# Django-Selectable Documentation

***Release 1.1.0dev***

**Mark Lavin**

**Jul 15, 2017**

# Contents

Tools and widgets for using/creating auto-complete selection widgets using Django and jQuery UI.

---

**Note:** This project is looking for additional maintainers to help with Django/jQuery compatibility issues as well as addressing support issues/questions. If you are looking to help out on this project and take a look at the open help-wanted or question and see if you can contribute a fix. Be bold! If you want to take a larger role on the project, please reach out on the mailing list. I'm happy to work with you to get you going on an issue.

---

CHAPTER 1

# Features

- Works with the latest jQuery UI Autocomplete library
- Auto-discovery/registration pattern for defining lookups

CHAPTER 2

# Installation Requirements

- Python 2.7, 3.3+
- Django >= 1.7, < 1.11
- jQuery >= 1.9, < 3.0
- jQuery UI >= 1.10, < 1.12

To install:

```
pip install django-selectable
```

Next add *selectable* to your *INSTALLED_APPS* to include the related css/js:

```
INSTALLED_APPS = (
    'contrib.staticfiles',
    # Other apps here
    'selectable',
)
```

The jQuery and jQuery UI libraries are not included in the distribution but must be included in your templates. See the example project for an example using these libraries from the Google CDN.

Once installed you should add the urls to your root url patterns:

```
urlpatterns = [
    # Other patterns go here
    url(r'^selectable/', include('selectable.urls')),
]
```

CHAPTER 3

# Documentation

Documentation for django-selectable is available on Read The Docs.

CHAPTER 4

# Additional Help/Support

You can find additional help or support on the mailing list: http://groups.google.com/group/django-selectable

CHAPTER 5

# Contributing

If you think you've found a bug or are interested in contributing to this project check out our contributing guide.

If you are interested in translating django-selectable into your native language you can join the Transifex project.

Contents:

## Overview

### Motivation

There are many Django apps related to auto-completion why create another? One problem was varying support for the jQuery UI auto-complete plugin versus the now deprecated bassistance version. Another was support for combo-boxes and multiple selects. And lastly was a simple syntax for defining the related backend views for the auto-completion.

**This library aims to meet all of these goals:**

- Built on jQuery UI auto-complete
- **Fields and widgets for a variety of use-cases:**
  - Text inputs and combo-boxes
  - Text selection
  - Value/ID/Foreign key selection
  - Multiple object selection
  - Allowing new values
- Simple and extendable syntax for defining backend views

### Related Projects

Much of the work here was inspired by things that I like (and things I don't like) about django-ajax-selects. To see some of the other Django apps for handling auto-completion see Django-Packages.

## Getting Started

**The workflow for using *django-selectable* involves two main parts:**

- Defining your lookups
- Defining your forms

This guide assumes that you have a basic knowledge of creating Django models and forms. If not you should first read through the documentation on defining models and using forms.

### Including jQuery & jQuery UI

The widgets in django-selectable define the media they need as described in the Django documentation on Form Media. That means to include the javascript and css you need to make the widgets work you can include `{{ form.media.css }}` and `{{ form.media.js }}` in your template. This is assuming your form is called *form* in the template context. For more information please check out the Django documentation.

The jQuery and jQuery UI libraries are not included in the distribution but must be included in your templates. However there is a template tag to easily add these libraries from the from the Google CDN.

```
{% load selectable_tags %}
{% include_jquery_libs %}
```

By default these will use jQuery v1.11.2 and jQuery UI v1.11.3. You can customize the versions used by pass them to the tag. The first version is the jQuery version and the second is the jQuery UI version.

```
{% load selectable_tags %}
{% include_jquery_libs '1.11.2' '1.11.3' %}
```

Django-Selectable should work with jQuery >= 1.9 and jQuery UI >= 1.10.

You must also include a jQuery UI theme stylesheet. There is also a template tag to easily add this style sheet from the Google CDN.

```
{% load selectable_tags %}
{% include_ui_theme %}
```

By default this will use the base theme for jQuery UI v1.11.4. You can configure the theme and version by passing them in the tag.

```
{% load selectable_tags %}
{% include_ui_theme 'ui-lightness' '1.11.4' %}
```

Or only change the theme.

```
{% load selectable_tags %}
{% include_ui_theme 'ui-lightness' %}
```

See the the jQuery UI documentation for a full list of available stable themes: http://jqueryui.com/download#stable-themes

Of course you can choose to include these rescources manually:

```
.. code-block:: html

    <link rel="stylesheet" href="//ajax.googleapis.com/ajax/libs/jqueryui/1.11.3/
↪themes/base/jquery-ui.css" type="text/css">
    <link href="{% static 'selectable/css/dj.selectable.css' %}" type="text/css"␣
↪media="all" rel="stylesheet">
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></
↪script>
    <script src="//ajax.googleapis.com/ajax/libs/jqueryui/1.11.3/jquery-ui.js"></
↪script>
    <script type="text/javascript" src="{% static 'selectable/js/jquery.dj.selectable.
↪js' %}"></script>
```

**Note:** jQuery UI shares a few plugin names with the popular Twitter Bootstrap framework. There are notes on using Bootstrap along with django-selectable in the *advanced usage section*.

## Defining a Lookup

The lookup classes define the backend views. The most common case is defining a lookup which searchs models based on a particular field. Let's define a simple model:

```
from __future__ import unicode_literals

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


@python_2_unicode_compatible
class Fruit(models.Model):
    name = models.CharField(max_length=200)

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

In a *lookups.py* we will define our lookup:

```
from __future__ import unicode_literals

from selectable.base import ModelLookup
from selectable.registry import registry

from .models import Fruit


class FruitLookup(ModelLookup):
    model = Fruit
    search_fields = ('name__icontains', )
```

This lookups extends `selectable.base.ModelLookup` and defines two things: one is the model on which we will be searching and the other is the field which we are searching. This syntax should look familiar as it is the same

as the field lookup syntax for making queries in Django.

Below this definition we will register our lookup class.

```
registry.register(FruitLookup)
```

**Note:** You should only register your lookup once. Attempting to register the same lookup class more than once will lead to `LookupAlreadyRegistered` errors. A common problem related to the `LookupAlreadyRegistered` error is related to inconsistant import paths in your project. Prior to Django 1.4 the default `manage.py` allows for importing both with and without the project name (i.e. `from myproject.myapp import lookups` or `from myapp import lookups`). This leads to the `lookup.py` file being imported twice and the registration code executing twice. Thankfully this is no longer the default in Django 1.4. Keeping your import consistant to include the project name (when your app is included inside the project directory) will avoid these errors.

### Defining Forms

Now that we have a working lookup we will define a form which uses it:

```
from django import forms

from selectable.forms import AutoCompleteWidget

from .lookups import FruitLookup


class FruitForm(forms.Form):
    autocomplete = forms.CharField(
        label='Type the name of a fruit (AutoCompleteWidget)',
        widget=AutoCompleteWidget(FruitLookup),
        required=False,
    )
```

This replaces the default widget for the `CharField` with the `AutoCompleteWidget`. This will allow the user to fill this field with values taken from the names of existing `Fruit` models.

And that's pretty much it. Keep on reading if you want to learn about the other types of fields and widgets that are available as well as defining more complicated lookups.

## Defining Lookups

### What are Lookups?

Lookups define the corresponding ajax views used by the auto-completion fields and widgets. They take in the current request and return the JSON needed by the jQuery auto-complete plugin.

### Defining a Lookup

django-selectable uses a registration pattern similar to the Django admin. Lookups should be defined in a *lookups.py* in your application's module. Once defined you must register in with django-selectable. All lookups must extend from `selectable.base.LookupBase` which defines the API for every lookup.

```
from selectable.base import LookupBase
from selectable.registry import registry

class MyLookup(LookupBase):
    def get_query(self, request, term):
        data = ['Foo', 'Bar']
        return [x for x in data if x.startswith(term)]

registry.register(MyLookup)
```

## Lookup API

LookupBase.**get_query**(*request*, *term*)
This is the main method which takes the current request from the user and returns the data which matches their search.

**Parameters**

- **request** – The current request object.
- **term** – The search term from the widget input.

**Returns** An iterable set of data of items matching the search term.

LookupBase.**get_item_label**(*item*)
This is first of three formatting methods. The label is shown in the drop down menu of search results. This defaults to `item.__unicode__`.

**Parameters** **item** – An item from the search results.

**Returns** A string representation of the item to be shown in the search results. The label can include HTML. For changing the label format on the client side see *Advanced Label Formats*.

LookupBase.**get_item_id**(*item*)
This is second of three formatting methods. The id is the value that will eventually be returned by the field/widget. This defaults to `item.__unicode__`.

**Parameters** **item** – An item from the search results.

**Returns** A string representation of the item to be returned by the field/widget.

LookupBase.**get_item_value**(*item*)
This is last of three formatting methods. The value is shown in the input once the item has been selected. This defaults to `item.__unicode__`.

**Parameters** **item** – An item from the search results.

**Returns** A string representation of the item to be shown in the input.

LookupBase.**get_item**(*value*)
`get_item` is the reverse of `get_item_id`. This should take the value from the form initial values and return the current item. This defaults to simply return the value.

**Parameters** **value** – Value from the form inital value.

**Returns** The item corresponding to the initial value.

LookupBase.**create_item**(*value*)
If you plan to use a lookup with a field or widget which allows the user to input new values then you must define what it means to create a new item for your lookup. By default this raises a `NotImplemented` error.

**Parameters** **value** – The user given value.

**Returns** The new item created from the item.

`LookupBase.`**`format_item`**(*item*)

By default `format_item` creates a dictionary with the three keys used by the UI plugin: id, value, label. These are generated from the calls to `get_item_id`, `get_item_value` and `get_item_label`. If you want to add additional keys you should add them here.

The results of `get_item_label` is conditionally escaped to prevent Cross Site Scripting (XSS) similar to the templating language. If you know that the content is safe and you want to use these methods to include HTML should mark the content as safe with `django.utils.safestring.mark_safe` inside the `get_item_label` method.

`get_item_id` and `get_item_value` are not escapted by default. These are not a XSS vector with the built-in JS. If you are doing additional formating using these values you should be conscience of this fake and be sure to escape these values.

**Parameters** **`item`** – An item from the search results.

**Returns** A dictionary of information for this item to be sent back to the client.

There are also some additional methods that you could want to use/override. These are for more advanced use cases such as using the lookups with JS libraries other than jQuery UI. Most users will not need to override these methods.

`LookupBase.`**`format_results`**(*self*, *raw_data*, *options*)

Returns a python structure that later gets serialized. This makes a call to *paginate_results* prior to calling *format_item* on each item in the current page.

**Parameters**

- **`raw_data`** – The set of all matched results.
- **`options`** – Dictionary of `cleaned_data` from the lookup form class.

**Returns** A dictionary with two keys `meta` and `data`. The value of `data` is an iterable extracted from page_data. The value of `meta` is a dictionary. This is a copy of options with one additional element `more` which is a translatable "Show more" string (useful for indicating more results on the javascript side).

`LookupBase.`**`paginate_results`**(*results*, *options*)

If *SELECTABLE_MAX_LIMIT* is defined or `limit` is passed in request.GET then `paginate_results` will return the current page using Django's built in pagination. See the Django docs on pagination for more info.

**Parameters**

- **`results`** – The set of all matched results.
- **`options`** – Dictionary of `cleaned_data` from the lookup form class.

**Returns** The current Page object of results.

## Lookups Based on Models

Perhaps the most common use case is to define a lookup based on a given Django model. For this you can extend `selectable.base.ModelLookup`. To extend `ModelLookup` you should set two class attributes: `model` and `search_fields`.

```
from __future__ import unicode_literals

from selectable.base import ModelLookup
from selectable.registry import registry
```

```
from .models import Fruit


class FruitLookup(ModelLookup):
    model = Fruit
    search_fields = ('name__icontains', )

registry.register(FruitLookup)
```

The syntax for `search_fields` is the same as the Django field lookup syntax. Each of these lookups are combined as OR so any one of them matching will return a result. You may optionally define a third class attribute `filters` which is a dictionary of filters to be applied to the model queryset. The keys should be a string defining a field lookup and the value should be the value for the field lookup. Filters on the other hand are combined with AND.

## User Lookup Example

Below is a larger model lookup example using multiple search fields, filters and display options for the auth.User model.

```
from django.contrib.auth.models import User
from selectable.base import ModelLookup
from selectable.registry import registry


class UserLookup(ModelLookup):
    model = User
    search_fields = (
        'username__icontains',
        'first_name__icontains',
        'last_name__icontains',
    )
    filters = {'is_active': True, }

    def get_item_value(self, item):
        # Display for currently selected item
        return item.username

    def get_item_label(self, item):
        # Display for choice listings
        return u"%s (%s)" % (item.username, item.get_full_name())

registry.register(UserLookup)
```

## Lookup Decorators

Registering lookups with django-selectable creates a small API for searching the lookup data. While the amount of visible data is small there are times when you want to restrict the set of requests which can view the data. For this purpose there are lookup decorators. To use them you simply decorate your lookup class.

```
from django.contrib.auth.models import User
from selectable.base import ModelLookup
from selectable.decorators import login_required
from selectable.registry import registry
```

```
@login_required
class UserLookup(ModelLookup):
    model = User
    search_fields = ('username__icontains', )
    filters = {'is_active': True, }

registry.register(UserLookup)
```

**Note:** The class decorator syntax was introduced in Python 2.6. If you are using django-selectable with Python 2.5 you can still make use of these decorators by applying the without the decorator syntax.

```
class UserLookup(ModelLookup):
    model = User
    search_fields = ('username__icontains', )
    filters = {'is_active': True, }

UserLookup = login_required(UserLookup)

registry.register(UserLookup)
```

Below are the descriptions of the available lookup decorators.

### ajax_required

The django-selectable javascript will always request the lookup data via XMLHttpRequest (AJAX) request. This decorator enforces that the lookup can only be accessed in this way. If the request is not an AJAX request then it will return a 400 Bad Request response.

### login_required

This decorator requires the user to be authenticated via `request.user.is_authenticated`. If the user is not authenticated this will return a 401 Unauthorized response. `request.user` is set by the `django.contrib.auth.middleware.AuthenticationMiddleware` which is required for this decorator to work. This middleware is enabled by default.

### staff_member_required

This decorator builds from `login_required` and in addition requires that `request.user.is_staff` is `True`. If the user is not authenticatated this will continue to return at 401 response. If the user is authenticated but not a staff member then this will return a 403 Forbidden response.

## Advanced Usage

We've gone through the most command and simple use cases for django-selectable. Now we'll take a look at some of the more advanced features of this project. This assumes that you are comfortable reading and writing a little bit of Javascript making use of jQuery.

## Additional Parameters

The basic lookup is based on handling a search based on a single term string. If additional filtering is needed it can be inside the lookup `get_query` but you would need to define this when the lookup is defined. While this fits a fair number of use cases there are times when you need to define additional query parameters that won't be known until either the form is bound or until selections are made on the client side. This section will detail how to handle both of these cases.

### How Parameters are Passed

As with the search term, the additional parameters you define will be passed in `request.GET`. Since `get_query` gets the current request, you will have access to them. Since they can be manipulated on the client side, these parameters should be treated like all user input. It should be properly validated and sanitized.

### Limiting the Result Set

The number of results are globally limited/paginated by the *SELECTABLE_MAX_LIMIT* but you can also lower this limit on the field or widget level. Each field and widget takes a `limit` argument in the `__init__` that will be passed back to the lookup through the `limit` query parameter. The result set will be automatically paginated for you if you use either this parameter or the global setting.

### Adding Parameters on the Server Side

Each of the widgets define `update_query_parameters` which takes a dictionary. The most common way to use this would be in the form `__init__`.

```
class FruitForm(forms.Form):
    autocomplete = forms.CharField(
        label='Type the name of a fruit (AutoCompleteWidget)',
        widget=selectable.AutoCompleteWidget(FruitLookup),
        required=False,
    )

    def __init__(self, *args, **kwargs):
        super(FruitForm, self).__init__(*args, **kwargs)
        self.fields['autocomplete'].widget.update_query_parameters({'foo':
↪'bar'})
```

You can also pass the query parameters into the widget using the `query_params` keyword argument. It depends on your use case as to whether the parameters are known when the form is defined or when an instance of the form is created.

### Adding Parameters on the Client Side

There are times where you want to filter the result set based other selections by the user such as a filtering cities by a previously selected state. In this case you will need to bind a `prepareQuery` to the field. This function should accept the query dictionary. You are free to make adjustments to the query dictionary as needed.

```
<script type="text/javascript">
    function newParameters(query) {
        query.foo = 'bar';
    }
```

```
    $(document).ready(function() {
        $('#id_autocomplete').djselectable('option', 'prepareQuery',␣
↪newParameters);
    });
</script>
```

**Note:** In v0.7 the scope of `prepareQuery` was updated so that `this` refers to the current `djselectable` plugin instance. Previously `this` refered to the plugin `options` instance.

## Chained Selection

It's a fairly common pattern to have two or more inputs depend one another such City/State/Zip. In fact there are other Django apps dedicated to this purpose such as django-smart-selects or django-ajax-filtered-fields. It's possible to handle this kind of selection with django-selectable if you are willing to write a little javascript.

Suppose we have city model

```
from __future__ import unicode_literals

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

from localflavor.us.models import USStateField


@python_2_unicode_compatible
class City(models.Model):
    name = models.CharField(max_length=200)
    state = USStateField()

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

Then in our lookup we will grab the state value and filter our results on it:

```
from __future__ import unicode_literals

from selectable.base import ModelLookup
from selectable.registry import registry

from .models import City


class CityLookup(ModelLookup):
    model = City
    search_fields = ('name__icontains', )

    def get_query(self, request, term):
        results = super(CityLookup, self).get_query(request, term)
        state = request.GET.get('state', '')
        if state:
            results = results.filter(state=state)
        return results

    def get_item_label(self, item):
```

```
        return "%s, %s" % (item.name, item.state)


registry.register(CityLookup)
```

and a simple form

```
from django import forms

from localflavor.us.forms import USStateField, USStateSelect

from selectable.forms import AutoCompleteSelectField,␣
↪AutoComboboxSelectWidget

from .lookups import CityLookup


class ChainedForm(forms.Form):
    city = AutoCompleteSelectField(
        lookup_class=CityLookup,
        label='City',
        required=False,
        widget=AutoComboboxSelectWidget
    )
    state = USStateField(widget=USStateSelect, required=False)
```

We want our users to select a city and if they choose a state then we will only show them cities in that state. To do this we will pass back chosen state as addition parameter with the following javascript:

```
<script type="text/javascript">
    $(document).ready(function() {
        function newParameters(query) {
            query.state = $('#id_state').val();
        }
        $('#id_city_0').djselectable('option', 'prepareQuery',␣
↪newParameters);
    });
</script>
```

And that's it! We now have a working chained selection example. The full source is included in the example project.

### Detecting Client Side Changes

The previous example detected selection changes on the client side to allow passing parameters to the lookup. Since django-selectable is built on top of the jQuery UI Autocomplete plug-in, the widgets expose the events defined by the plugin.

- djselectablecreate
- djselectablesearch
- djselectableopen
- djselectablefocus
- djselectableselect
- djselectableclose

- djselectablechange

For the most part these event names should be self-explanatory. If you need additional detail you should refer to the jQuery UI docs on these events.

The multiple select widgets include additional events which indicate when a new item is added or removed from the current list. These events are `djselectableadd` and `djselectableremove`. These events pass a dictionary of data with the following keys

- element: The original text input
- input: The hidden input to be added for the new item
- wrapper: The `<li>` element to be added to the deck
- deck: The outer `<ul>` deck element

You can use these events to prevent items from being added or removed from the deck by returning `false` in the handling function. A simple example is given below:

```
<script type="text/javascript">
    $(document).ready(function() {
        $(':input[name=my_field_0]').bind('djselectableadd', function(event,␣
↪item) {
            // Don't allow foo to be added
            if ($(item.input).val() === 'foo') {
                return false;
            }
        });
    });
</script>
```

## Submit On Selection

You might want to help your users by submitting the form once they have selected a valid item. To do this you simply need to listen for the `djselectableselect` event. This event is fired by the text input which has an index of 0. If your field is named `my_field` then input to watch would be `my_field_0` such as:

```
<script type="text/javascript">
    $(document).ready(function() {
        $(':input[name=my_field_0]').bind('djselectableselect',␣
↪function(event, ui) {
            $(this).parents("form").submit();
        });
    });
</script>
```

## Dynamically Added Forms

django-selectable can work with dynamically added forms such as inlines in the admin. To make django-selectable work in the admin there is nothing more to do than include the necessary static media as described in the *Admin Integration* section.

If you are making use of the popular django-dynamic-formset then you can make django-selectable work by passing `bindSelectables` to the added option:

```
<script type="text/javascript">
    $(document).ready(function() {
        $('#my-formset').formset({
                added: bindSelectables
        });
    });
</script>
```

Currently you must include the django-selectable javascript below this formset initialization code for this to work. See django-selectable issue #31 for some additional detail on this problem.

## Label Formats on the Client Side

The lookup label is the text which is shown in the list before it is selected. You can use the *get_item_label* method in your lookup to do this on the server side. This works for most applications. However if you don't want to write your HTML in Python or need to adapt the format on the client side you can use the *formatLabel* option.

`formatLabel` takes two paramaters the current label and the current selected item. The item is a dictionary object matching what is returned by the lookup's *format_item*. `formatLabel` should return the string which should be used for the label.

Going back to the `CityLookup` we can adjust the label to wrap the city and state portions with their own classes for additional styling:

```
<script type="text/javascript">
    $(document).ready(function() {
        function formatLabel(label, item) {
            var data = label.split(',');
            return '<span class="city">' + data[0] + '</span>, <span class=
↪"state">' + data[1] + '</span>';
        }
        $('#id_city_0').djselectable('option', 'formatLabel', formatLabel);
    });
</script>
```

This is a rather simple example but you could also pass additional information in `format_item` such as a flag of whether the city is the capital and render the state captials differently.

## Using with Twitter Bootstrap

django-selectable can work along side with Twitter Bootstrap but there are a few things to take into consideration. Both jQuery UI and Bootstrap define a `$.button` plugin. This plugin is used by default by django-selectable and expects the UI version. If the jQuery UI JS is included after the Bootstrap JS then this will work just fine but the Bootstrap button JS will not be available. This is the strategy taken by the jQuery UI Bootstrap theme.

Another option is to rename the Bootstrap plugin using the `noConflict` option.

```
<!-- Include Bootstrap JS -->
<script>$.fn.bootstrapBtn = $.fn.button.noConflict();</script>
<!-- Include jQuery UI JS -->
```

Even with this some might complain that it's too resource heavy to include all of jQuery UI when you just want the autocomplete to work with django-selectable. For this you can use the Download Builder to build a minimal set of jQuery UI widgets. django-selectable requires the UI core, autocomplete, menu and button widgets. None of the

effects or interactions are needed. Minified this totals around 100 kb of JS, CSS and images (based on jQuery UI 1.10).

**Note:** For a comparison this is smaller than the minified Bootstrap 2.3.0 CSS which is 105 kb not including the responsive CSS or the icon graphics.

It is possible to remove the dependency on the UI button plugin and instead use the Bootstrap button styles. This is done by overriding the `_comboButtonTemplate` and `_removeButtonTemplate` functions used to create the buttons. An example is given below.

```
<script>
    $.ui.djselectable.prototype._comboButtonTemplate = function (input) {
        var icon = $("<i>").addClass("icon-chevron-down");
        // Remove current classes on the text input
        $(input).attr("class", "");
        // Wrap with input-append
        $(input).wrap('<div class="input-append" />');
        // Return button link with the chosen icon
        return $("<a>").append(icon).addClass("btn btn-small");
    };
    $.ui.djselectable.prototype._removeButtonTemplate = function (item) {
        var icon = $("<i>").addClass("icon-remove-sign");
        // Return button link with the chosen icon
        return $("<a>").append(icon).addClass("btn btn-small pull-right");
    };
</script>
```

# Admin Integration

## Overview

Django-Selectables will work in the admin. To get started on integrated the fields and widgets in the admin make sure you are familiar with the Django documentation on the ModelAdmin.form and ModelForms particularly on overriding the default widgets. As you will see integrating django-selectable in the adminis the same as working with regular forms.

## Including jQuery & jQuery UI

As noted *in the quick start guide*, the jQuery and jQuery UI libraries are not included in the distribution but must be included in your templates. For the Django admin that means overriding admin/base_site.html. You can include this media in the block name *extrahead* which is defined in admin/base.html.

```
{% block extrahead %}
    {% load selectable_tags %}
    {% include_ui_theme %}
    {% include_jquery_libs %}
    {{ block.super }}
{% endblock %}
```

See the Django documentation on overriding admin templates. See the example project for the full template example.

## Using Grappelli

Grappelli is a popular customization of the Django admin interface. It includes a number of interface improvements which are also built on top of jQuery UI. When using Grappelli you do not need to make any changes to the `admin/base_site.html` template. django-selectable will detect jQuery and jQuery UI versions included by Grappelli and make use of them.

## Basic Example

For example, we may have a `Farm` model with a foreign key to `auth.User` and a many to many relation to our `Fruit` model.

```
from __future__ import unicode_literals

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


@python_2_unicode_compatible
class Fruit(models.Model):
    name = models.CharField(max_length=200)

    def __str__(self):
        return self.name


@python_2_unicode_compatible
class Farm(models.Model):
    name = models.CharField(max_length=200)
    owner = models.ForeignKey('auth.User', related_name='farms')
    fruit = models.ManyToManyField(Fruit)

    def __str__(self):
        return "%s's Farm: %s" % (self.owner.username, self.name)
```

In *admin.py* we will define the form and associate it with the *FarmAdmin*.

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.models import User
from django import forms

from selectable.forms import AutoCompleteSelectField,␣
↪AutoCompleteSelectMultipleWidget

from .models import Fruit, Farm
from .lookups import FruitLookup, OwnerLookup


class FarmAdminForm(forms.ModelForm):
    owner = AutoCompleteSelectField(lookup_class=OwnerLookup, allow_new=True)

    class Meta(object):
        model = Farm
        widgets = {
            'fruit': AutoCompleteSelectMultipleWidget(lookup_
↪class=FruitLookup),
```

```
        }
        exclude = ('owner', )

    def __init__(self, *args, **kwargs):
        super(FarmAdminForm, self).__init__(*args, **kwargs)
        if self.instance and self.instance.pk and self.instance.owner:
            self.initial['owner'] = self.instance.owner.pk

    def save(self, *args, **kwargs):
        owner = self.cleaned_data['owner']
        if owner and not owner.pk:
            owner = User.objects.create_user(username=owner.username, email='
↪')
        self.instance.owner = owner
        return super(FarmAdminForm, self).save(*args, **kwargs)


class FarmAdmin(admin.ModelAdmin):
    form = FarmAdminForm


admin.site.register(Farm, FarmAdmin)
```

You'll note this form also allows new users to be created and associated with the farm, if no user is found matching the given name. To make use of this feature we need to add `owner` to the exclude so that it will pass model validation. Unfortunately that means we must set the owner manual in the save and in the initial data because the `ModelForm` will no longer do this for you. Since `fruit` does not allow new items you'll see these steps are not necessary.

The django-selectable widgets are compatitible with the add another popup in the admin. It's that little green plus sign that appears next to `ForeignKey` or `ManyToManyField` items. This makes django-selectable a user friendly replacement for the ModelAdmin.raw_id_fields when the default select box grows too long.

## Inline Example

With our `Farm` model we can also associate the `UserAdmin` with a `Farm` by making use of the InlineModelAdmin. We can even make use of the same `FarmAdminForm`.

```
# continued from above

class FarmInline(admin.TabularInline):
    model = Farm
    form = FarmAdminForm


class NewUserAdmin(UserAdmin):
    inlines = [
        FarmInline,
    ]


admin.site.unregister(User)
admin.site.register(User, NewUserAdmin)
```

The auto-complete functions will be bound as new forms are added dynamically.

# Testing Forms and Lookups

django-selectable has its own test suite for testing the rendering, validation and server-side logic it provides. However, depending on the additional customizations you add to your forms and lookups you most likely will want to include tests of your own. This section contains some tips or techniques for testing your lookups.

This guide assumes that you are reasonable familiar with the concepts of unit testing including Python's unittest module and Django's testing guide.

## Testing Forms with django-selectable

For the most part testing forms which use django-selectable's custom fields and widgets is the same as testing any Django form. One point that is slightly different is that the select and multi-select widgets are MultiWidgets. The effect of this is that there are two names in the post rather than one. Take the below form for example.

```
# models.py

from django.db import models

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

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

```
# lookups.py

from selectable.base import ModelLookup
from selectable.registry import registry

from .models import Thing

class ThingLookup(ModelLookup):
    model = Thing
    search_fields = ('name__icontains', )

registry.register(ThingLookup)
```

```
# forms.py

from django import forms

from selectable.forms import AutoCompleteSelectField

from .lookups import ThingLookup

class SimpleForm(forms.Form):
    "Basic form for testing."
    thing = AutoCompleteSelectField(lookup_class=ThingLookup)
```

This form has a single field to select a `Thing`. It does not allow new items. Let's write some simple tests for this form.

```
# tests.py

from django.test import TestCase

from .forms import SimpleForm
from .models import Thing

class SimpleFormTestCase(TestCase):

    def test_valid_form(self):
        "Submit valid data."
        thing = Thing.objects.create(name='Foo', description='Bar')
        data = {
            'thing_0': thing.name,
            'thing_1': thing.pk,
        }
        form = SimpleForm(data=data)
        self.assertTrue(form.is_valid())

    def test_invalid_form(self):
        "Thing is required but missing."
        data = {
            'thing_0': 'Foo',
            'thing_1': '',
        }
        form = SimpleForm(data=data)
        self.assertFalse(form.is_valid())
```

Here you will note that while there is only one field `thing` it requires two items in the POST the first is for the text input and the second is for the hidden input. This is again due to the use of MultiWidget for the selection.

There is compatibility code in the widgets to lookup the original name from the POST. This makes it easier to transition to the the selectable widgets without breaking existing tests.

## Testing Lookup Results

Testing the lookups used by django-selectable is similar to testing your Django views. While it might be tempting to use the Django test client, it is slightly easier to use the request factory. A simple example is given below.

```
# tests.py

import json

from django.test import TestCase
from django.test.client import RequestFactory

from .lookups import ThingLookup
from .models import Thing

class ThingLookupTestCase(TestCase):

    def setUp(self):
        self.factory = RequestFactory()
        self.lookup = ThingLookup()
        self.test_thing = Thing.objects.create(name='Foo', description='Bar')

    def test_results(self):
```

```
        "Test full response."
        request = self.factory.get("/", {'term': 'Fo'})
        response = self.lookup.results(request)
        data = json.loads(response.content)['data']
        self.assertEqual(1, len(data))
        self.assertEqual(self.test_thing.pk, data[1]['id'])

    def test_label(self):
        "Test item label."
        label = self.lookup.get_item_label(self.test_thing)
        self.assertEqual(self.test_thing.name, label)
```

As shown in the `test_label` example it is not required to test the full request/response. You can test each of the methods in the lookup API individually. When testing your lookups you should focus on testing the portions which have been customized by your application.

# Fields

Django-Selectable defines a number of fields for selecting either single or multiple lookup items. Item in this context corresponds to the object return by the underlying lookup `get_item`. The single select select field *AutoCompleteSelectField* allows for the creation of new items. To use this feature the field's lookup class must define `create_item`. In the case of lookups extending from *Lookups Based on Models* newly created items have not yet been saved into the database and saving should be handled by the form. All fields take the lookup class as the first required argument.

## AutoCompleteSelectField

Field tied to *AutoCompleteSelectWidget* to bind the selection to the form and create new items, if allowed. The `allow_new` keyword argument (default: `False`) which determines if the field allows new items. This field cleans to a single item.

```
from django import forms

from selectable.forms import AutoCompleteSelectField

from .lookups import FruitLookup


class FruitSelectionForm(forms.Form):
    fruit = AutoCompleteSelectField(lookup_class=FruitLookup, label='Select␣
↪a fruit')
```

*lookup_class'* may also be a dotted path.

## AutoCompleteSelectMultipleField

Field tied to *AutoCompleteSelectMultipleWidget* to bind the selection to the form. This field cleans to a list of items. *AutoCompleteSelectMultipleField* does not allow for the creation of new items.

```
from django import forms

from selectable.forms import AutoCompleteSelectMultipleField
```

```
from .lookups import FruitLookup


class FruitsSelectionForm(forms.Form):
    fruits = AutoCompleteSelectMultipleField(lookup_class=FruitLookup,
        label='Select your favorite fruits')
```

# Widgets

Below are the custom widgets defined by Django-Selectable. All widgets take the lookup class as the first required argument.

These widgets all support a `query_params` keyword argument which is used to pass additional query parameters to the lookup search. See the section on *Adding Parameters on the Server Side* for more information.

You can configure the plugin options by passing the configuration dictionary in the `data-selectable-options` attribute. The set of options availble include those define by the base autocomplete plugin as well as the *removeIcon*, *comboboxIcon*, and *highlightMatch* options which are unique to django-selectable.

```
attrs = {'data-selectable-options': {'highlightMatch': True, 'minLength': 5}}
selectable.AutoCompleteSelectWidget(lookup_class=FruitLookup, attrs=attrs)
```

## AutoCompleteWidget

Basic widget for auto-completing text. The widget returns the item value as defined by the lookup `get_item_value`. If the `allow_new` keyword argument is passed as true it will allow the user to type any text they wish.

## AutoComboboxWidget

Similar to *AutoCompleteWidget* but has a button to reveal all options.

## AutoCompleteSelectWidget

Widget for selecting a value/id based on input text. Optionally allows selecting new items to be created. This widget should be used in conjunction with the *AutoCompleteSelectField* as it will return both the text entered by the user and the id (if an item was selected/matched).

*AutoCompleteSelectWidget* works directly with Django's ModelChoiceField. You can simply replace the widget without replacing the entire field.

```
class FarmAdminForm(forms.ModelForm):

    class Meta(object):
        model = Farm
        widgets = {
            'owner': selectable.AutoCompleteSelectWidget(lookup_
↪class=FruitLookup),
        }
```

The one catch is that you must use `allow_new=False` which is the default.

`lookup_class` may also be a dotted path.

```
widget = selectable.AutoCompleteWidget(lookup_class='core.lookups.FruitLookup
↪')
```

### AutoComboboxSelectWidget

Similar to *AutoCompleteSelectWidget* but has a button to reveal all options.

*AutoComboboxSelectWidget* works directly with Django's ModelChoiceField. You can simply replace the widget without replacing the entire field.

```
class FarmAdminForm(forms.ModelForm):

    class Meta(object):
        model = Farm
        widgets = {
            'owner': selectable.AutoComboboxSelectWidget(lookup_
↪class=FruitLookup),
        }
```

The one catch is that you must use `allow_new=False` which is the default.

### AutoCompleteSelectMultipleWidget

Builds a list of selected items from auto-completion. This widget will return a list of item ids as defined by the lookup `get_item_id`. Using this widget with the *AutoCompleteSelectMultipleField* will clean the items to the item objects. This does not allow for creating new items. There is another optional keyword argument `postion` which can take four possible values: *bottom*, *bottom-inline*, *top* or *top-inline*. This determine the position of the deck list of currently selected items as well as whether this list is stacked or inline. The default is *bottom*.

### AutoComboboxSelectMultipleWidget

Same as *AutoCompleteSelectMultipleWidget* but with a combobox.

## Settings

### SELECTABLE_MAX_LIMIT

This setting is used to limit the number of results returned by the auto-complete fields. Each field/widget can individually lower this maximum. The result sets will be paginated allowing the client to ask for more results. The limit is passed as a query parameter and validated against this value to ensure the client cannot manipulate the query string to retrive more values.

Default: `25`

## SELECTABLE_ESCAPED_KEYS

The `LookupBase.format_item` will conditionally escape result keys based on this setting. The label is escaped by default to prevent a XSS flaw when using the jQuery UI autocomplete. If you are using the lookup responses for a different autocomplete plugin then you may need to esacpe more keys by default.

Default: `('label', )`

**Note:** You probably don't want to include `id` in this setting.

## Javascript Plugin Options

Below the options for configuring the Javascript behavior of the django-selectable widgets.

### removeIcon

This is the class name used for the remove buttons for the multiple select widgets. The set of icon classes built into the jQuery UI framework can be found here: http://jqueryui.com/themeroller/

Default: `ui-icon-close`

### comboboxIcon

This is the class name used for the combobox dropdown icon. The set of icon classes built into the jQuery UI framework can be found here: http://jqueryui.com/themeroller/

Default: `ui-icon-triangle-1-s`

### prepareQuery

`prepareQuery` is a function that is run prior to sending the search request to the server. It is an oppotunity to add additional parameters to the search query. It takes one argument which is the current search parameters as a dictionary. For more information on its usage see *Adding Parameters on the Client Side*.

Default: `null`

### highlightMatch

If true the portions of the label which match the current search term will be wrapped in a span with the class `highlight`.

Default: `true`

### formatLabel

`formatLabel` is a function that is run prior to rendering the search results in the dropdown menu. It takes two arguments: the current item label and the item data dictionary. It should return the label which should be used. For more information on its usage see *Label Formats on the Client Side*.

Default: `null`

# Contributing

There are plenty of ways to contribute to this project. If you think you've found a bug please submit an issue. If there is a feature you'd like to see then please open an ticket proposal for it. If you've come up with some helpful examples then you can add to our example project.

## Getting the Source

The source code is hosted on Github. You can download the full source by cloning the git repo:

```
git clone git://github.com/mlavin/django-selectable.git
```

Feel free to fork the project and make your own changes. If you think that it would be helpful for other then please submit a pull request to have it merged in.

## Submit an Issue

The issues are also managed on Github issue page. If you think you've found a bug it's helpful if you indicate the version of django-selectable you are using the ticket version flag. If you think your bug is javascript related it is also helpful to know the version of jQuery, jQuery UI, and the browser you are using.

Issues are also used to track new features. If you have a feature you would like to see you can submit a proposal ticket. You can also see features which are planned here.

## Submit a Translation

We are working towards translating django-selectable into different languages. There are not many strings to be translated so it is a reasonably easy task and a great way to be involved with the project. The translations are managed through Transifex.

## Running the Test Suite

There are a number of tests in place to test the server side code for this project. To run the tests you need Django and mock installed and run:

```
python runtests.py
```

tox is used to test django-selectable against multiple versions of Django/Python. With tox installed you can run:

```
tox
```

to run all the version combinations. You can also run tox against a subset of supported environments:

```
tox -e py27-django15
```

For more information on running/installing tox please see the tox documentation: http://tox.readthedocs.org/en/latest/index.html

Client side tests are written using QUnit. They can be found in `selectable/tests/qunit/index.html`. The test suite also uses PhantomJS to run the tests. You can install PhantomJS from NPM:

```
# Install requirements
npm install -g phantomjs jshint
make test-js
```

### Building the Documentation

The documentation is built using Sphinx and available on Read the Docs. With Sphinx installed you can build the documentation by running:

```
make html
```

inside the docs directory. Documentation fixes and improvements are always welcome.

## Release Notes

### v1.0.0 (Released 2017-04-14)

This project has been stable for quite some time and finally declaring a 1.0 release. With that comes new policies on official supported versions for Django, Python, jQuery, and jQuery UI.

- New translations for German and Czech.
- Various bug and compatibility fixes.
- Updated example project.

Special thanks to Raphael Merx for helping track down issues related to this release and an updating the example project to work on Django 1.10.

#### Backwards Incompatible Changes

- Dropped support Python 2.6 and 3.2
- Dropped support for Django < 1.7. Django 1.11 is not yet supported.
- `LookupBase.serialize_results` had been removed. This is now handled by the built-in `JsonResponse` in Django.
- jQuery and jQuery UI versions for the `include_jquery_libs` and `include_ui_theme` template tags have been increased to 1.12.4 and 1.11.4 respectively.
- Dropped testing support for jQuery < 1.9 and jQuery UI < 1.10. Earlier versions may continue to work but it is recommended to upgrade.

### v0.9.0 (Released 2014-10-21)

This release primarily addresses incompatibility with Django 1.7. The app-loading refactor both broke the previous registration and at the same time provided better utilities in Django core to make it more robust.

- Compatibility with Django 1.7. Thanks to Calvin Spealman for the fixes.
- Fixes for Python 3 support.

### Backwards Incompatible Changes

- Dropped support for jQuery < 1.7

## v0.8.0 (Released 2014-01-20)

- Widget media references now include a version string for cache-busting when upgrading django-selectable. Thanks to Ustun Ozgur.
- Added compatibility code for *SelectWidgets to handle POST data for the default SelectWidget. Thanks to leo-the-manic.
- Development moved from Bitbucket to Github.
- Update test suite compatibility with new test runner in Django 1.6. Thanks to Dan Poirier for the report and fix.
- Tests now run on Travis CI.
- Added French and Chinese translations.

### Backwards Incompatible Changes

- Support for Django < 1.5 has been dropped. Most pieces should continue to work but there was an ugly JS hack to make django-selectable work nicely in the admin which too flakey to continue to maintain. If you aren't using the selectable widgets in inline-forms in the admin you can most likely continue to use Django 1.4 without issue.

## v0.7.0 (Released 2013-03-01)

This release features a large refactor of the JS plugin used by the widgets. While this over makes the plugin more maintainable and allowed for some of the new features in this release, it does introduce a few incompatible changes. For the most part places where you might have previously used the `autocomplete` namespace/plugin, those references should be updated to reference the `djselectable` plugin.

This release also adds experimental support for Python 3.2+ to go along with Django's support in 1.5. To use Python 3 with django-selectable you will need to use Django 1.5+.

- Experimental Python 3.2+ support
- Improved the scope of `prepareQuery` and `formatLabel` options. Not fully backwards compatible. Thanks to Augusto Men.
- Allow passing the Python path string in place of the lookup class to the fields and widgets. Thanks to Michael Manfre.
- Allow passing JS plugin options through the widget `attrs` option. Thanks to Felipe Prenholato.
- Tests for compatibility with jQuery 1.6 through 1.9 and jQuery UI 1.8 through 1.10.
- Added notes on Bootstrap compatibility.
- Added compatibility with Grappelli in the admin.
- Added Spanish translation thanks to Manuel Alvarez.
- Added documentation notes on testing.

### Bug Fixes

- Fixed bug with matching hidden input when the name contains '_1'. Thanks to Augusto Men for the report and fix.
- Fixed bug where the enter button would open the combobox options rather than submit the form. Thanks to Felipe Prenholato for the report.
- Fixed bug with using `allow_new=True` creating items when no data was submitted. See #91.
- Fixed bug with widget `has_changed` when there is no initial data. See #92.

### Backwards Incompatible Changes

- The JS event namespace has changed from `autocomplete` to `djselectable`.
- `data('autocomplete')` is no longer available on the widgets on the client-side. Use `data('djselectable')` instead.
- Combobox button was changed from a `<button>` to `<a>`. Any customized styles you may have should be updated.
- Combobox no longer changes the `minLength` or `delay` options.

## v0.6.2 (Released 2012-11-07)

### Bug Fixes

- Fixed bug with special characters when highlighting matches. Thanks to Chad Files for the report.
- Fixed javascript bug with spaces in `item.id`. Thanks to @dc for the report and fix.

## v0.6.1 (Released 2012-10-13)

### Features

- Added Polish translation. Thanks to Sławomir Ehlert.

### Bug Fixes

- Fixed incompatibility with jQuery UI 1.9.

## v0.6.0 (Released 2012-10-09)

This release continues to clean up the API and JS. This was primarily motivated by Sławomir Ehlert (@slafs) who is working on an alternate implementation which uses Select2 rather than jQuery UI. This opens the door for additional apps which use the same lookup declaration API with a different JS library on the front end.

Python 2.5 support has been dropped to work towards Python 3 support. This also drops Django 1.2 support which is no longer receiving security fixes.

### Features

- Initial translations (pt_BR). Thanks to Felipe Prenholato for the patch.
- Upgraded default jQuery UI version included by the template tags from 1.8.18 to 1.8.23
- Added `djselectableadd` and `djselectableremove` events fired when items are added or removed from a mutliple select

### Bug Fixes

- Cleaned up JS scoping problems when multiple jQuery versions are used on the page. Thanks Antti Kaihola for the report.
- Fixed minor JS bug where text input was not cleared when selected via the combobox in the multiselect. Thanks Antti Kaihola for the report and Lukas Pirl for a hotfix.

### Backwards Incompatible Changes

- `get_item_value` and `get_item_id` are no longer marked as safe by default.
- Removed AutoComboboxSelectField and AutoComboboxSelectMultipleField. These were deprecated in 0.5.
- Dropping official Python 2.5 support.
- Dropping official Django 1.2 support.
- `paginate_results` signature changed as part of the lookup refactor.
- `SELECTABLE_MAX_LIMIT` can no longer be `None`.

## v0.5.2 (Released 2012-06-27)

### Bug Fixes

- Fixed XSS flaw with lookup `get_item_*` methods. Thanks slafs for the report.
- Fixed bug when passing widget instance rather than widget class to `AutoCompleteSelectField` or `AutoCompleteSelectMultipleField`.

## v0.5.1 (Released 2012-06-08)

### Bug Fixes

- Fix for double `autocompleteselect` event firing.
- Fix for broken pagination in search results. Thanks David Ray for report and fix.

## v0.4.2 (Released 2012-06-08)

### Bug Fixes

- Backported fix for double `autocompleteselect` event firing.
- Backported fix for broken pagination in search results.

## v0.5.0 (Released 2012-06-02)

### Features

- Template tag to add necessary jQuery and jQuery UI libraries. Thanks to Rick Testore for the initial implementation
- *Lookup decorators* for requiring user authentication or staff access to use the lookup
- Additional documentation
- Minor updates to the example project

### Backwards Incompatible Changes

- Previously the minimal version of jQuery was listed as 1.4.3 when it fact there was a bug a that made django-selectable require 1.4.4. Not a new incompatibility but the docs have now been updated and 1.4.3 compatibility will not be added. Thanks to Rick Testore for the report and the fix
- Started deprecation path for AutoComboboxSelectField and AutoComboboxSelectMultipleField

## v0.4.1 (Released 2012-03-11)

### Bug Fixes

- Cleaned up whitespace in css/js. Thanks Dan Poirier for the report and fix.
- Fixed issue with saving M2M field data with AutoCompleteSelectMultipleField. Thanks Raoul Thill for the report.

## v0.4.0 (Released 2012-02-25)

### Features

- Better compatibility with *AutoCompleteSelectWidget*/*AutoComboboxSelectWidget* and Django's ModelChoiceField
- Better compatibility with the Django admin *add another popup*
- Easier passing of query parameters. See the *Additional Parameters* section
- Additional documentation
- QUnit tests for JS functionality

### Backwards Incompatible Changes

- Support for `ModelLookup.search_field` string has been removed. You should use the `ModelLookup.search_fields` tuple instead.

## v0.3.1 (Released 2012-02-23)

### Bug Fixes

- Fixed issue with media urls when not using staticfiles.

## v0.3.0 (Released 2012-02-15)

### Features

- Multiple search fields for *model based lookups*
- Support for *highlighting term matches*
- Support for HTML in *result labels*
- Support for *client side formatting*
- Additional documentation
- Expanded examples in example project

### Bug Fixes

- Fixed issue with Enter key removing items from select multiple widgets #24

### Backwards Incompatible Changes

- The fix for #24 changed the remove items from a button to an anchor tag. If you were previously using the button tag for additional styling then you will need to adjust your styles.
- The static resources were moved into a *selectable* sub-directory. This makes the media more in line with the template directory conventions. If you are using the widgets in the admin there is nothing to change. If you are using `{{ form.media }}` then there is also nothing to change. However if you were including static media manually then you will need to adjust them to include the selectable prefix.

## v0.2.0 (Released 2011-08-13)

### Features

- Additional documentation
- *Positional configuration* for multiple select fields/widgets
- *Settings/configuration* for limiting/paginating result sets
- Compatibility and examples for *Admin inlines*
- JS updated for jQuery 1.6 compatibility
- *JS hooks* for updating query parameters
- *Chained selection example*

## v0.1.2 (Released 2011-05-25)

### Bug Fixes

- Fixed issue #17

## v0.1.1 (Release 2011-03-21)

### Bug Fixes

- Fixed/cleaned up multiple select fields and widgets
- Added media definitions to widgets

### Features

- Additional documentation
- Added *update_query_parameters* to widgets
- Refactored JS for easier configuration

## v0.1 (Released 2011-03-13)

Initial public release

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Index