Django-Select2 Documentation

Release 4.2.2

Nirupam Biswas

November 20, 2013

Contents

Contents:

CHAPTER 1

Get Started

1.1 Overview

This is a Django integration of Select2.

The app includes Select2 driven Django Widgets and Form Fields.

1.1.1 Widgets

These components are responsible for rendering the necessary JavaScript and HTML markups. Since this whole package is to render choices using Select2 JavaScript library, hence these components are meant to be used with choice fields.

Widgets are generally of two types :-

1. **Light** – They are not meant to be used when there are too many options, say, in thousands. This is because all those options would have to be pre-rendered onto the page and JavaScript would be used to search through them. Said that, they are also one the most easiest to use. They are almost drop-in-replacement for Django's default select widgets.

2. **Heavy** – They are suited for scenarios when the number of options are large and need complex queries (from maybe different sources) to get the options. This dynamic fetching of options undoubtedly requires Ajax communication with the server. Django-Select2 includes a helper JS file which is included automatically, so you need not worry about writing any Ajax related JS code. Although on the server side you do need to create a view specifically to respond to the queries.

   Heavies have further specialized versions called – **Auto Heavy**. These do not require views to server Ajax request. When they are instantiated, they register themselves with one central view which handles Ajax requests for them.

Heavy widgets have the word 'Heavy' in their name. Light widgets are normally named, i.e. there is no 'Light' word in their names.

Available widgets:

`Select2Widget`, `Select2MultipleWidget`, `HeavySelect2Widget`, `HeavySelect2MultipleWidget`, `AutoHeavySelect2Widget`, `AutoHeavySelect2MultipleWidget`, `HeavySelect2TagWidget`, `AutoHeavySelect2TagWidget`

Read more

1.1.2 Fields

These are pre-implemented choice fields which use the above widgets. It is highly recommended that you use them instead of rolling your own.

The fields available are good for general purpose use, although more specialized versions too are available for your ease.

Available fields:

`Select2ChoiceField`, `Select2MultipleChoiceField`, `HeavySelect2ChoiceField`, `HeavySelect2MultipleChoiceField`, `HeavyModelSelect2ChoiceField`, `HeavyModelSelect2MultipleChoiceField`, `ModelSelect2Field`, `ModelSelect2MultipleField`, `AutoSelect2Field`, `AutoSelect2MultipleField`, `AutoModelSelect2Field`, `AutoModelSelect2MultipleField`, `HeavySelect2TagField`, `AutoSelect2TagField`, `HeavyModelSelect2TagField`, `AutoModelSelect2TagField`

1.1.3 Views

The view - *Select2View*, exposed here is meant to be used with 'Heavy' fields and widgets.

Imported:

`Select2View`, `NO_ERR_RESP`

Read more

1.2 Installation

1. Install *django_select2*:

   ```
   pip install django_select2
   ```

2. Add *django_select2* to your *INSTALLED_APPS* in your project settings.
3. When deploying on production server, run:

   ```
   python manage.py collectstatic
   ```

4. Add *django_select* to your urlconf **if** you use any 'Auto' fields:

   ```
   url(r'^select2/', include('django_select2.urls')),
   ```

5. (Optionally) If you need multiple processes support, then:

   ```
   python manage.py syncdb
   ```

1.3 Available Settings

1.3.1 `AUTO_RENDER_SELECT2_STATICS` [Default `True`]

This, when specified and set to `False` in `settings.py` then Django_Select2 widgets won't automatically include the required scripts and stylesheets. When this setting is `True` then every Select2 field on the page will output `<script>` and `<link>` tags to include the required JS and CSS files. This is convenient but will output the same JS and CSS files multiple times if there are more than one Select2 fields on the page.

When this settings is `False` then you are responsible for including the JS and CSS files. To help you with this the following template tags are available in `django_select2_tags`.

- `import_django_select2_js` - Outputs `<script>` tags to include all the JS files, required by Light and Heavy widgets.
- `import_django_select2_css` - Outputs `<link>` tags to include all the CSS files, required by Light and Heavy widgets.
- `import_django_select2_js_css` - Outputs both `<script>` and `<link>` tags to include all the JS and CSS files, required by Light and Heavy widgets.

Tip: Make sure to include them at the top of the page, preferably in `<head>...</head>`.

Note: (Since version 3.3.1) The above template tags accept one argument `light`. Default value for that is `0`. If that is set to `1` then only the JS and CSS libraries needed by Select2Widget (Light fields) are rendered. That effectively leaves out `heavy.js` and `extra.css`.

1.3.2 `GENERATE_RANDOM_SELECT2_ID` [Default `False`]

As of version 4.0.0 the field's Ids are their paths which have been hashed by SHA1. This Id generation scheme should be sufficient for most applications.

However, if you have a secret government project and fear that SHA1 hashes could be cracked (which is not impossible) to reveal the path and names of your fields then you can enable this mode. This will use timestamps as Ids which have no correlation to the field's name or path.

Tip: The field's paths are first salted with Django generated `SECRET_KEY` before hashing them.

1.3.3 `ENABLE_SELECT2_MULTI_PROCESS_SUPPORT` [Default `False`]

This setting cannot be enabled as it is not required when `GENERATE_RANDOM_SELECT2_ID` is `False`.

In production servers usually multiple server processes are run to handle the requests. This poses a problem for Django Select2's Auto fields since they generate unique Id at runtime when `GENERATE_RANDOM_SELECT2_ID` is enabled. The clients can identify the fields in Ajax query request using only these generated ids. In multi-processes scenario there is no guarantee that the process which rendered the page is the one which will respond to Ajax queries.

When this mode is enabled then Django Select2 maintains an id to field key mapping in DB for all processes. Whenever a process does not find an id in its internal map it looks-up in the central DB. From DB it finds the field key. Using the key, the process then looks-up a field instance with that key, since all instances with same key are assumed to be equivalent.

Tip: Make sure to run `python manage.py syncdb` to create the `KeyMap` table.

Warning: You need to write your own script to periodically purge old data from `KeyMap` table. You can take help of `accessed_on` column. You need to decide the criteria on which basis you will purge the rows.

1.3.4 `SELECT2_MEMCACHE_HOST` [Default `None`], `SELECT2_MEMCACHE_PORT` [Default `None`], `SELECT2_MEMCACHE_TTL` [Default `900`]

When `ENABLE_SELECT2_MULTI_PROCESS_SUPPORT` is enabled then all processes will hit DB to get the mapping for the ids they are not aware of. For performance reasons it is recommended that you install Memcached and set the above settings appropriately.

Also note that, when you set the above you need to install `python-memcached` library too.

1.3.5 `SELECT2_BOOTSTRAP` [Default `False`]

Setting to True will include the CSS for making Select2 fit in with Bootstrap a bit better using the css found here https://github.com/fk/select2-bootstrap-css.

1.4 External Dependencies

- Django - This is obvious.
- jQuery - This is not included in the package since it is expected that in most scenarios this would already be available. The above template tags also won't output `<script>` tag to include this. You need to do this yourself.
- Memcached (python-memcached) - If you plan on running multiple Python processes, which is usually the case in production, then you need to turn on `ENABLE_SELECT2_MULTI_PROCESS_SUPPORT`. In that mode it is highly recommended that you use Memcached, to minimize DB hits.

1.5 Example Application

Please see `testapp` application. This application is used to manually test the functionalities of this package. This also serves as a good example.

You need only Django 1.4 or above to run that. It might run on older versions but that is not tested.

CHAPTER 2

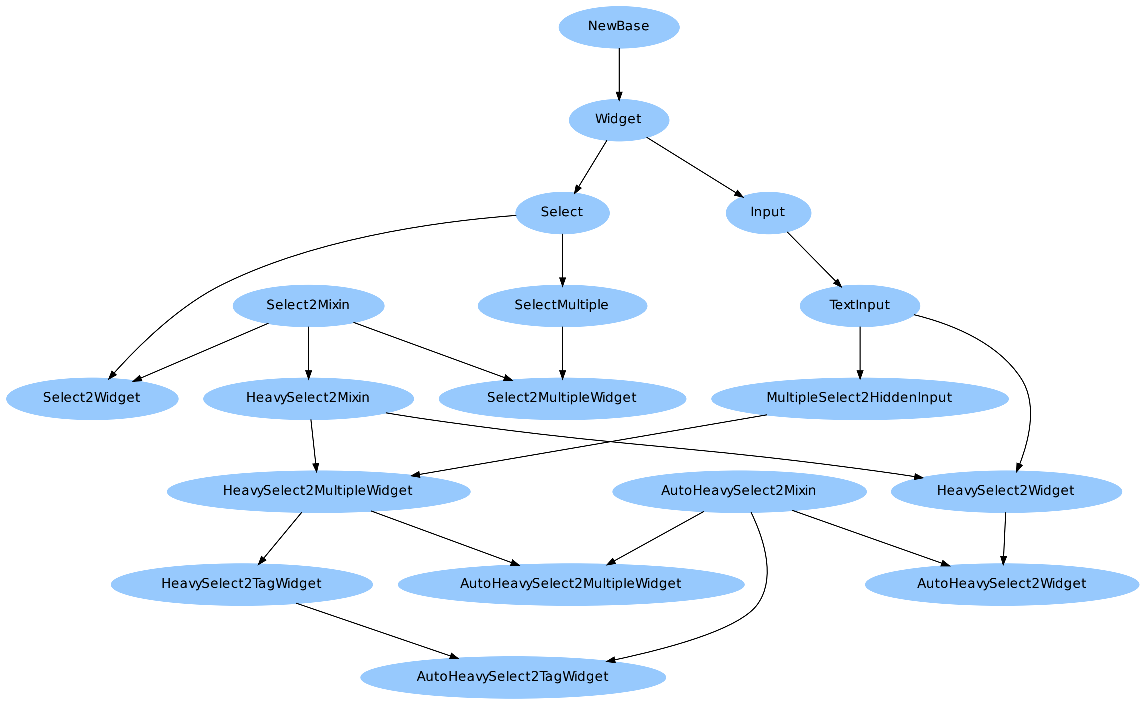
API Reference

The API references also include examples and suggestions, where relevant.

Contents:

2.1 Widgets

2.1.1 Class Diagram



2.1.2 Reference

Contains all the Django widgets for Select2.

class `django_select2.widgets.`**`Select2Mixin`**(***kwargs*)

Bases: `object`

The base mixin of all Select2 widgets.

This mixin is responsible for rendering the necessary JavaScript and CSS codes which turns normal `<select>` markups into Select2 choice list.

The following Select2 options are added by this mixin:-

- minimumResultsForSearch: `6`
- placeholder: ''
- allowClear: `True`
- multiple: `False`
- closeOnSelect: `False`

Note: Many of them would be removed by sub-classes depending on requirements.

`__init__`(***kwargs*)

Constructor of the class.

The following additional kwarg is allowed:-

Parameters **select2_options** (`dict` or None) – This is similar to standard Django way to pass extra attributes to widgets. This is meant to override values of existing `options`.

Example:

```
class MyForm(ModelForm):
    class Meta:
        model = MyModel
        widgets = {
            'name': Select2WidgetName(select2_options={
                'minimumResultsForSearch': 10,
                'closeOnSelect': True,
                })
        }
```

Tip: You cannot introduce new options using this. For that you should sub-class and overried `init_options()`. The reason for this is, few options are not compatible with each other or are not applicable in some scenarios. For example, when Select2 is attached to `<select>` tag, it can get if it is multiple or single valued from that tag itself. In this case if you specify `multiple` option then not only it is useless but an error in Select2 JS' point of view.

There are other such intricacies, based on which some options are removed. By enforcing this restriction we make sure to not break the code by passing some wrong concotion of options.

`options` = **{'minimumResultsForSearch': 6, 'allowClear': True, 'closeOnSelect': False, 'placeholder': '', 'multiple': Fal**

The options listed here are rendered as JS map and passed to Select2 JS code. Complete description of theses options are available in Select2 JS' site.

`init_options`()

Sub-classes can use this to suppress or override options passed to Select2 JS library.

Example:

```
def init_options(self):
    self.options['createSearchChoice'] = JSFunction('Your_js_function')
```

In the above example we are setting `Your_js_function` as Select2's `createSearchChoice` function.

Tip: If you want to run `Your_js_function` in the context of the Select2 DOM element, i.e. `this` inside your JS function should point to the component instead of `window`, then use `JSFunctionInContext` instead of `JSFunction`.

`set_placeholder`(*val*)

Placeholder is a value which Select2 JS library shows when nothing is selected. This should be string.

Returns None

`get_options`()

Returns Dictionary of options to be passed to Select2 JS.

Return type `dict`

`render_select2_options_code`(*options*, *id_*)
Renders options for Select2 JS.

Returns The rendered JS code.

Return type `unicode`

`render_js_code`(*id_*, **args*)
Renders the `<script>` block which contains the JS code for this widget.

Returns The rendered JS code enclosed inside `<script>` block.

Return type `unicode`

`render_inner_js_code`(*id_*, **args*)
Renders all the JS code required for this widget.

Returns The rendered JS code which will be later enclosed inside `<script>` block.

Return type `unicode`

`render`(*name*, *value*, *attrs=None*, *choices=()*)
Renders this widget. HTML and JS code blocks all are rendered by this.

Returns The rendered markup.

Return type `unicode`

class `django_select2.widgets.`**`Select2Widget`**(***kwargs*)
Bases: `django_select2.widgets.Select2Mixin`, `django.forms.widgets.Select`

Drop-in Select2 replacement for `forms.Select`.

Following Select2 option from `Select2Mixin.options` is removed:-

- multiple

class `django_select2.widgets.`**`Select2MultipleWidget`**(***kwargs*)
Bases: `django_select2.widgets.Select2Mixin`, `django.forms.widgets.SelectMultiple`

Drop-in Select2 replacement for `forms.SelectMultiple`.

Following Select2 options from `Select2Mixin.options` are removed:-

- multiple
- allowClear
- minimumResultsForSearch

class `django_select2.widgets.`**`MultipleSelect2HiddenInput`**(*attrs=None*)
Bases: `django.forms.widgets.TextInput`

Multiple hidden input for Select2.

This is a specialized multiple Hidden Input widget. This includes a special JS component which renders multiple Hidden Input boxes as there are values. So, if user suppose chooses values 1, 4 and 9 then Select2 would would write them to the primary hidden input. The JS component of this widget will read that value and will render three more hidden input boxes each with values 1, 4 and 9 respectively. They will all share the name of this field, and the name of the primary source hidden input would be removed. This way, when submitted all the selected values would be available as list.

class `django_select2.widgets.`**`HeavySelect2Mixin`**(***kwargs*)
Bases: `django_select2.widgets.Select2Mixin`

The base mixin of all Heavy Select2 widgets. It sub-classes `Select2Mixin`.

This mixin adds more Select2 options to `Select2Mixin.options`. These are:-

- minimumInputLength: `2`
- initSelection: `JSFunction('django_select2.onInit')`
- **ajax:**
  - dataType: `'json'`
  - quietMillis: `100`
  - data: `JSFunctionInContext('django_select2.get_url_params')`
  - results: `JSFunctionInContext('django_select2.process_results')`

Tip: You can override these options by passing `select2_options` kwarg to `__init__()`.

`__init__`(***kwargs*)

Constructor of the class.

The following kwargs are allowed:-

Parameters

- **data_view** (`django.views.generic.base.View` or None) – A `Select2View` sub-class which can respond to this widget's Ajax queries.
- **data_url** (`str` or None) – Url which will respond to Ajax queries with JSON object.

Tip: When `data_view` is provided then it is converted into Url using `reverse()`.

Warning: Either of `data_view` or `data_url` must be specified, else `ValueError` would be raised.

Parameters

- **choices** (`list` or `tuple`) – The list of available choices. If not provided then empty list is used instead. It should be of the form – `[(val1, 'Label1'), (val2, 'Label2'), ...]`.
- **userGetValTextFuncName** (`str`) – The name of the custom JS function which you want to use to convert value to label.

  In `heavy_data.js`, `django_select2.getValText()` employs the following logic to convert value to label :-

  1. First check if the Select2 input field has `txt` attribute set along with `value`. If found then use it.
  2. Otherwise, check if user has provided any custom method for this. Then use that. If it returns a label then use it.
  3. Otherwise, check the cached results. When the user searches in the fields then all the returned responses from server, which has the value and label mapping, are cached by `heavy_data.js`.

> **Tip:** Since version 3.2.0, cookies or localStorage are no longer checked or used. All `HeavyChoiceField` must override `get_val_txt()`. If you are only using heavy widgets in your own fields then you should override `render_texts()`.

`render_texts`(*selected_choices*, *choices*)

Renders a JS array with labels for the `selected_choices`.

Parameters

- **selected_choices** (`list` or `tuple`) – List of selected choices' values.
- **choices** (`list` or `tuple`) – Extra choices, if any. This is a list of tuples. In each tuple, the first item is the choice value and the second item is choice label.

Returns The rendered JS array code.

Return type `unicode`

`render_texts_for_value`(*id_*, *value*, *choices*)

Renders the JS code which sets the `txt` attribute on the field. It gets the array of lables from `render_texts()`.

Parameters

- **id** (`str`) – Id of the field. This can be used to get reference of this field's DOM in JS.
- **value** (*Any*) – Currently set value on the field.
- **choices** (`list` or `tuple`) – Extra choices, if any. This is a list of tuples. In each tuple, the first item is the choice value and the second item is choice label.

Returns JS code which sets the `txt` attribute.

Return type `unicode`

class `django_select2.widgets.`**`HeavySelect2Widget`**(***kwargs*)

Bases: `django_select2.widgets.HeavySelect2Mixin`, `django.forms.widgets.TextInput`

Single selection heavy widget.

Following Select2 option from `Select2Mixin.options` is added or set:-

- multiple: `False`

class `django_select2.widgets.`**`HeavySelect2MultipleWidget`**(***kwargs*)

Bases: `django_select2.widgets.HeavySelect2Mixin`, `django_select2.widgets.MultipleSelect2Hi`

Multiple selection heavy widget.

Following Select2 options from `Select2Mixin.options` are removed:-

- allowClear
- minimumResultsForSearch

Following Select2 options from `Select2Mixin.options` are added or set:-

- multiple: `True`
- separator: `JSVar('django_select2.MULTISEPARATOR')`

`render_texts_for_value`(*id_*, *value*, *choices*)

Renders the JS code which sets the `txt` attribute on the field. It gets the array of lables from `render_texts()`.

Parameters

- **id** (`str`) – Id of the field. This can be used to get reference of this field's DOM in JS.
- **value** (`list`) – **List** of currently set value on the field.
- **choices** (`list` or `tuple`) – Extra choices, if any. This is a list of tuples. In each tuple, the first item is the choice value and the second item is choice label.

Returns JS code which sets the `txt` attribute.

Return type `unicode`

class `django_select2.widgets.`**`HeavySelect2TagWidget`**`(`*`**kwargs`*`)`

Bases: `django_select2.widgets.HeavySelect2MultipleWidget`

Heavy widget with tagging support. Based on `HeavySelect2MultipleWidget`, unlike other widgets this allows users to create new options (tags).

Following Select2 options from `Select2Mixin.options` are removed:-

- allowClear
- minimumResultsForSearch
- closeOnSelect

Following Select2 options from `Select2Mixin.options` are added or set:-

- multiple: `True`
- separator: `JSVar('django_select2.MULTISEPARATOR')`
- tags: `True`
- tokenSeparators: `,` and ‘‘ ‘‘
- createSearchChoice: `JSFunctionInContext('django_select2.createSearchChoice')`
- minimumInputLength: `1`

class `django_select2.widgets.`**`AutoHeavySelect2Mixin`**`(`*`*args`*, *`**kwargs`*`)`

Bases: `object`

This mixin is needed for Auto heavy fields.

This mxin adds extra JS code to notify the field's DOM object of the generated id. The generated id is not the same as the `id` attribute of the field's HTML markup. This id is generated by `register_field()` when the Auto field is registered. The client side (DOM) sends this id along with the Ajax request, so that the central view can identify which field should be used to serve the request.

class `django_select2.widgets.`**`AutoHeavySelect2Widget`**`(`*`*args`*, *`**kwargs`*`)`

Bases: `django_select2.widgets.AutoHeavySelect2Mixin`, `django_select2.widgets.HeavySelect2Wi`

Auto version of `HeavySelect2Widget`

class `django_select2.widgets.`**`AutoHeavySelect2MultipleWidget`**`(`*`*args`*, *`**kwargs`*`)`

Bases: `django_select2.widgets.AutoHeavySelect2Mixin`, `django_select2.widgets.HeavySelect2Mu`

Auto version of `HeavySelect2MultipleWidget`

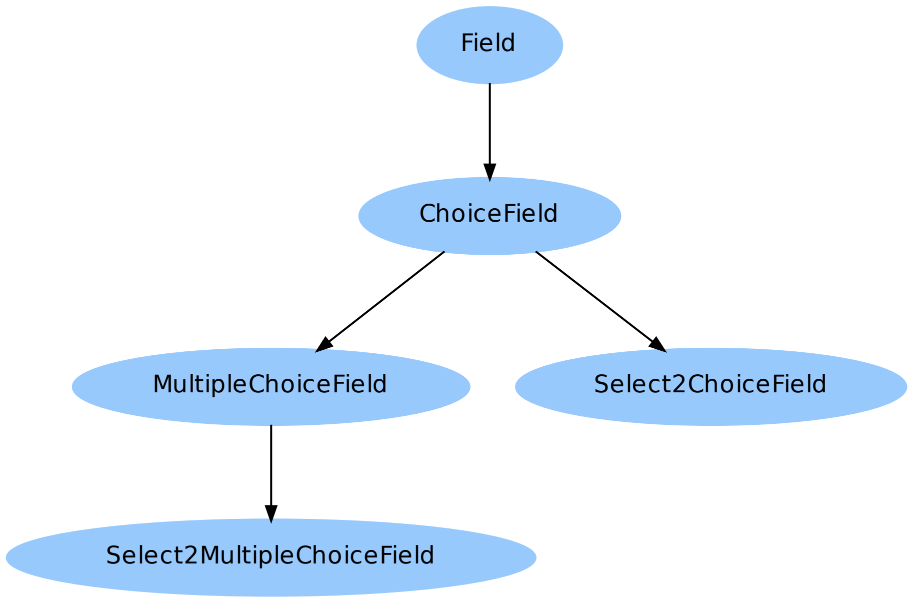
class `django_select2.widgets.`**`AutoHeavySelect2TagWidget`**`(`*`*args`*, *`**kwargs`*`)`

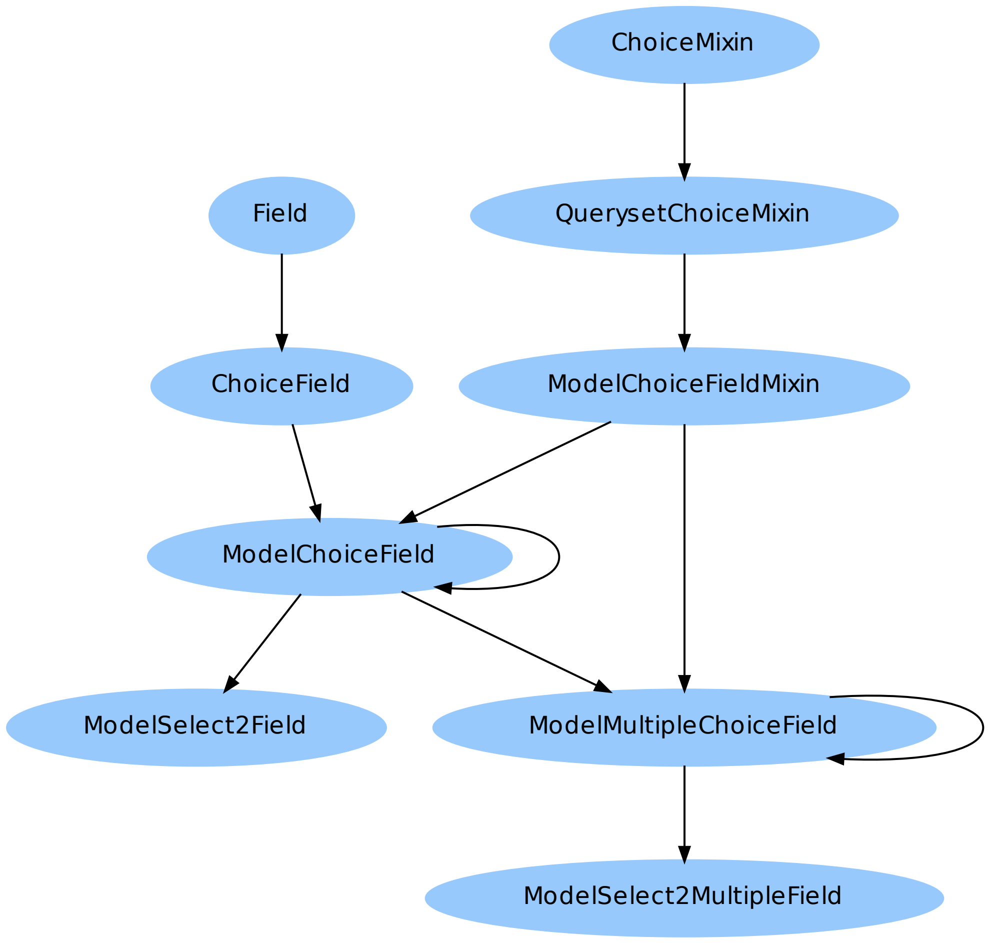
Bases: `django_select2.widgets.AutoHeavySelect2Mixin`, `django_select2.widgets.HeavySelect2Ta`

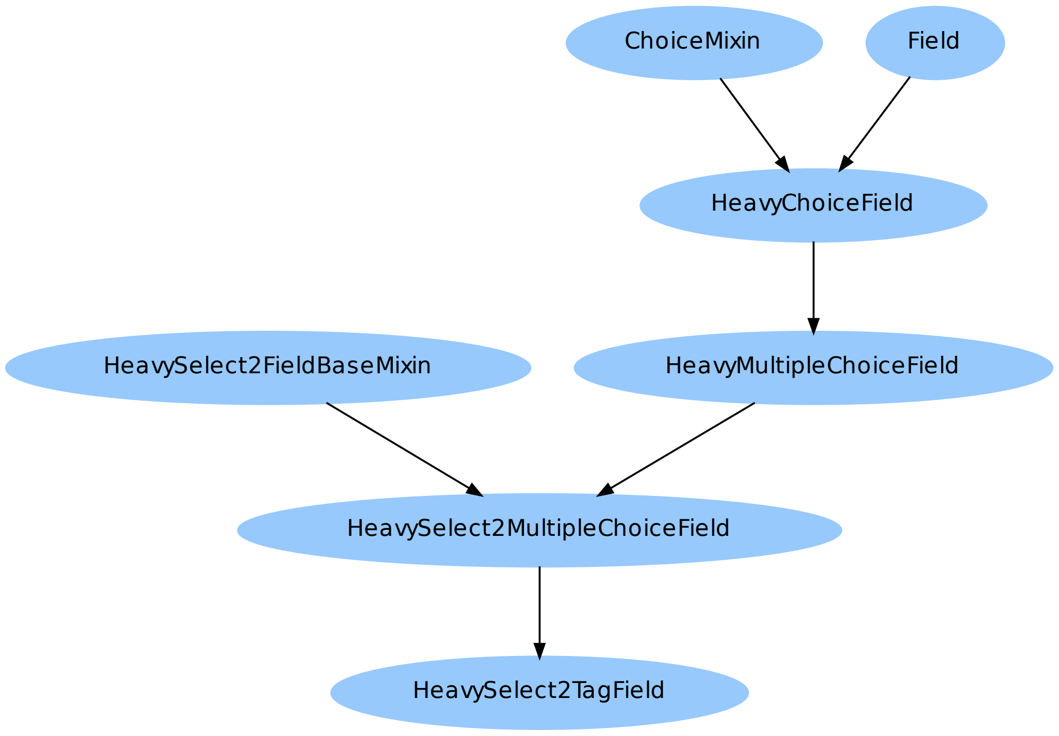
Auto version of `HeavySelect2TagWidget`

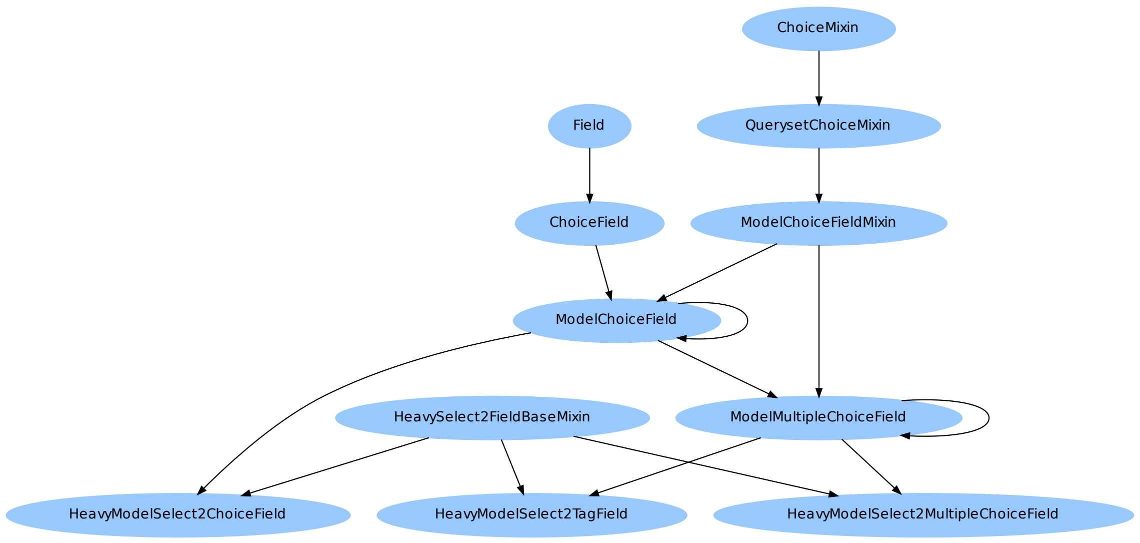
2.2 Fields

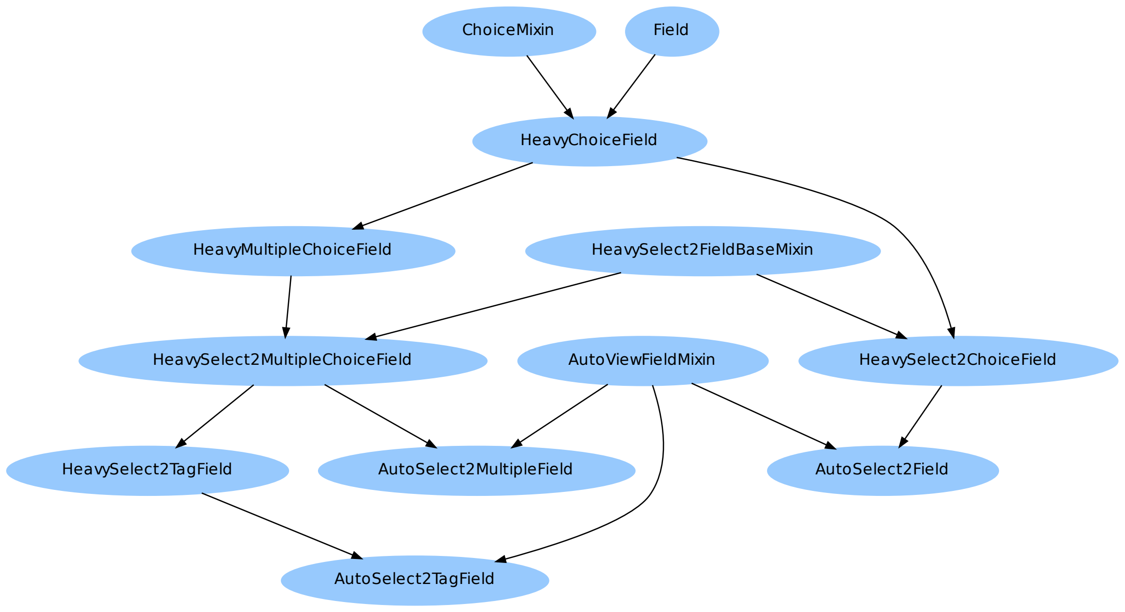
2.2.1 Class Diagrams

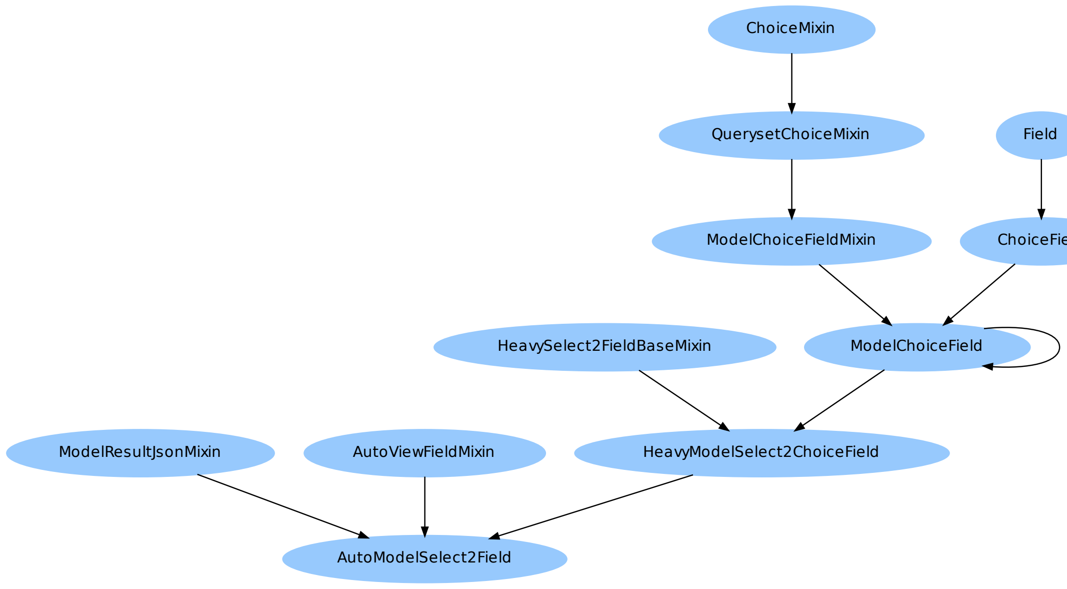


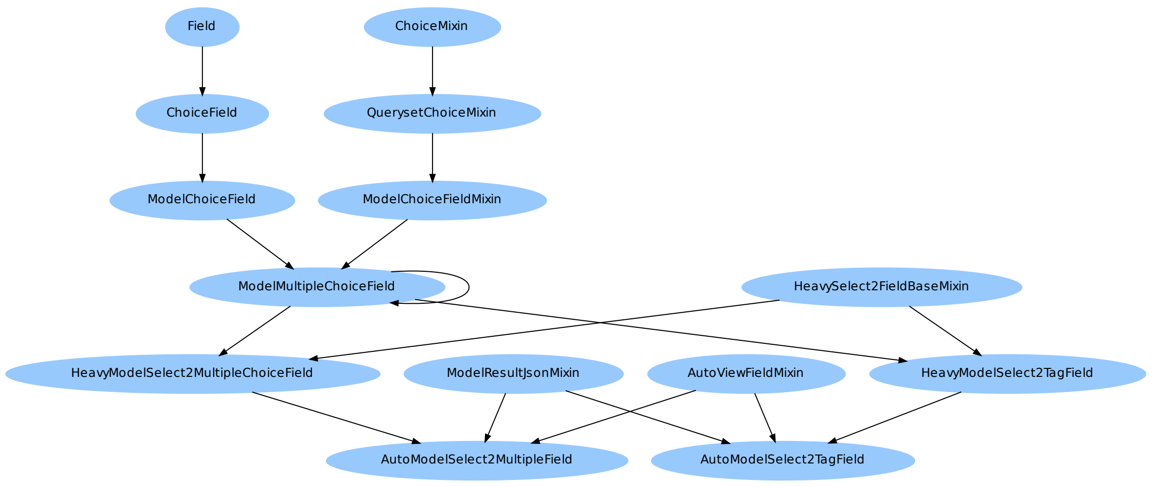

ChoiceMixin
Field
QuerysetChoiceMixin
ChoiceField
ModelChoiceFieldMixin
ModelChoiceField
ModelSelect2Field
ModelMultipleChoiceField
ModelSelect2MultipleField


ChoiceMixin
Field
HeavyChoiceField
HeavySelect2FieldBaseMixin
HeavyMultipleChoiceField
HeavySelect2MultipleChoiceField
HeavySelect2TagField


ChoiceMixin
Field
QuerysetChoiceMixin
ChoiceField
ModelChoiceFieldMixin
ModelChoiceField
HeavySelect2FieldBaseMixin
ModelMultipleChoiceField
HeavyModelSelect2ChoiceField
HeavyModelSelect2TagField
HeavyModelSelect2MultipleChoiceField


ChoiceMixin
Field
HeavyChoiceField
HeavyMultipleChoiceField
HeavySelect2FieldBaseMixin
HeavySelect2MultipleChoiceField
AutoViewFieldMixin
HeavySelect2ChoiceField
HeavySelect2TagField
AutoSelect2MultipleField
AutoSelect2Field
AutoSelect2TagField


ChoiceMixin
QuerysetChoiceMixin
Field
ModelChoiceFieldMixin
ChoiceField
HeavySelect2FieldBaseMixin
ModelChoiceField
ModelResultJsonMixin
AutoViewFieldMixin
HeavyModelSelect2ChoiceField
AutoModelSelect2Field



2.2.2 Reference

Contains all the Django fields for Select2.

class `django_select2.fields.`**`AutoViewFieldMixin`**`(`**args*, ***kwargs*`)`

Bases: `object`

Registers itself with AutoResponseView.

All Auto fields must sub-class this mixin, so that they are registered.

Warning: Do not forget to include `'django_select2.urls'` in your url conf, else, central view used to serve Auto fields won't be available.

`__init__``(`**args*, ***kwargs*`)`

Class constructor.

Parameters **auto_id** (`unicode`) – The key to use while registering this field. If it is not provided then an auto generated key is used.

Tip: This mixin uses full class name of the field to register itself. This is used like key in a `dict` by `util.register_field()`.

If that key already exists then the instance is not registered again. So, eventually all instances of an Auto field share one instance to respond to the Ajax queries for its fields.

If for some reason any instance needs to be isolated then `auto_id` can be used to provide a unique key which has never occured before.

`security_check``(`*request*, **args*, ***kwargs*`)`

Returns `False` if security check fails.

Parameters

- **request** (`django.http.HttpRequest`) – The Ajax request object.
- **args** – The `*args` passed to `django.views.generic.base.View.dispatch()`.

- **kwargs** – The `**kwargs` passed to `django.views.generic.base.View.dispatch()`.

Returns A boolean value, signalling if check passed or failed.

Return type `bool`

> **Warning:** Sub-classes should override this. You really do not want random people making Http reqeusts to your server, be able to get access to sensitive information.

get_results(*request*, *term*, *page*, *context*)
See `views.Select2View.get_results()`.

class `django_select2.fields.`**Select2ChoiceField**(*choices=()*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=None*, **args*, ***kwargs*)

Bases: `django.forms.fields.ChoiceField`

Drop-in Select2 replacement for `forms.ChoiceField`.

widget
alias of `Select2Widget`

class `django_select2.fields.`**Select2MultipleChoiceField**(*choices=()*, *required=True*, *widget=None*, *label=None*, *initial=None*, *help_text=None*, **args*, ***kwargs*)

Bases: `django.forms.fields.MultipleChoiceField`

Drop-in Select2 replacement for `forms.MultipleChoiceField`.

widget
alias of `Select2MultipleWidget`

class `django_select2.fields.`**ModelResultJsonMixin**(**args*, ***kwargs*)
Bases: `object`

Makes `heavy_data.js` parsable JSON response for queries on its model.

On query it uses `prepare_qs_params()` to prepare query attributes which it then passes to `self.queryset.filter()` to get the results.

It is expected that sub-classes will defined a class field variable `search_fields`, which should be a list of field names to search for.

..note:: As of version 3.1.3, `search_fields` is optional if sub-class overrides `get_results`.

__init__(**args*, ***kwargs*)
Class constructor.

Parameters

- **queryset** (`django.db.models.query.QuerySet` or None) – This can be passed as kwarg here or defined as field variabel, like `search_fields`.
- **max_results** (`int`) – Maximum number to results to return per Ajax query.
- **to_field_name** (`str`) – Which field's value should be returned as result tuple's value. (Default is `pk`, i.e. the id field of the model)

get_queryset()
Returns the queryset.

The default implementation returns the `self.queryset`, which is usually the one set by sub-classes at class-level. However, if that is `None` then `ValueError` is thrown.

Returns queryset

Return type `django.db.models.query.QuerySet`

`label_from_instance`(*obj*)
Sub-classes should override this to generate custom label texts for values.

Parameters **obj** (`django.model.Model`) – The model object.

Returns The label string.

Return type `unicode`

`extra_data_from_instance`(*obj*)
Sub-classes should override this to generate extra data for values. These are passed to JavaScript and can be used for custom rendering.

Parameters **obj** (`django.model.Model`) – The model object.

Returns The extra data dictionary.

Return type `dict`

`prepare_qs_params`(*request*, *search_term*, *search_fields*)
Prepares queryset parameter to use for searching.

Parameters

- **search_term** (`list`) – The search term.
- **search_fields** – The list of search fields. This is same as `self.search_fields`.

Returns

A dictionary of parameters to 'or' and 'and' together. The output format should be

```
{
    'or': [
    Q(attr11=term11) | Q(attr12=term12) | ...,
    Q(attrN1=termN1) | Q(attrN2=termN2) | ...,
    ...],

    'and': {
        'attrX1': termX1,
        'attrX2': termX2,
        ...
    }
}
```

The above would then be coaxed into `filter()` as below:

```
queryset.filter(
    Q(attr11=term11) | Q(attr12=term12) | ...,
    Q(attrN1=termN1) | Q(attrN2=termN2) | ...,
    ...,
    attrX1=termX1,
    attrX2=termX2,
    ...
    )
```

In this implementation, `term11, term12, termN1, ...` etc., all are actually `search_term`. Also then `and` part is always empty.

So, let's take an example.

Assume, `search_term == 'John'`

`self.search_fields == ['first_name__icontains', 'last_name__icontains']`

So, the prepared query would be:

```
{
    'or': [
        Q(first_name__icontains=search_term) | Q(last_name__icontains=search_term)
    ],
    'and': {}
}
```

Return type `dict`

`get_results`(*request*, *term*, *page*, *context*)

See `views.Select2View.get_results()`.

This implementation takes care of detecting if more results are available.

class `django_select2.fields.`**`UnhideableQuerysetType`**

Bases: `type`

This does some pretty nasty hacky stuff, to make sure users can also define `queryset` as class-level field variable, instead of passing it to constructor.

class `django_select2.fields.`**`ChoiceMixin`**

Bases: `object`

Simple mixin which provides a property – `choices`. When `choices` is set, then it sets that value to `self.widget.choices` too.

class `django_select2.fields.`**`FilterableModelChoiceIterator`**(*field*)

Bases: `django.forms.models.ModelChoiceIterator`

Extends ModelChoiceIterator to add the capability to apply additional filter on the passed queryset.

`set_extra_filter`(***filter_map*)

Applies additional filter on the queryset. This can be called multiple times.

Parameters **filter_map** – The `**kwargs` to pass to `django.db.models.query.QuerySet.filter()`. If this is not set then additional filter (if) applied before is removed.

class `django_select2.fields.`**`QuerysetChoiceMixin`**

Bases: `django_select2.fields.ChoiceMixin`

Overrides `choices`‘ getter to return instance of `FilterableModelChoiceIterator` instead.

class `django_select2.fields.`**`ModelSelect2Field`**(**args*, ***kwargs*)

Bases: `django_select2.fields.ModelChoiceField`

Light Select2 field, specialized for Models.

Select2 replacement for `forms.ModelChoiceField`.

`widget`

alias of `Select2Widget`

class `django_select2.fields.`**`ModelSelect2MultipleField`**(**args*, ***kwargs*)

Bases: `django_select2.fields.ModelMultipleChoiceField`

Light multiple-value Select2 field, specialized for Models.

Select2 replacement for `forms.ModelMultipleChoiceField`.

`widget`
alias of `Select2MultipleWidget`

class `django_select2.fields.`**`HeavySelect2FieldBaseMixin`**`(`**args*, ***kwargs*`)`
Bases: `object`

Base mixin field for all Heavy fields.

Note: Although Heavy fields accept `choices` parameter like all Django choice fields, but these fields are backed by big data sources, so `choices` cannot possibly have all the values.

For Heavies, consider `choices` to be a subset of all possible choices. It is available because users might expect it to be available.

`__init__``(`**args*, ***kwargs*`)`
Class constructor.

Parameters

- **data_view** (`django.views.generic.base.View` or None) – A `Select2View` sub-class which can respond to this widget's Ajax queries.
- **widget** (`django.forms.widgets.Widget` or None) – A widget instance.

Warning: Either of `data_view` or `widget` must be specified, else `ValueError` would be raised.

class `django_select2.fields.`**`HeavyChoiceField`**`(`**args*, ***kwargs*`)`
Bases: `django_select2.fields.ChoiceMixin`, `django.forms.fields.Field`

Reimplements `django.forms.TypedChoiceField` in a way which suites the use of big data.

Note: Although this field accepts `choices` parameter like all Django choice fields, but these fields are backed by big data sources, so `choices` cannot possibly have all the values. It is meant to be a subset of all possible choices.

`empty_value` **= u''**
Sub-classes can set this other value if needed.

`coerce_value``(`*value*`)`
Coerces `value` to a Python data type.

Sub-classes should override this if they do not want Unicode values.

`validate_value``(`*value*`)`
Sub-classes can override this to validate the value entered against the big data.

Parameters **value** (As coerced by `coerce_value()`.) – Value entered by the user.

Returns `True` means the `value` is valid.

`get_val_txt``(`*value*`)`
If Heavy widgets encounter any value which it can't find in `choices` then it calls this method to get the label for the value.

Parameters **value** (As coerced by `coerce_value()`.) – Value entered by the user.

Returns The label for this value.

Return type `unicode` or None (when no possible label could be found)

class `django_select2.fields.`**`HeavyMultipleChoiceField`**`(`**args*, ***kwargs*`)`

Bases: `django_select2.fields.HeavyChoiceField`

Reimplements `django.forms.TypedMultipleChoiceField` in a way which suites the use of big data.

Note: Although this field accepts `choices` parameter like all Django choice fields, but these fields are backed by big data sources, so `choices` cannot possibly have all the values. It is meant to be a subset of all possible choices.

`hidden_widget`

alias of `MultipleHiddenInput`

class `django_select2.fields.`**`HeavySelect2ChoiceField`**`(`**args*, ***kwargs*`)`

Bases: `django_select2.fields.HeavySelect2FieldBaseMixin`, `django_select2.fields.HeavyChoiceField`

Heavy Select2 Choice field.

`widget`

alias of `HeavySelect2Widget`

class `django_select2.fields.`**`HeavySelect2MultipleChoiceField`**`(`**args*, ***kwargs*`)`

Bases: `django_select2.fields.HeavySelect2FieldBaseMixin`, `django_select2.fields.HeavyMultipleChoiceField`

Heavy Select2 Multiple Choice field.

`widget`

alias of `HeavySelect2MultipleWidget`

class `django_select2.fields.`**`HeavySelect2TagField`**`(`**args*, ***kwargs*`)`

Bases: `django_select2.fields.HeavySelect2MultipleChoiceField`

Heavy Select2 field for tagging.

Warning: `NotImplementedError` would be thrown if `create_new_value()` is not implemented.

`widget`

alias of `HeavySelect2TagWidget`

`create_new_value``(`*value*`)`

This is called when the input value is not valid. This allows you to add the value into the data-store. If that is not done then eventually the validation will fail.

Parameters **value** (As coerced by `HeavyChoiceField.coerce_value()`.) – Invalid value entered by the user.

Returns The a new value, which could be the id (pk) of the created value.

Return type Any

class `django_select2.fields.`**`HeavyModelSelect2ChoiceField`**`(`**args*, ***kwargs*`)`

Bases: `django_select2.fields.HeavySelect2FieldBaseMixin`, `django_select2.fields.ModelChoiceField`

Heavy Select2 Choice field, specialized for Models.

`widget`

alias of `HeavySelect2Widget`

class `django_select2.fields.`**`HeavyModelSelect2MultipleChoiceField`**(**args*, ***kwargs*)

Bases: `django_select2.fields.HeavySelect2FieldBaseMixin`, `django_select2.fields.ModelMultipleChoiceField`

Heavy Select2 Multiple Choice field, specialized for Models.

`widget`
alias of `HeavySelect2MultipleWidget`

class `django_select2.fields.`**`HeavyModelSelect2TagField`**(**args*, ***kwargs*)

Bases: `django_select2.fields.HeavySelect2FieldBaseMixin`, `django_select2.fields.ModelMultipleChoiceField`

Heavy Select2 field for tagging, specialized for Models.

> **Warning:** `NotImplementedError` would be thrown if `get_model_field_values()` is not implemented.

`widget`
alias of `HeavySelect2TagWidget`

`create_new_value`(*value*)
This is called when the input value is not valid. This allows you to add the value into the data-store. If that is not done then eventually the validation will fail.

Parameters **value** (As coerced by `HeavyChoiceField.coerce_value()`.) – Invalid value entered by the user.

Returns The a new value, which could be the id (pk) of the created value.

Return type Any

`get_model_field_values`(*value*)
This is called when the input value is not valid and the field tries to create a new model instance.

Parameters **value** (*unicode*) – Invalid value entered by the user.

Returns Dict with attribute name - attribute value pair.

Return type dict

class `django_select2.fields.`**`AutoSelect2Field`**(**args*, ***kwargs*)

Bases: `django_select2.fields.AutoViewFieldMixin`, `django_select2.fields.HeavySelect2ChoiceF`

Auto Heavy Select2 field.

This needs to be subclassed. The first instance of a class (sub-class) is used to serve all incoming json query requests for that type (class).

> **Warning:** `NotImplementedError` would be thrown if `get_results()` is not implemented.

`widget`
alias of `AutoHeavySelect2Widget`

class `django_select2.fields.`**`AutoSelect2MultipleField`**(**args*, ***kwargs*)

Bases: `django_select2.fields.AutoViewFieldMixin`, `django_select2.fields.HeavySelect2Multipl`

Auto Heavy Select2 field for multiple choices.

This needs to be subclassed. The first instance of a class (sub-class) is used to serve all incoming json query requests for that type (class).

> **Warning:** `NotImplementedError` would be thrown if `get_results()` is not implemented.

`widget`
alias of `AutoHeavySelect2MultipleWidget`

class `django_select2.fields.`**`AutoSelect2TagField`**`(`**args*, ***kwargs*`)`
Bases: `django_select2.fields.AutoViewFieldMixin`, `django_select2.fields.HeavySelect2TagFiel`

Auto Heavy Select2 field for tagging.

This needs to be subclassed. The first instance of a class (sub-class) is used to serve all incoming json query requests for that type (class).

> **Warning:** `NotImplementedError` would be thrown if `get_results()` is not implemented.

`widget`
alias of `AutoHeavySelect2TagWidget`

class `django_select2.fields.`**`AutoModelSelect2Field`**`(`**args*, ***kwargs*`)`
Bases: `django_select2.fields.ModelResultJsonMixin`, `django_select2.fields.AutoViewFieldMixi` `django_select2.fields.HeavyModelSelect2ChoiceField`

Auto Heavy Select2 field, specialized for Models.

This needs to be subclassed. The first instance of a class (sub-class) is used to serve all incoming json query requests for that type (class).

`widget`
alias of `AutoHeavySelect2Widget`

class `django_select2.fields.`**`AutoModelSelect2MultipleField`**`(`**args*, ***kwargs*`)`
Bases: `django_select2.fields.ModelResultJsonMixin`, `django_select2.fields.AutoViewFieldMixi` `django_select2.fields.HeavyModelSelect2MultipleChoiceField`

Auto Heavy Select2 field for multiple choices, specialized for Models.

This needs to be subclassed. The first instance of a class (sub-class) is used to serve all incoming json query requests for that type (class).

`widget`
alias of `AutoHeavySelect2MultipleWidget`

class `django_select2.fields.`**`AutoModelSelect2TagField`**`(`**args*, ***kwargs*`)`
Bases: `django_select2.fields.ModelResultJsonMixin`, `django_select2.fields.AutoViewFieldMixi` `django_select2.fields.HeavyModelSelect2TagField`

Auto Heavy Select2 field for tagging, specialized for Models.

This needs to be subclassed. The first instance of a class (sub-class) is used to serve all incoming json query requests for that type (class).

> **Warning:** `NotImplementedError` would be thrown if `get_model_field_values()` is not implemented.

Example:

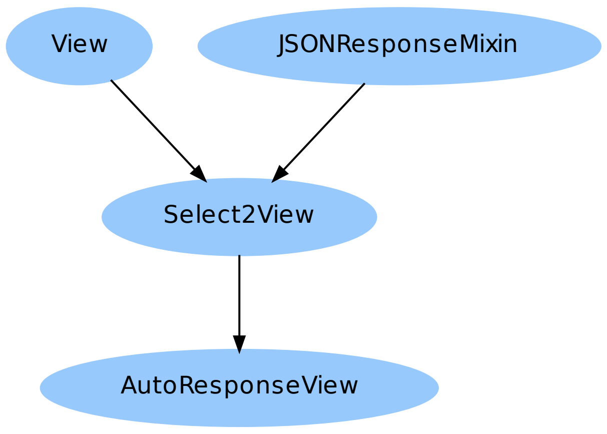
```python
class Tag(models.Model):
    tag = models.CharField(max_length=10, unique=True)
    def __unicode__(self):
        return unicode(self.tag)
```

```
class TagField(AutoModelSelect2TagField):
    queryset = Tag.objects
    search_fields = ['tag__icontains', ]
    def get_model_field_values(self, value):
        return {'tag': value}
```

widget
alias of `AutoHeavySelect2TagWidget`

2.3 Views

2.3.1 Class Diagram



2.3.2 Reference

`django_select2.views.`**NO_ERR_RESP = 'nil'**
Equals to 'nil' constant.

Use this in `Select2View.get_results()` to mean no error, instead of hardcoding 'nil' value.

class `django_select2.views.`**JSONResponseMixin**
Bases: `object`

A mixin that can be used to render a JSON response.

response_class
alias of `HttpResponse`

render_to_response(*context*, ***response_kwargs*)
Returns a JSON response, transforming 'context' to make the payload.

convert_context_to_json(*context*)
Convert the context dictionary into a JSON object

class `django_select2.views.`**`Select2View`**`(`*`**kwargs`*`)`

Bases: `django_select2.views.JSONResponseMixin`, `django.views.generic.base.View`

Base view which is designed to respond with JSON to Ajax queries from heavy widgets/fields.

Although the widgets won't enforce the type of data_view it gets, but it is recommended to sub-class this view instead of creating a Django view from scratch.

Note: Only GET Http requests are supported.

`respond_with_exception``(`*`e`*`)`

Parameters **e** (*Exception*) – Exception object.

Returns Response with status code of 404 if e is `Http404` object, else 400.

Return type HttpResponse

`check_all_permissions``(`*`request`*, *`*args`*, *`**kwargs`*`)`

Sub-classes can use this to raise exception on permission check failures, or these checks can be placed in `urls.py`, e.g. `login_required(SelectClass.as_view())`.

Parameters

- **request** (`django.http.HttpRequest`) – The Ajax request object.
- **args** – The `*args` passed to `django.views.generic.View.dispatch()`.
- **kwargs** – The `**kwargs` passed to `django.views.generic.View.dispatch()`.

Warning: Sub-classes should override this. You really do not want random people making Http reqeusts to your server, be able to get access to sensitive information.

`get_results``(`*`request`*, *`term`*, *`page`*, *`context`*`)`

Returns the result for the given search `term`.

Parameters

- **request** (`django.http.HttpRequest`) – The Ajax request object.
- **term** (`str`) – The search term.
- **page** (`int`) – The page number. If in your last response you had signalled that there are more results, then when user scrolls more a new Ajax request would be sent for the same term but with next page number. (Page number starts at 1)
- **context** (`str` or None) – Can be anything which persists across the lifecycle of queries for the same search term. It is reset to `None` when the term changes.

  Note: Currently this is not used by `heavy_data.js`.

Expected output is of the form:

```
(err, has_more, [results])
```

Where `results = [(id1, text1), (id2, text2), ...]`

For example:

```
('nil', False,
    [
    (1, 'Value label1'),
    (20, 'Value label2'),
    ])
```

When everything is fine then the *err* must be 'nil'. *has_more* should be true if there are more rows.

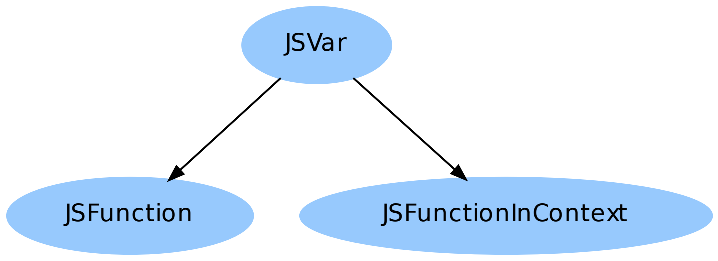
class `django_select2.views.`**`AutoResponseView`**`(`***kwargs*`)`
Bases: `django_select2.views.Select2View`

A central view meant to respond to Ajax queries for all Heavy widgets/fields. Although it is not mandatory to use, but is immensely helpful.

Tip: Fields which want to use this view must sub-class `AutoViewFieldMixin`.

2.4 Util

2.4.1 Class Diagram



2.4.2 Reference

class `django_select2.util.`**`JSVar`**
Bases: `unicode`

A JS variable.

This is a simple Unicode string. This class type acts as a marker that this string is a JS variable name, so it must not be quoted by `convert_py_to_js_data()` while rendering the JS code.

class `django_select2.util.`**`JSFunction`**
Bases: `django_select2.util.JSVar`

A JS function name.

From rendering point of view, rendering this is no different from `JSVar`. After all, a JS variable can refer a function instance, primitive constant or any other object. They are still all variables.

Tip: Do use this marker for JS functions. This will make the code clearer, and the purpose more easier to understand.

---

class `django_select2.util.`**`JSFunctionInContext`**

Bases: `django_select2.util.JSVar`

A JS function name to run in context of some other HTML DOM element.

Like `JSFunction`, this too flags the string as JS function, but with a special requirement. The JS function needs to be invoked in the context of a HTML DOM, such that, `this` inside the function refers to that DOM instead of `window`.

Tip: JS functions of this type are wrapped inside special another JS function – `django_select2.runInContextHelper`.

`django_select2.util.`**`render_js_script`**`(`*inner_code*`)`

This wraps `inner_code` string inside the following code block:

```
<script type="text/javascript">
    jQuery(function ($) {
        // inner_code here
    });
</script>
```

Return type `unicode`

`django_select2.util.`**`extract_some_key_val`**`(`*dct*, *keys*`)`

Gets a sub-set of a `dict`.

Parameters

- **dct** (`dict`) – Source dictionary.
- **keys** (`list` or any iterable.) – List of subset keys, which to extract from `dct`.

Return type `dict`

`django_select2.util.`**`convert_py_to_js_data`**`(`*val*, *id_*`)`

Converts Python data type to JS data type.

Practically what this means is, convert `False` to `false`, `True` to `true` and so on. It also takes care of the conversion of `JSVar`, `JSFunction` and `JSFunctionInContext`. It takes care of recursively converting lists and dictionaries too.

Parameters

- **val** (*Any*) – The Python data to convert.
- **id** (`str`) – The DOM id of the element in which context `JSFunctionInContext` functions should run. (This is not needed if `val` contains no `JSFunctionInContext`)

Return type `unicode`

`django_select2.util.`**`convert_dict_to_js_map`**`(`*dct*, *id_*`)`

Converts a Python dictionary to JS map.

Parameters

- **dct** (`dict`) – The Python dictionary to convert.

- **id** (`str`) – The DOM id of the element in which context `JSFunctionInContext` functions should run. (This is not needed if `dct` contains no `JSFunctionInContext`)

Return type `unicode`

`django_select2.util.`**`convert_to_js_arr`**(*lst*, *id_*)

Converts a Python list (or any iterable) to JS array.

Parameters

- **lst** (`list` or Any iterable) – The Python iterable to convert.
- **id** (`str`) – The DOM id of the element in which context `JSFunctionInContext` functions should run. (This is not needed if `lst` contains no `JSFunctionInContext`)

Return type `unicode`

`django_select2.util.`**`convert_to_js_string_arr`**(*lst*)

Converts a Python list (or any iterable) of strings to JS array.

`convert_to_js_arr()` can always be used instead of this. However, since it knows that it only contains strings, it cuts down on unnecessary computations.

Return type `unicode`

`django_select2.util.`**`synchronized`**(*f*)

Decorator to synchronize multiple calls to a functions.

`django_select2.util.`**`is_valid_id`**(*val*)

Checks if `val` is a valid generated Id.

Parameters **val** (`str`) – The value to check.

Return type `bool`

`django_select2.util.`**`register_field`**(**args*, ***kwargs*)

Registers an Auto field for use with `views.AutoResponseView`.

Parameters

- **key** (`unicode`) – The key to use while registering this field.
- **field** (`AutoViewFieldMixin`) – The field to register.

Returns The generated Id for this field. If given `key` was already registered then the Id generated that time, would be returned.

Return type `unicode`

`django_select2.util.`**`get_field`**(*id_*)

Returns an Auto field instance registered with the given Id.

Parameters **id** (`unicode`) – The generated Id the field is registered with.

Return type `AutoViewFieldMixin` or None

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index