django-betterforms Documentation

Release 1.1.1

Fusionbox, Inc.

August 18, 2015

Contents

Making django forms suck less.

Contents:

CHAPTER 1

Introduction

`django-betterforms` provides a suite of tools to make working with forms in Django easier.

1.1 Installation

1. Install the package:

```
$ pip install django-betterforms
```

   Or you can install it from source:

```
$ pip install -e git://github.com/fusionbox/django-betterforms@master#egg=django-betterforms-dev
```

2. Add `betterforms` to your `INSTALLED_APPS`.

1.2 Quick Start

Getting started with `betterforms` is easy. If you are using the build in form base classes provided by django, its as simple as switching to the form base classes provided by `betterforms`.

```
from betterforms.forms import BaseForm

class RegistrationForm(BaseForm):
    ...
    class Meta:
        fieldsets = (
            ('info', {'fields': ('username', 'email')}),
            ('location', {'fields': ('address', ('city', 'state', 'zip'))}),
            ('password', {'password1', 'password2'}),
        )
```

And then in your template.

```
<form method="post">
     {% include 'betterforms/form_as_fieldsets.html' %}
</form>
```

Which will render the following.

```
<fieldset class="formFieldset info">
        <div class="required username formField">
                <label for="id_username">Username</label>
                <input id="id_username" name="username" type="text" />
        </div>
        <div class="required email formField">
                <label for="id_email">Email</label>
                <input id="id_email" name="email" type="text" />
        </div>
</fieldset>
<fieldset class="formFieldset location">
        <div class="required address formField">
                <label for="id_address">Address</label>
                <input id="id_address" name="address" type="text" />
        </div>
        <fieldset class="formFieldset location_1">
                <div class="required city formField">
                        <label for="id_city">City</label>
                        <input id="id_city" name="city" type="text" />
                </div>
                <div class="required state formField">
                        <label for="id_state">State</label>
                        <input id="id_state" name="state" type="text" />
                </div>
                <div class="required zip formField">
                        <label for="id_zip">Zip</label>
                        <input id="id_zip" name="zip" type="text" />
                </div>
        </fieldset>
</fieldset>
<fieldset class="formFieldset password">
        <div class="required password1 formField">
                <label for="id_password1">Password</label>
                <input id="id_password1" name="password1" type="password" />
        </div>
        <div class="required password2 formField">
                <label for="id_password2">Confirm Password</label>
                <input id="id_password2" name="password2" type="password" />
        </div>
</fieldset>
```

CHAPTER 2

Forms

django-betterforms provides two new form base classes to use in place of django.forms.Form. and django.forms.ModelForm.

class betterforms.forms.**BetterForm**
Base form class for non-model forms.

class betterforms.forms.**BetterModelForm**
Base form class for model forms.

2.1 Errors

Adding errors in betterforms is easy:

```
>>> form = BlogEntryForm(request.POST)
>>> form.is_valid()
True
>>> form.field_error('title', 'This title is already taken')
>>> form.is_valid()
False
>>> form.errors
{'title': ['This title is already taken']}
```

You can also add global errors:

```
>>> form = BlogEntryForm(request.POST)
>>> form.form_error('Not accepting new entries at this time')
>>> form.is_valid()
False
>>> form.errors
{'__all__': ['Not accepting new entries at this time']}
```

form_error is simply a wrapper around *field_error* that uses the key *__all__* for the field name.

2.2 Fieldsets

One of the most powerful features in betterforms is the ability to declare field groupings. Both *BetterForm* and *BetterModelForm* provide a common interface for working with fieldsets.

Fieldsets can be declared in any of three formats, or any mix of the three formats.

- As Two Tuples

  Similar to admin fieldsets, as a list of two tuples. The two tuples should be in the format `(name, fieldset_options)` where `name` is a string representing the title of the fieldset and `fieldset_options` is a dictionary which will be passed as `kwargs` to the constructor of the fieldset.

```
from betterforms.forms import BetterForm

class RegistrationForm(BetterForm):
    ...
    class Meta:
        fieldsets = (
            ('info', {'fields': ('username', 'email')}),
            ('location', {'fields': ('address', ('city', 'state', 'zip'))}),
            ('password', {'password1', 'password2'}),
        )
```

- As a list of field names

  Fieldsets can be declared as a list of field names.

```
from betterforms.forms import BetterForm

class RegistrationForm(BetterForm):
    ...
    class Meta:
        fieldsets = (
            ('username', 'email'),
            ('address', ('city', 'state', 'zip')),
            ('password1', 'password2'),
        )
```

- As instantiated `Fieldset` instances or subclasses of `Fieldset`.

  Fieldsets can be declared as a list of field names.

```
from betterforms.forms import BetterForm, Fieldset

class RegistrationForm(BetterForm):
    ...
    class Meta:
        fieldsets = (
            Fieldset('info', fields=('username', 'email')),
            Fieldset('location', ('address', ('city', 'state', 'zip'))),
            Fieldset('password', ('password1', 'password2')),
        )
```

All three of these examples will have *appoximately* the same output. All of these formats can be mixed and matched and nested within each other. And Unlike django-admin, you may nest fieldsets as deep as you would like.

A `Fieldset` can also optionally be declared with a legend kwarg, which will then be made available as a property to the associated `BoundFieldset`.

```
Fieldset('location', ('address', ('city', 'state', 'zip')), legend='Place of Residence')
```

Should you choose to render the form using the betterform templates detailed below, each fieldset with a legend will be rendered with an added legend tag in the template.

2.3 Rendering

To render a form, use the provided template partial.

```
<form method="post">
     {% include 'betterforms/form_as_fieldsets.html' %}
</form>
```

This partial assumes there is a variable `form` available in its context. This template does the following things.

- outputs the `csrf_token`.
- outputs a hidden field named `next` if there is a `next` value available in the template context.
- outputs the form media
- **loops over `form.fieldsets`.**
  - for each fieldsets, renders the fieldset using the template `betterforms/fieldset_as_div.html`
    - for each item in the fieldset, if it is a fieldset, it is rendered using the same template, and if it is a field, renders it using the field template.
  - for each field, renders the field using the template `betterforms/field_as_div.html`

If you want to output the form without the CSRF token (for example on a GET form), you can do so by passing in the csrf_exempt variable.

```
<form method="post">
     {% include 'betterforms/form_as_fieldsets.html' csrf_exempt=True %}
</form>
```

If you wish to override the label suffix, `django-betterforms` provides a convenient class attribute on the *`BetterForm`* and *`BetterModelForm`* classes.

```
class MyForm(forms.BetterForm):
    # ... fields

    label_suffix = '->'
```

Warning: Due to a bug in dealing with the label suffix in Django < 1.6, the `label_suffix` will not appear in any forms rendered using the betterforms templates. For more information, refer to the Django bug #18134.

CHAPTER 3

Changelist Forms

Changelist Forms are designed to facilitate easy searching and sorting on django models, along with providing a framework for building other functionality that deals with operations on lists of model instances.

class betterforms.changelist.**ChangeListForm**
Base form class for all **Changelist** forms.

- setting the queryset:

All changelist forms need to *know* what queryset to begin with. You can do this by either passing a named keyword parameter into the contructor of the form, or by defining a model attribute on the class.

class betterforms.changelist.**SearchForm**

Form class which facilitates searching across a set of specified fields for a model. This form adds a field to the model q for the search query.

SEARCH_FIELDS
The list of fields that will be searched against.

CASE_SENSITIVE

Whether the search should be case sensitive.

Here is a simple *SearchForm* example for searching across users.

```
# my_app/forms.py
from django.contrib.auth.models import get_user_model
from betterforms.forms import SearchForm

class UserSearchForm(SearchForm):
    SEARCH_FIELDS = ('username', 'email', 'name')
    model = get_user_model()

 # my_app.views.py
 from my_app.forms import UserSearchForm

 def user_list_view(request):
     form = UserSearchForm(request.GET)
     context = {'form': form}
     if form.is_valid:
         context['queryset'] = form.get_queryset()
     return render_to_response(context, ...)
```

SearchForm checks to see if the query value is present in any of the fields declared in SEARCH_FIELDS by or-ing together Q objects using __contains or __icontains queries on those fields.

class `betterforms.changelist.`**SortForm**

Form which facilitates the sorting instances of a model. This form adds a hidden field `sorts` to the model which is used to dictate the columns which should be sorted on and in what order.

HEADERS

The list of *Header* objects for sorting.

Headers can be declared in multiple ways.

•As instantiated *Header* objects.:

```
# my_app/forms.py
from betterforms.forms import SortForm, Header

class UserSortForm(SortForm):
    HEADERS = (
        Header('username', ..),
        Header('email', ..),
        Header('name', ..),
    )
    model = User
```

•As a string:

```
# my_app/forms.py
from betterforms.forms import SortForm

class UserSortForm(SortForm):
    HEADERS = (
        'username',
        'email',
        'name',
    )
    model = User
```

•As an iterable of `*args` which will be used to instantiate the *Header* object.:

```
# my_app/forms.py
from betterforms.forms import SortForm

class UserSortForm(SortForm):
    HEADERS = (
        ('username', ..),
        ('email', ..),
        ('name', ..),
    )
    model = User
```

•As a two-tuple of **header name** and `**kwargs`. The name and provided `**kwargs` will be used to instantiate the *Header* objects.

```
# my_app/forms.py
from betterforms.forms import SortForm

class UserSortForm(SortForm):
    HEADERS = (
        ('username', {..}),
        ('email', {..}),
```

```
            ('name', {..}),
        )
        model = User
```

All of these examples are roughly equivilent, resulting in the form haveing three sortable headers, `('username', 'email', 'name')`, which will map to those named fields on the `User` model.

See documentation on the *`Header`* class for more information on how sort headers can be configured.

`get_order_by``()`
Returns a list of column names that are used in the `order_by` call on the returned queryset.

During instantiation, all declared headers on `form.HEADERS` are converted to *`Header`* objects and are accessible from `form.headers`.

```
>>> [header for header in form.headers]  # Iterate over all headers.
>>> form.headers[2]  #  Get the header at index-2
>>> form.headers['username']  #  Get the header named 'username'
```

class `betterforms.changelist.`**`Header`**(*name*, *label=None*, *column_name=None*, *is_sortable=True*)

Headers are the the mechanism through which *`SortForm`* shines. They provide querystrings for operations related to sorting by whatever query parameter that header is tied to, as well as other values that are helpful during rendering.

`name`

The name of the header.

`label`

The human readable name of the header.

`is_active`

`Boolean` as to whether this header is currently being used to sort.

`is_ascending`

`Boolean` as to whether this header is being used to sort the data in ascending order.

`is_descending`

`Boolean` as to whether this header is being used to sort the data in descending order.

`css_classes`

Space separated list of css classes, suitable for output in the template as an HTML element's css class attribute.

`priority`

1-indexed number as to the priority this header is at in the list of sorts. Returns `None` if the header is not active.

`querystring`

The querystring that will sort by this header as the primary sort, moving all other active sorts in their current order after this one. Preserves all other query parameters.

`remove_querystring`

The querystring that will remove this header from the sorts. Preserves all other query parameters.

`singular_querystring`

The querystring that will sort by this header, and deactivate all other active sorts. Preserves all other query parameters.

3.1 Working With Changelists

Outputting sort form headers can be done using a provided template partial located at `betterforms/sort_form_header.html`

```
<th class="{{ header.css_classes }}">
  {% if header.is_sortable %}
    <a href="?{{ header.querystring }}">{{ header.label }}</a>
    {% if header.is_active %}
      {% if header.is_ascending %}

      {% elif header.is_descending %}

      {% endif %}
      <a href="" data-sort_by="title" data-direction="up"></a>
      <span class="filterActive"><span>{{ header.priority }}</span> <a href="?{{ header.remove_querys
    {% endif %}
  {% else %}
    {{ header.label }}
  {% endif %}
</th>
```

This example assumes that you are using a table to output your data. It should be trivial to modify this to suite your needs.

class `betterforms.views.`**`BrowseView`**

Class-based view for working with changelists. It is a combination of `FormView` and `ListView` in that it handles form submissions and providing a optionally paginated queryset for rendering in the template.

Works similarly to the standard `FormView` class provided by django, except that the form is instantiated using `request.GET`, and the `object_list` passed into the template context comes from `form.get_queryset()`.

CHAPTER 4

MultiForm and MultiModelForm

A container that allows you to treat multiple forms as one form. This is great for using more than one form on a page that share the same submit button. *MultiForm* imitates the Form API so that it is invisible to anybody else (for example, generic views) that you are using a *MultiForm*.

There are a couple of differences, though. One lies in how you initialize the form. See this example:

```
class UserProfileMultiForm(MultiForm):
    form_classes = {
        'user': UserForm,
        'profile': ProfileForm,
    }

UserProfileMultiForm(initial={
    'user': {
        # User's initial data
    },
    'profile': {
        # Profile's initial data
    },
})
```

The initial data argument has to be a nested dictionary so that we can associate the right initial data with the right form class.

The other major difference is that there is no direct field access because this could lead to namespace clashes. You have to access the fields from their forms. All forms are available using the key provided in *form_classes*:

```
form = UserProfileMultiForm()
# get the Field object
form['user'].fields['name']
# get the BoundField object
form['user']['name']
```

MultiForm, however, does all you to iterate over all the fields of all the forms.

```
{% for field in form %}
  {{ field }}
{% endfor %}
```

If you are relying on the fields to come out in a consistent order, you should use an OrderedDict to define the *form_classes*.

```
from collections import OrderedDict
```

```
class UserProfileMultiForm(MultiForm):
    form_classes = OrderedDict((
        ('user', UserForm),
        ('profile', ProfileForm),
    ))
```

4.1 Working with ModelForms

MultiModelForm adds ModelForm support on top of MultiForm. That simply means that it includes support for the instance parameter in initialization and adds a save method.

```
class UserProfileMultiForm(MultiModelForm):
    form_classes = {
        'user': UserForm,
        'profile': ProfileForm,
    }

user = User.objects.get(pk=123)
UserProfileMultiForm(instance={
    'user': user,
    'profile': user.profile,
})
```

4.2 Working with CreateView

It is pretty easy to use MultiModelForms with Django's `CreateView`, usually you will have to override the `form_valid()` method to do some specific saving functionality. For example, you could have a signup form that created a user and a user profile object all in one:

```
# forms.py
from django import forms
from authtools.forms import UserCreationForm
from betterforms.multiform import MultiModelForm
from .models import UserProfile

class UserProfileForm(forms.ModelForm):
    class Meta:
        fields = ('favorite_color',)

class UserCreationMultiForm(MultiModelForm):
    form_classes = {
        'user': UserCreationForm,
        'profile': UserProfileForm,
    }

# views.py
from django.views.generic import CreateView
from django.core.urlresolvers import reverse_lazy
from django.shortcuts import redirect
from .forms import UserCreationMultiForm

class UserSignupView(CreateView):
    form_class = UserCreationMultiForm
```

```
    success_url = reverse_lazy('home')

    def form_valid(self, form):
        # Save the user first, because the profile needs a user before it
        # can be saved.
        user = form['user'].save()
        profile = form['profile'].save(commit=False)
        profile.user = user
        profile.save()
        return redirect(self.get_success_url())
```

Note: In this example, we used the `UserCreationForm` from the django-authtools package just for the purposes of brevity. You could of course use any ModelForm that you wanted to.

Of course, we could put the save logic in the `UserCreationMultiForm` itself by overriding the *`MultiModelForm.save()`* method.

```
class UserCreationMultiForm(MultiModelForm):
    form_classes = {
        'user': UserCreationForm,
        'profile': UserProfileForm,
    }

    def save(self, commit=True):
        objects = super(UserCreationMultiForm, self).save(commit=False)

        if commit:
            user = objects['user']
            user.save()
            profile = objects['profile']
            profile.user = user
            profile.save()

        return objects
```

If we do that, we can simplify our view to this:

```
class UserSignupView(CreateView):
    form_class = UserCreationMultiForm
    success_url = reverse_lazy('home')
```

4.3 Working with UpdateView

Working with `UpdateView` likewise is quite easy, but you most likely will have to override the `django.views.generic.edit.FormMixin.get_form_kwargs` method in order to pass in the instances that you want to work on. If we keep with the user/profile example, it would look something like this:

```
# forms.py
from django import forms
from django.contrib.auth import get_user_model
from betterforms.multiform import MultiModelForm
from .models import UserProfile

User = get_user_model()
```

```
class UserEditForm(forms.ModelForm):
    class Meta:
        fields = ('email',)

class UserProfileForm(forms.ModelForm):
    class Meta:
        fields = ('favorite_color',)

class UserEditMultiForm(MultiModelForm):
    form_classes = {
        'user': UserEditForm,
        'profile': UserProfileForm,
    }

# views.py
from django.views.generic import UpdateView
from django.core.urlresolvers import reverse_lazy
from django.shortcuts import redirect
from django.contrib.auth import get_user_model
from .forms import UserEditMultiForm

User = get_user_model()

class UserSignupView(UpdateView):
    model = User
    form_class = UserEditMultiForm
    success_url = reverse_lazy('home')

    def get_form_kwargs(self):
        kwargs = super(UserSignupView, self).get_form_kwargs()
        kwargs.update(instance={
            'user': self.object,
            'profile': self.object.profile,
        })
        return kwargs
```

4.4 API Reference

class betterforms.multiform.**MultiForm**

The main interface for customizing *MultiForms* is through overriding the *form_classes* class attribute.

Once a MultiForm is instantiated, you can access the child form instances with their names like this:

```
>>> class MyMultiForm(MultiForm):
        form_classes = {
            'foo': FooForm,
            'bar': BarForm,
        }
>>> forms = MyMultiForm()
>>> foo_form = forms['foo']
```

You may also iterate over a multiform to get all of the fields for each child instance.

MultiForm API

The following attributes and methods are made available for customizing the instantiation of multiforms.

__init__(**args*, ***kwargs*)
The *__init__()* is basically just a pass-through to the children form classes' initialization methods, the only thing that it does is provide special handling for the `initial` parameter. Instead of being a dictionary of initial values, `initial` is now a dictionary of form name, initial data pairs.

```
UserProfileMultiForm(initial={
    'user': {
        # User's initial data
    },
    'profile': {
        # Profile's initial data
    },
})
```

form_classes
This is a dictionary of form name, form class pairs. If the order of the forms is important (for example for output), you can use an OrderedDict instead of a plain dictionary.

get_form_args_kwargs(*key*, *args*, *kwargs*)
This method is available for customizing the instantiation of each form instance. It should return a two-tuple of args and kwargs that will get passed to the child form class that corresponds with the key that is passed in. The default implementation just adds a prefix to each class to prevent field value clashes.

Form API

The following attributes and methods are made available for mimicking the `Form` API.

media

is_bound

cleaned_data
Returns an OrderedDict of the `cleaned_data` for each of the child forms.

is_valid()

non_field_errors()

as_table()

as_ul()

as_p()

is_multipart()

hidden_fields()

visible_fields()

class betterforms.multiform.**MultiModelForm**
MultiModelForm differs from *MultiForm* only in that adds special handling for the `instance` parameter for initialization and has a *save()* method.

__init__(**args*, ***kwargs*)
MultiModelForm's initialization method provides special handling for the `instance` parameter.

Instead of being one object, the `instance` parameter is expected to be a dictionary of form name, instance object pairs.

```
UserProfileMultiForm(instance={
    'user': user,
    'profile': user.profile,
})
```

save(*commit=True*)

The *save()* method will iterate through the child classes and call save on each of them. It returns an OrderedDict of form name, object pairs, where the object is what is returned by the save method of the child form class. Like the `ModelForm.save` method, if `commit` is `False`, *MultiModelForm.save()* will add a `save_m2m` method to the *MultiModelForm* instance to aid in saving the many-to-many relations later.

4.5 Addendum About django-multiform

There is another Django app that provides a similar wrapper called django-multiform that provides essentially the same features as betterform's *MultiForm*. I searched for an app that did this feature when I started work on betterform's version, but couldn't find one. I have looked at django-multiform now and I think that while they are pretty similar, but there are some differences which I think should be noted:

1. django-multiform's `MultiForm` class actually inherits from Django's Form class. I don't think it is very clear if this is a benefit or a disadvantage, but to me it seems that it means that there is Form API that exposed by django-multiform's `MultiForm` that doesn't actually delegate to the child classes.
2. I think that django-multiform's method of dispatching the different values for instance and initial to the child classes is more complicated that it needs to be. Instead of just accepting a dictionary like betterform's *MultiForm* does, with django-multiform, you have to write a *dispatch_init_initial* method.

CHAPTER 5

Changelog

5.1 1.1.1 (2014-08-22)

- (Bugfix) Output both the prefixed and non-prefixed name when the Form is prefixed. [Rocky Meza, #17]

5.2 1.1.0 (2014-08-04)

- Output required for fields even on forms that don't define required_css_class [#16]

5.3 1.0.1 (2014-07-07)

- (Bugfix) Handle None initial values more robustly in MultiForm

5.4 1.0.0 (2014-07-04)

Backwards-incompatible changes:

- Moved all the partials to live the betterforms directory
- Dropped support for Django 1.3

New Features and Bugfixes:

- Support Python 3
- Support Django 1.6
- Add MultiForm and MultiModelForm
- Make NonBraindamagedErrorMixin use error_class
- Use NON_FIELD_ERRORS constant instead of hardcoded value
- Add csrf_exempt argument for form_as_fieldsets
- Add legend attribute to Fieldset

5.5 0.1.3 (2013-10-17)

- Add `betterforms.changelist` module with form classes that assist in listing, searching and filtering querysets.

5.6 0.1.2 (2013-07-25)

- actually update the package.

5.7 0.1.1 (2013-07-25)

- fix form rendering for forms with no fieldsets

5.8 0.1.0 (2013-07-25)

Initial Release

CHAPTER 6

Development

Development for django-betterforms happens on GitHub. Pull requests are welcome. Continuous integration is hosted on Travis CI.

Symbols

A

B

C

F

G

H

I

L

M

N

P

Q

R

S

V