django-superform Documentation

Release 0.3.1

Gregor Müllegger

January 21, 2016

Contents

A `SuperForm` lets you nest other forms and formsets inside a form. That way handling multiple forms on one page gets *super* easy.

Contents:

CHAPTER 1

Quickstart with `django-superform`

TODO.

CHAPTER 2

Available Forms

2.1 `SuperForm`

class `django_superform.forms.`**`SuperForm`**`(`**args*, ***kwargs*`)`

2.2 `SuperFormMixin`

class `django_superform.forms.`**`SuperFormMixin`**`(`**args*, ***kwargs*`)`

The base class for all super forms. It behaves just like a normal django form but will also take composite fields, like *`FormField`* and *`FormSetField`*.

Cleaning, validation, etc. should work totally transparent.

2.3 `SuperModelForm`

class `django_superform.forms.`**`SuperModelForm`**`(`**args*, ***kwargs*`)`

2.4 `SuperModelFormMixin`

class `django_superform.forms.`**`SuperModelFormMixin`**`(`**args*, ***kwargs*`)`

CHAPTER 3

Fields

This is the class hierachy of all the available composite fields to be used in a `SuperForm`:

```
+ CompositeField
|
+-+ FormField
| |
| +-+ ModelFormField
|   |
|   +-- ForeignKeyFormField
|
+-+ FormSetField
  |
  +-+ ModelFormSetField
    |
    +-+ InlineFormSetField
```

3.1 CompositeField

class `django_superform.fields.`**`CompositeField`**(**args*, ***kwargs*)
Implements the base structure that is relevant for all composite fields. This field cannot be used directly, use a subclass of it.

`get_initial`(*form*, *name*)
Get the initial data that got passed into the superform for this composite field. It should return `None` if no initial values where given.

`get_kwargs`(*form*, *name*)
Return the keyword arguments that are used to instantiate the formset.

`get_prefix`(*form*, *name*)
Return the prefix that is used for the formset.

3.2 FormField

class `django_superform.fields.`**`FormField`**(*form_class*, *kwargs=None*, ***field_kwargs*)
A field that can be used to nest a form inside another form:

```
from django import forms
from django_superform import SuperForm

class AddressForm(forms.Form):
    street = forms.CharField()
    city = forms.CharField()

class RegistrationForm(SuperForm):
    first_name = forms.CharField()
    last_name = forms.CharField()
    address = FormField(AddressForm)
```

You can then display the fields in the template with (given that `registration_form` is an instance of `RegistrationForm`):

```
{{ registration_form.address.street }}
{{ registration_form.address.street.errors }}
{{ registration_form.address.city }}
{{ registration_form.address.city.errors }}
```

The fields will all have a prefix in their name so that the naming does not clash with other fields on the page. The name attribute of the input tag for the `street` field in this example will be: `form-address-street`. The name will change if you set a prefix on the superform:

```
form = RegistrationForm(prefix='registration')
```

Then the field name will be `registration-form-address-street`.

You can pass the `kwargs` argument to the `__init__` method in order to give keyword arguments that you want to pass through to the form when it is instaniated. So you could use this to pass in initial values:

```
class RegistrationForm(SuperForm):
    address = FormField(AddressForm, kwargs={
        'initial': {'street': 'Stairway to Heaven 1'}
    })
```

But you can also use nested initial values which you pass into the superform:

```
RegistrationForm(initial={
    'address': {'street': 'Highway to Hell 666'}
})
```

The first method (using `kwargs`) will take precedence.

`get_form`(*form*, *name*)
Get an instance of the form.

`get_form_class`(*form*, *name*)
Return the form class that will be used for instantiation in `get_form`. You can override this method in subclasses to change the behaviour of the given form class.

3.3 `ModelFormField`

class `django_superform.fields.`**`ModelFormField`**(*form_class*, *kwargs=None*, ***field_kwargs*)
This class is the to *`FormField`* what Django's `ModelForm` is to `Form`. It has the same behaviour as *`FormField`* but will also save the nested form if the super form is saved. Here is an example:

```
from django_superform import ModelFormField

class EmailForm(forms.ModelForm):
    class Meta:
        model = EmailAddress
        fields = ('email',)

class UserForm(SuperModelForm):
    email = ModelFormField(EmailForm)

    class Meta:
        model = User
        fields = ('username',)

user_form = UserForm(
    {'username': 'john', 'form-email-email': 'john@example.com'})
if user_form.is_valid():
    user_form.save()
```

This will save the `user_form` and create a new instance of `User` model and it will also save the `EmailForm` and therefore create an instance of `EmailAddress`!

However you usually want to use one of the exsting subclasses, like *`ForeignKeyFormField`* or extend from `ModelFormField` class and override the *`get_instance()`* method.

Note: Usually the *`ModelFormField`* is used inside a *`SuperModelForm`*. You actually can use it within a *`SuperForm`*, but since this form type does not have a `save()` method, you will need to take care of saving the nested model form yourself.

`get_instance`(*form*, *name*)
Provide an instance that shall be used when instantiating the modelform. The `form` argument is the super-form instance that this `ModelFormField` is used in. `name` is the name of this field on the super-form.

This returns `None` by default. So you usually want to override this method in a subclass.

`get_kwargs`(*form*, *name*)
Return the keyword arguments that are used to instantiate the formset.

The `instance` kwarg will be set to the value returned by *`get_instance()`*. The `empty_permitted` kwarg will be set to the inverse of the `required` argument passed into the constructor of this field.

`save`(*form*, *name*, *composite_form*, *commit*)
This method is called by `django_superform.forms.SuperModelForm.save()` in order to save the modelform that this field takes care of and calls on the nested form's `save()` method. But only if *`shall_save()`* returns `True`.

`shall_save`(*form*, *name*, *composite_form*)
Return `True` if the given `composite_form` (the nested form of this field) shall be saved. Return `False` if the form shall not be saved together with the super-form.

By default it will return `False` if the form was not changed and the `empty_permitted` argument for the form was set to `True`. That way you can allow empty forms.

3.4 `ForeignKeyFormField`

class `django_superform.fields.`**`ForeignKeyFormField`**(*form_class*, *kwargs=None*, *field_name=None*, *blank=None*, ***field_kwargs*)

3.5 `FormSetField`

class `django_superform.fields.`**`FormSetField`**(*formset_class*, *kwargs=None*, ***field_kwargs*)

First argument is a formset class that is instantiated by this FormSetField.

You can pass the `kwargs` argument to specify kwargs values that are used when the `formset_class` is instantiated.

3.6 `ModelFormSetField`

class `django_superform.fields.`**`ModelFormSetField`**(*formset_class*, *kwargs=None*, ***field_kwargs*)

3.7 `InlineFormSetField`

class `django_superform.fields.`**`InlineFormSetField`**(*parent_model=None*, *model=None*, *formset_class=None*, *kwargs=None*, ***factory_kwargs*)

The `InlineFormSetField` helps when you want to use a inline formset.

You can pass in either the keyword argument `formset_class` which is a ready to use formset that inherits from `BaseInlineFormSet` or was created by the `inlineformset_factory`.

The other option is to provide the arguments that you would usually pass into the `inlineformset_factory`. The required arguments for that are:

`model` The model class which should be represented by the forms in the formset.

`parent_model` The parent model is the one that is referenced by the model in a foreignkey.

`form` (optional) The model form that is used as a baseclass for the forms in the inline formset.

You can use the `kwargs` keyword argument to pass extra arguments for the formset that are passed through when the formset is instantiated.

All other not mentioned keyword arguments, like `extra`, `max_num` etc. will be passed directly to the `inlineformset_factory`.

Example:

class Gallery(models.Model): name = models.CharField(max_length=50)

class Image(models.Model): gallery = models.ForeignKey(Gallery) image = models.ImageField(...)

class GalleryForm(ModelFormWithFormSets):

class Meta: model = Gallery fields = (‘name’,)

images = InlineFormSetField(parent_model=Gallery, model=Image, extra=1)

TODO: This document is quite raw. Needs improvement.

Form class needs to subclass from *SuperForm* or *SuperModelForm*.

During instantiation:

- composite fields get initialized
- The fields `get_form` and `get_formsets` methods are called which instantiate the nested form/formset. They get the same data/files that are passed into the super form. Initial values are passed through. EXAMPLE.
- Those get attached into `form.forms` and `form.formsets`.

In template you can get a bound field (like with django's normal form fields) with {{ form.composite_field_name }}. Or you can get the real form instance with `{{ form.forms.composite_field_name }}`, or the formset: `{{ form.formsets.composite_field_name }}`.

Then when it gets to validation, the super form's `full_clean()` and `is_valid()` methods will clean and validate the nested forms/formsets as well. So `is_valid()` will return `False` when the super form's fields are valid but any of the nested forms/formsets is not.

Errors will be attached to `form.errors`. For forms it will be a error dict, for formsets it will be a list of the errors of the formset's forms.

CHAPTER 4

On saving `SuperModelForm`

The super form's `save()` method will first save the model that it takes care of. Then the nested forms and then the nested formsets. It will only return the saved model from the super form, but none of the objects from nested forms/formsets. This is to keep the API to the normal model forms the same.

The `commit` argument is respected and passed down. So nothing is saved to the DB if you don't want it to. In that case, django forms will get a dynamically created `save_m2m` method that can be called later on to then save all the related stuff. The super form hooks in there to also save the nested forms and formsets then (TODO: check, really?). And ofcourse it calls their `save_m2m` methods :)

CHAPTER 5

Changelog

5.1 0.3.1

- `SuperForm.composite_fields` dict is not longer shared between form instances. Every new form instances get's a deep copy. So changes to it won't leak into other instances of the same form class.

5.2 0.3.0

- #11: Fix `CompositeBoundField` to allow direct access to nested form fields via `form['nested_form']['field']`.
- Support for Django's Media handling in nested forms. See #3 and #5.
- Do not populate errorlist representations without any errors of nested formsets into the errors of the super form. See #5 for details.

5.3 0.2.0

- Django 1.8 support.
- Initial values given to the `__init__` method of the super-form will get passed through to the nested forms.
- The `empty_permitted` argument for modelforms used in a `ModelFormField` is set depending on the `required` attribute given to the field.

5.4 0.1.0

- Initial release with proof of concept.

genindex | modindex | search

Index