class in autocomplete_light.autocomplete.model), 58
AutocompleteRegistry (class in autocomplete_light.registry), 53
AutocompleteRestModel (class in autocomplete_light.autocomplete.rest_model), 77
AutocompleteTemplate (class in autocomplete_light.autocomplete.template), 56
AutocompleteView (class in autocomplete_light.views), 59
autodiscover() (in module autocomplete_light.registry), 54

C

choice_html() (autocomplete_light.autocomplete.base.AutocompleteBase method), 56
choice_html() (autocomplete_light.autocomplete.template.AutocompleteTemplate method), 56
choice_html_format (autocomplete_light.autocomplete.base.AutocompleteBase attribute), 55
choice_label() (autocomplete_light.autocomplete.base.AutocompleteBase

method), 56
choice_label() (autocomplete_light.autocomplete.choice_list.AutocompleteChoiceList method), 57
choice_label() (autocomplete_light.autocomplete.model.AutocompleteModel method), 58
choice_template (autocomplete_light.autocomplete.template.AutocompleteTemplate attribute), 56
choice_value() (autocomplete_light.autocomplete.base.AutocompleteBase method), 56
choice_value() (autocomplete_light.autocomplete.choice_list.AutocompleteChoiceList method), 57
choice_value() (autocomplete_light.autocomplete.generic.AutocompleteGeneric method), 59
choice_value() (autocomplete_light.autocomplete.model.AutocompleteModel method), 58
ChoiceField (class in autocomplete_light.fields), 62
choices (autocomplete_light.autocomplete.choice_list.AutocompleteChoiceList attribute), 57
choices (autocomplete_light.autocomplete.generic.AutocompleteGeneric attribute), 58
choices (autocomplete_light.autocomplete.list.AutocompleteList attribute), 57
choices (autocomplete_light.autocomplete.model.AutocompleteModel attribute), 58
choices_for_request() (autocomplete_light.autocomplete.base.AutocompleteBase method), 56
choices_for_request() (autocomplete_light.autocomplete.choice_list.AutocompleteChoiceList method), 57
choices_for_request() (autocomplete_light.autocomplete.generic.AutocompleteGeneric method), 59
choices_for_request() (autocomplete_light.autocomplete.list.AutocompleteList method), 57
choices_for_request() (autocomplete_light.autocomplete.model.AutocompleteModel method), 58
choices_for_values() (autocomplete_light.autocomplete.base.AutocompleteInterface method), 55
choices_for_values() (autocomplete_light.autocomplete.choice_list.AutocompleteChoiceList method), 57
choices_for_values() (autocomplete_light.autocomplete.generic.AutocompleteGeneric method), 59
choices_for_values() (autocomplete_light.autocomplete.list.AutocompleteList method), 57
choices_for_values() (autocomplete_light.autocomplete.model.AutocompleteModel method), 58
ChoiceWidget (class in autocomplete_light.widgets), 61
CreateView (class in autocomplete_light.views), 59

D

download() (autocomplete_light.autocomplete.rest_model.AutocompleteRe method), 77
download_choice() (autocomplete_light.autocomplete.rest_model.AutocompleteRestModel method), 77

E

empty_html_format (autocomplete_light.autocomplete.base.AutocompleteBase attribute), 55
extra_context (autocomplete_light.widgets.WidgetBase attribute), 61
extract_args() (autocomplete_light.registry.AutocompleteRegistry class method), 53

F

FieldBase (class in autocomplete_light.fields), 62
form_valid() (autocomplete_light.views.CreateView method), 59

G

GenericModelChoiceField (class in autocomplete_light.fields), 63
GenericModelMultipleChoiceField (class in autocomplete_light.fields), 64
get() (autocomplete_light.views.AutocompleteView method), 59
get_absolute_url() (autocomplete_light.autocomplete.base.AutocompleteInterface method), 55
get_add_another_url() (autocomplete_light.autocomplete.base.AutocompleteBase method), 56
get_base_context() (autocomplete_light.autocomplete.template.AutocompleteTemplate method), 56
get_remote_choices() (autocomplete_light.autocomplete.rest_model.AutocompleteRestModel method), 77
get_source_url() (autocomplete_light.autocomplete.rest_model.AutocompleteRestModel method), 77

get_source_url_data() (autocomplete_light.autocomplete.rest_model.AutocompleteRestModel method), 77

L

limit_choices (autocomplete_light.autocomplete.choice_list.AutocompleteChoiceList attribute), 57
limit_choices (autocomplete_light.autocomplete.list.AutocompleteList attribute), 57
limit_choices (autocomplete_light.autocomplete.model.AutocompleteModel attribute), 58

M

model_for_source_url() (autocomplete_light.autocomplete.rest_model.AutocompleteRestModel method), 77
ModelChoiceField (class in autocomplete_light.fields), 63
ModelMultipleChoiceField (class in autocomplete_light.fields), 63
MultipleChoiceField (class in autocomplete_light.fields), 62
MultipleChoiceWidget (class in autocomplete_light.widgets), 61

O

order_by (autocomplete_light.autocomplete.choice_list.AutocompleteChoiceList attribute), 57
order_by (autocomplete_light.autocomplete.list.AutocompleteList attribute), 57
order_by (autocomplete_light.autocomplete.model.AutocompleteModel attribute), 58
order_choices() (autocomplete_light.autocomplete.list.AutocompleteList method), 57
order_choices() (autocomplete_light.autocomplete.model.AutocompleteModel method), 58

P

post() (autocomplete_light.views.AutocompleteView method), 59
prepare_value() (autocomplete_light.fields.GenericModelChoiceField method), 63

R

register() (autocomplete_light.registry.AutocompleteRegistry method), 53
register() (in module autocomplete_light.registry), 54
render() (autocomplete_light.widgets.TextWidget method), 62
render_template_context() (autocomplete_light.autocomplete.template.AutocompleteTemplate method), 56
request (autocomplete_light.autocomplete.base.AutocompleteInterface attribute), 54

S

search_fields (autocomplete_light.autocomplete.generic.AutocompleteGeneric attribute), 58
search_fields (autocomplete_light.autocomplete.model.AutocompleteModel attribute), 58
split_words (autocomplete_light.autocomplete.model.AutocompleteModel attribute), 58

T

TextWidget (class in autocomplete_light.widgets), 62
to_python() (autocomplete_light.fields.GenericModelChoiceField method), 63

U

unregister() (autocomplete_light.registry.AutocompleteRegistry method), 54

V

validate_values() (autocomplete_light.autocomplete.base.AutocompleteBase method), 56
validate_values() (autocomplete_light.autocomplete.base.AutocompleteInterface method), 55
validate_values() (autocomplete_light.autocomplete.generic.AutocompleteGeneric method), 59
validate_values() (autocomplete_light.autocomplete.model.AutocompleteModel method), 58
values (autocomplete_light.autocomplete.base.AutocompleteInterface attribute), 54

W

widget (autocomplete_light.fields.ChoiceField attribute), 62
widget (autocomplete_light.fields.GenericModelChoiceField attribute), 63
widget (autocomplete_light.fields.GenericModelMultipleChoiceField attribute), 64
widget (autocomplete_light.fields.ModelChoiceField attribute), 63
widget (autocomplete_light.fields.ModelMultipleChoiceField attribute), 63

widget (autocomplete_light.fields.MultipleChoiceField attribute), 62
widget_attrs (autocomplete_light.widgets.WidgetBase attribute), 60
widget_js_attributes (autocomplete_light.widgets.WidgetBase attribute), 60
widget_template (autocomplete_light.autocomplete.base.AutocompleteBase attribute), 55
widget_template (autocomplete_light.widgets.WidgetBase attribute), 61
WidgetBase (class in autocomplete_light.widgets), 60