Django Extra Views Documentation

Release 0.14.0

Andrew Ingram

Jan 07, 2022

Contents

1 Features **3**

2 Table of Contents **5**
- 2.1 Getting Started . . . 5
- 2.2 Formset Views . . . 7
- 2.3 Formset Customization Examples . . . 11
- 2.4 List Views . . . 14

3 Reference **17**
- 3.1 Change History . . . 17

Django Extra Views provides a number of additional class-based generic views to complement those provide by Django itself. These mimic some of the functionality available through the standard admin interface, including Model, Inline and Generic Formsets.

CHAPTER 1

Features

- `FormSet` and `ModelFormSet` views - The formset equivalents of `FormView` and `ModelFormView`.
- `InlineFormSetView` - Lets you edit a formset related to a model (using Django's `inlineformset_factory`).
- `CreateWithInlinesView` and `UpdateWithInlinesView` - Lets you edit a model and multiple inline formsets all in one view.
- `GenericInlineFormSetView`, the equivalent of `InlineFormSetView` but for `GenericForeignKeys`.
- Support for generic inlines in `CreateWithInlinesView` and `UpdateWithInlinesView`.
- Support for naming each inline or formset in the template context with `NamedFormsetsMixin`.
- `SortableListMixin` - Generic mixin for sorting functionality in your views.
- `SearchableListMixin` - Generic mixin for search functionality in your views.
- `SuccessMessageMixin` and `FormSetSuccessMessageMixin` - Generic mixins to display success messages after form submission.

CHAPTER 2

Table of Contents

2.1 Getting Started

2.1.1 Installation

Install the stable release from pypi (using pip):

```
pip install django-extra-views
```

Or install the current master branch from github:

```
pip install -e git://github.com/AndrewIngram/django-extra-views.git#egg=django-extra-
↪views
```

Then add `'extra_views'` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'extra_views',
    ...
]
```

2.1.2 Quick Examples

FormSetView

Define a `FormSetView`, a view which creates a single formset from `django.forms.formset_factory` and adds it to the context.

```
from extra_views import FormSetView
from my_forms import AddressForm
```

(continues on next page)

(continued from previous page)

```
class AddressFormSet(FormSetView):
    form_class = AddressForm
    template_name = 'address_formset.html'
```

Then within `address_formset.html`, render the formset like this:

```
<form method="post">
  ...
  {{ formset }}
  ...
  <input type="submit" value="Submit" />
</form>
```

ModelFormSetView

Define a `ModelFormSetView`, a view which works as `FormSetView` but instead renders a model formset using `django.forms.modelformset_factory`.

```
from extra_views import ModelFormSetView


class ItemFormSetView(ModelFormSetView):
    model = Item
    fields = ['name', 'sku']
    template_name = 'item_formset.html'
```

CreateWithInlinesView or UpdateWithInlinesView

Define `CreateWithInlinesView` and `UpdateWithInlinesView`, views which render a form to create/update a model instance and its related inline formsets. Each of the `InlineFormSetFactory` classes use similar class definitions as the `ModelFormSetView`.

```
from extra_views import CreateWithInlinesView, UpdateWithInlinesView,␣
↪InlineFormSetFactory


class ItemInline(InlineFormSetFactory):
    model = Item
    fields = ['sku', 'price', 'name']


class ContactInline(InlineFormSetFactory):
    model = Contact
    fields = ['name', 'email']


class CreateOrderView(CreateWithInlinesView):
    model = Order
    inlines = [ItemInline, ContactInline]
    fields = ['customer', 'name']
    template_name = 'order_and_items.html'
```

(continues on next page)

(continued from previous page)

```
class UpdateOrderView(UpdateWithInlinesView):
    model = Order
    inlines = [ItemInline, ContactInline]
    fields = ['customer', 'name']
    template_name = 'order_and_items.html'
```

Then within `order_and_items.html`, render the formset like this:

```
<form method="post">
  ...
  {{ form }}

  {% for formset in inlines %}
    {{ formset }}
  {% endfor %}
  ...
  <input type="submit" value="Submit" />
</form>
```

2.2 Formset Views

For all of these views we've tried to mimic the API of Django's existing class-based views as closely as possible, so they should feel natural to anyone who's already familiar with Django's views.

2.2.1 FormSetView

This is the formset equivalent of Django's FormView. Use it when you want to display a single (non-model) formset on a page.

A simple formset:

```
from extra_views import FormSetView
from my_app.forms import AddressForm


class AddressFormSetView(FormSetView):
    template_name = 'address_formset.html'
    form_class = AddressForm
    success_url = 'success/'

    def get_initial(self):
        # return whatever you'd normally use as the initial data for your formset.
      return data

    def formset_valid(self, formset):
        # do whatever you'd like to do with the valid formset
        return super(AddressFormSetView, self).formset_valid(formset)
```

and in `address_formset.html`:

```
<form method="post">
  ...
```

(continues on next page)

(continued from previous page)

```
  {{ formset }}
  ...
  <input type="submit" value="Submit" />
</form>
```

This view will render the template `address_formset.html` with a context variable `formset` representing the `AddressFormSet`. Once POSTed and successfully validated, `formset_valid` will be called (which is where your handling logic goes), then the view will redirect to `success_url`.

Formset constructor and factory kwargs

FormSetView exposes all the parameters you'd normally be able to pass to the `django.forms.BaseFormSet` constructor and `django.forms.formset_factory()`. This can be done by setting the respective attribute on the class, or `formset_kwargs` and `factory_kwargs` at the class level.

Below is an exhaustive list of all formset-related attributes which can be set at the class level for `FormSetView`:

```
...
from my_app.forms import AddressForm, BaseAddressFormSet


class AddressFormSetView(FormSetView):
    template_name = 'address_formset.html'
    form_class = AddressForm
    formset_class = BaseAddressFormSet
    initial = [{'type': 'home'}, {'type': 'work'}]
    prefix = 'address-form'
    success_url = 'success/'
    factory_kwargs = {'extra': 2, 'max_num': None,
                      'can_order': False, 'can_delete': False}
    formset_kwargs = {'auto_id': 'my_id_%s'}
```

In the above example, BaseAddressFormSet would be a subclass of `django.forms.BaseFormSet`.

2.2.2 ModelFormSetView

ModelFormSetView makes use of `django.forms.modelformset_factory()`, using the declarative syntax used in `FormSetView` as well as Django's own class-based views. So as you'd expect, the simplest usage is as follows:

```
from extra_views import ModelFormSetView
from my_app.models import Item


class ItemFormSetView(ModelFormSetView):
    model = Item
    fields = ['name', 'sku', 'price']
    template_name = 'item_formset.html'
```

Rather than setting `fields`, `exclude` can be defined at the class level as a list of fields to be excluded.

It is not necessary to define `fields` or `exclude` if a `form_class` is defined at the class level:

```
...
from django.forms import ModelForm


class ItemForm(ModelForm):
    # Custom form definition goes here
    fields = ['name', 'sku', 'price']


class ItemFormSetView(ModelFormSetView):
    model = Item
    form_class = ItemForm
    template_name = 'item_formset.html'
```

Like `FormSetView`, the `formset` variable is made available in the template context. By default this will populate the formset with all the instances of `Item` in the database. You can control this by overriding `get_queryset` on the class, which could filter on a URL kwarg (`self.kwargs`), for example:

```
class ItemFormSetView(ModelFormSetView):
    model = Item
    template_name = 'item_formset.html'

    def get_queryset(self):
        sku = self.kwargs['sku']
        return super(ItemFormSetView, self).get_queryset().filter(sku=sku)
```

2.2.3 InlineFormSetView

When you want to edit instances of a particular model related to a parent model (using a ForeignKey), you'll want to use InlineFormSetView. An example use case would be editing addresses associated with a particular contact.

```
from extra_views import InlineFormSetView


class EditContactAddresses(InlineFormSetView):
    model = Contact
    inline_model = Address

    ...
```

Aside from the use of `model` and `inline_model`, `InlineFormSetView` works more-or-less in the same way as `ModelFormSetView`, instead calling `django.forms.inlineformset_factory()`.

2.2.4 CreateWithInlinesView and UpdateWithInlinesView

These are the most powerful views in the library, they are effectively replacements for Django's own `CreateView` and `UpdateView`. The key difference is that they let you include any number of inline formsets (as well as the parent model's form). This provides functionality much like the Django Admin change forms. The API should be fairly familiar as well. The list of the inlines will be passed to the template as context variable *inlines*.

Here is a simple example that demonstrates the use of each view with normal inline relationships:

```
from extra_views import CreateWithInlinesView, UpdateWithInlinesView,␣
↪InlineFormSetFactory


class ItemInline(InlineFormSetFactory):
    model = Item
    fields = ['sku', 'price', 'name']


class ContactInline(InlineFormSetFactory):
    model = Contact
    fields = ['name', 'email']


class CreateOrderView(CreateWithInlinesView):
    model = Order
    inlines = [ItemInline, ContactInline]
    fields = ['customer', 'name']
    template_name = 'order_and_items.html'

    def get_success_url(self):
        return self.object.get_absolute_url()


class UpdateOrderView(UpdateWithInlinesView):
    model = Order
    inlines = [ItemInline, ContactInline]
    fields = ['customer', 'name']
    template_name = 'order_and_items.html'

    def get_success_url(self):
        return self.object.get_absolute_url()
```

and in the html template:

```
<form method="post">
  ...
  {{ form }}

  {% for formset in inlines %}
    {{ formset }}
  {% endfor %}
  ...
  <input type="submit" value="Submit" />
</form>
```

InlineFormSetFactory

This class represents all the configuration necessary to generate an inline formset from `django.inlineformset_factory()`. Each class within in `CreateWithInlines.inlines` and `UpdateWithInlines.inlines` should be a subclass of `InlineFormSetFactory`. All the same methods and attributes as `InlineFormSetView` are available, with the exception of any view-related attributes and methods, such as `success_url` or `formset_valid()`:

```
from my_app.forms import ItemForm, BaseItemFormSet
from extra_views import InlineFormSetFactory
```

(continues on next page)

(continued from previous page)

```
class ItemInline(InlineFormSetFactory):
    model = Item
    form_class = ItemForm
    formset_class = BaseItemFormSet
    initial = [{'name': 'example1'}, {'name', 'example2'}]
    prefix = 'item-form'
    factory_kwargs = {'extra': 2, 'max_num': None,
                      'can_order': False, 'can_delete': False}
    formset_kwargs = {'auto_id': 'my_id_%s'}
```

IMPORTANT: Note that when using `InlineFormSetFactory`, `model` should be the *inline* model and **not** the parent model.

2.2.5 GenericInlineFormSetView

In the specific case when you would usually use Django's `django.contrib.contenttypes.forms.generic_inlineformset_factory()`, you should use `GenericInlineFormSetView`. The kwargs `ct_field` and `fk_field` should be set in `factory_kwargs` if they need to be changed from their default values:

```
from extra_views.generic import GenericInlineFormSetView


class EditOrderTags(GenericInlineFormSetView):
    model = Order
    inline_model = Tag
    factory_kwargs = {'ct_field': 'content_type', 'fk_field': 'object_id',
                      'max_num': 1}
    formset_kwargs = {'save_as_new': True}

    ...
```

There is a `GenericInlineFormSetFactory` which is analogous to `InlineFormSetFactory` for use with generic inline formsets.

`GenericInlineFormSetFactory` can be used in `CreateWithInlines.inlines` and `UpdateWithInlines.inlines` in the obvious way.

2.3 Formset Customization Examples

2.3.1 Overriding formset_kwargs and factory_kwargs at run time

If the values in `formset_kwargs` and `factory_kwargs` need to be modified at run time, they can be set by overloading the `get_formset_kwargs()` and `get_factory_kwargs()` methods on any formset view (model, inline or generic) and the `InlineFormSetFactory` classes:

```
class AddressFormSetView(FormSetView):
    ...

    def get_formset_kwargs(self):
```

(continues on next page)

(continued from previous page)

```
        kwargs = super(AddressFormSetView, self).get_formset_kwargs()
        # modify kwargs here
        return kwargs

    def get_factory_kwargs(self):
        kwargs = super(AddressFormSetView, self).get_factory_kwargs()
        # modify kwargs here
        return kwargs
```

2.3.2 Overriding the the base formset class

The `formset_class` option should be used if you intend to override the formset methods of a view or a subclass of `InlineFormSetFactory`.

For example, imagine you'd like to add your custom `clean` method for an inline formset view. Then, define a custom formset class, a subclass of Django's `BaseInlineFormSet`, like this:

```
from django.forms.models import BaseInlineFormSet

class ItemInlineFormSet(BaseInlineFormSet):

    def clean(self):
        # ...
        # Your custom clean logic goes here
```

Now, in your `InlineFormSetView` sub-class, use your formset class via `formset_class` setting, like this:

```
from extra_views import InlineFormSetView
from my_app.models import Item
from my_app.forms import ItemForm

class ItemInlineView(InlineFormSetView):
    model = Item
    form_class = ItemForm
    formset_class = ItemInlineFormSet      # enables our custom inline
```

This will enable `clean` method being executed on the formset used by `ItemInlineView`.

2.3.3 Initial data for ModelFormSet and InlineFormSet

Passing initial data into ModelFormSet and InlineFormSet works slightly differently to a regular FormSet. The data passed in from `initial` will be inserted into the `extra` forms of the formset. Only the data from `get_queryset()` will be inserted into the initial rows:

```
from extra_views import ModelFormSetView
from my_app.models import Item


class ItemFormSetView(ModelFormSetView):
    template_name = 'item_formset.html'
    model = Item
    factory_kwargs = {'extra': 10}
    initial = [{'name': 'example1'}, {'name': 'example2'}]
```

The above will result in a formset containing a form for each instance of `Item` in the database, followed by 2 forms containing the extra initial data, followed by 8 empty forms.

Altenatively, initial data can be determined at run time and passed in by overloading `get_initial()`:

```
...
class ItemFormSetView(ModelFormSetView):
    model = Item
    template_name = 'item_formset.html'
    ...

    def get_initial(self):
        # Get a list of initial values for the formset here
        initial = [...]
        return initial
```

2.3.4 Passing arguments to the form constructor

In order to change the arguments which are passed into each form within the formset, this can be done by the 'form_kwargs' argument passed in to the FormSet constructor. For example, to give every form an initial value of 'example' in the 'name' field:

```
from extra_views import InlineFormSetFactory

class ItemInline(InlineFormSetFactory):
    model = Item
    formset_kwargs = {'form_kwargs': {'initial': {'name': 'example'}}}
```

If these need to be modified at run time, it can be done by `get_formset_kwargs()`:

```
from extra_views import InlineFormSetFactory

class ItemInline(InlineFormSetFactory):
    model = Item

    def get_formset_kwargs(self):
        kwargs = super(ItemInline, self).get_formset_kwargs()
        initial = get_some_initial_values()
        kwargs['form_kwargs'].update({'initial': initial})
        return kwargs
```

2.3.5 Named formsets

If you want more control over the names of your formsets (as opposed to iterating over `inlines`), you can use `NamedFormsetsMixin`:

```
from extra_views import NamedFormsetsMixin

class CreateOrderView(NamedFormsetsMixin, CreateWithInlinesView):
    model = Order
    inlines = [ItemInline, TagInline]
    inlines_names = ['Items', 'Tags']
    fields = '__all__'
```

Then use the appropriate names to render them in the html template:

```
...
{{ Tags }}
...
{{ Items }}
...
```

2.3.6 Success messages

When using Django's messages framework, mixins are available to send success messages in a similar way to `django.contrib.messages.views.SuccessMessageMixin`. Ensure that `'django.contrib.messages.middleware.MessageMiddleware'` is included in the `MIDDLEWARE` section of *settings.py*.

`extra_views.SuccessMessageMixin` is for use with views with multiple inline formsets. It is used in an identical manner to Django's SuccessMessageMixin, making `form.cleaned_data` available for string interpolation using the `%(field_name)s` syntax:

```
from extra_views import CreateWithInlinesView, SuccessMessageMixin
...

class CreateOrderView(SuccessMessageMixin, CreateWithInlinesView):
    model = Order
    inlines = [ItemInline, ContactInline]
    success_message = 'Order %(name)s successfully created!'
    ...

    # or instead, set at runtime:
    def get_success_message(self, cleaned_data, inlines):
        return 'Order with id {} successfully created'.format(self.object.pk)
```

Note that the success message mixins should be placed ahead of the main view in order of class inheritance.

`extra_views.FormSetSuccessMessageMixin` is for use with views which handle a single formset. In order to parse any data from the formset, you should override the `get_success_message` method as below:

```
from extra_views import FormSetView, FormSetSuccessMessageMixin
from my_app.forms import AddressForm


class AddressFormSetView(FormSetView):
    form_class = AddressForm
    success_url = 'success/'
    ...
    success_message = 'Addresses Updated!'

# or instead, set at runtime
def get_success_message(self, formset)
    # Here you can use the formset in the message if required
    return '{} addresses were updated.'.format(len(formset.forms))
```

2.4 List Views

2.4.1 Searchable List Views

You can add search functionality to your ListViews by adding SearchableListMixin and by setting search_fields:

```
from django.views.generic import ListView
from extra_views import SearchableListMixin

class SearchableItemListView(SearchableListMixin, ListView):
    template_name = 'extra_views/item_list.html'
    search_fields = ['name', 'sku']
    model = Item
```

In this case `object_list` will be filtered if the 'q' query string is provided (like /searchable/?q=query), or you can manually override `get_search_query` method, to define your own search functionality.

Also you can define some items in `search_fields` as tuple (e.g. `[('name', 'iexact', ), 'sku']`) to provide custom lookups for searching. Default lookup is `icontains`. We strongly recommend to use only string lookups, when number fields will convert to strings before comparison to prevent converting errors. This controlled by `check_lookups` setting of SearchableMixin.

2.4.2 Sortable List View

```
from django.views.generic import ListView
from extra_views import SortableListMixin

class SortableItemListView(SortableListMixin, ListView):
    sort_fields_aliases = [('name', 'by_name'), ('id', 'by_id'), ]
    model = Item
```

You can hide real field names in query string by define sort_fields_aliases attribute (see example) or show they as is by define sort_fields. SortableListMixin adds `sort_helper` variable of SortHelper class, then in template you can use helper functions: `{{ sort_helper.get_sort_query_by_FOO }}`, `{{ sort_helper.get_sort_query_by_FOO_asc }}`, `{{ sort_helper.get_sort_query_by_FOO_desc }}` and `{{ sort_helper.is_sorted_by_FOO }}`

CHAPTER 3

Reference

3.1 Change History

3.1.1 0.14.0 (2021-06-08)

Changes:

Supported Versions:

Python	Django
3.5	2.1–2.2
3.6-3.7	2.1–3.1
3.8	2.2–3.1

- Removed support for Python 2.7.
- Added support for Python 3.8 and Django 3.1.
- Removed the following classes (use the class in parentheses instead):
  - `BaseFormSetMixin` (use `BaseFormSetFactory`).
  - `BaseInlineFormSetMixin` (use `BaseInlineFormSetFactory`).
  - `InlineFormSet` (use `InlineFormSetFactory`).
  - `BaseGenericInlineFormSetMixin` (use `BaseGenericInlineFormSetFactory`).
  - `GenericInlineFormSet` (use `GenericInlineFormSetFactory`).

3.1.2 0.13.0 (2019-12-20)

Changes:

Supported Versions:

Python	Django
2.7	1.11
3.5	1.11–2.2
3.6-3.7	1.11–3.0

- Added `SuccessMessageMixin` and `FormSetSuccessMessageMixin`.
- `CreateWithInlinesView` and `UpdateWithInlinesView` now call `self.form_valid` method within `self.forms_valid`.
- Revert `view.object` back to it's original value from the GET request if validation fails for the inline formsets in `CreateWithInlinesView` and `UpdateWithInlinesview`.
- Added support for Django 3.0.

3.1.3 0.12.0 (2018-10-21)

Supported Versions:

Python	Django
2.7	1.11
3.4	1.11–2.0
3.5-3.7	1.11–2.1

Changes:

- Removed setting of `BaseInlineFormSetMixin.formset_class` and `GenericInlineFormSetMixin.formset_class` so that `formset` can be set in `factory_kwargs` instead.
- Removed `ModelFormSetMixin.get_context_data` and `BaseInlineFormSetMixin.get_context_data` as this code was duplicated from Django's `MultipleObjectMixin` and `SingleObjectMixin` respectively.
- Renamed `BaseFormSetMixin` to `BaseFormSetFactory`.
- Renamed `BaseInlineFormSetMixin` to `BaseInlineFormSetFactory`.
- Renamed `InlineFormSet` to `InlineFormSetFactory`.
- Renamed `BaseGenericInlineFormSetMixin` to `BaseGenericInlineFormSetFactory`.
- Renamed `GenericInlineFormSet` to `GenericInlineFormSetFactory`.

All renamed classes will be removed in a future release.

3.1.4 0.11.0 (2018-04-24)

Supported Versions:

Python	Django
2.7	1.11
3.4–3.6	1.11–2.0

Backwards-incompatible changes

- Dropped support for Django 1.7–1.10.
- Removed support for factory kwargs `extra`, `max_num`, `can_order`, `can_delete`, `ct_field`, `formfield_callback`, `fk_name`, `widgets`, `ct_fk_field` being set on `BaseFormSetMixin` and its subclasses. Use `BaseFormSetMixin.factory_kwargs` instead.
- Removed support for formset_kwarg `save_as_new` being set on `BaseInlineFormSetMixin` and its subclasses. Use `BaseInlineFormSetMixin.formset_kwargs` instead.
- Removed support for `get_extra_form_kwargs`. This can be set in the dictionary key `form_kwargs` in `BaseFormSetMixin.formset_kwargs` instead.

3.1.5 0.10.0 (2018-02-28)

New features:

- Added SuccessMessageWithInlinesMixin (#151)
- Allow the formset prefix to be overridden (#154)

Bug fixes:

- SearchableMixin: Fix reduce() of empty sequence error (#149)
- Add fields attributes (Issue #144, PR #150)
- Fix Django 1.11 AttributeError: This QueryDict instance is immutable (#156)

3.1.6 0.9.0 (2017-03-08)

This version supports Django 1.7, 1.8, 1.9, 1.10 (latest minor versions), and Python 2.7, 3.4, 3.5 (latest minor versions).

- Added Django 1.10 support
- Dropped Django 1.6 support

3.1.7 0.8 (2016-06-14)

This version supports Django 1.6, 1.7, 1.8, 1.9 (latest minor versions), and Python 2.7, 3.4, 3.5 (latest minor versions).

- Added `widgets` attribute setting; allow to change form widgets in the `ModelFormSetView`.
- Added Django 1.9 support.
- Fixed `get_context_data()` usage of `*args, **kwargs`.
- Fixed silent overwriting of `ModelForm` fields to `__all__`.

Backwards-incompatible changes

- Dropped support for Django <= 1.5 and Python 3.3.
- Removed the `extra_views.multi` module as it had neither documentation nor test coverage and was broken for some of the supported Django/Python versions.
- This package no longer implicitly set `fields = '__all__'`. If you face `ImproperlyConfigured` exceptions, you should have a look at the Django 1.6 release notes and set the `fields` or `exclude` attributes on your `ModelForm` or extra-views views.

3.1.8 0.7.1 (2015-06-15)

Beginning of this changelog.