Model formsets - - -

***class models.BaseModelFormSet***

Like regular formsets, Django provides a couple of enhanced formset classes to make working with Django

models more convenient. Let’s reuse the Author model from above:

>>> from django.forms import modelformset\_factory

>>> from myapp.models import Author

>>> AuthorFormSet = modelformset\_factory(Author, fields=["name", "title"])

Using fields restricts the formset to use only the given fields. Alternatively, you can take an “opt-out”

approach, specifying which fields to exclude:

>>> AuthorFormSet = modelformset\_factory(Author, exclude=["birth\_date"])

This will create a formset that is capable of working with the data associated with the Author model. It

works just like a regular formset:

>>> formset = AuthorFormSet()

>>> print(formset)

<input type="hidden" name="form-TOTAL\_FORMS" value="1" id="id\_form-TOTAL\_FORMS"><input type="hidden

,→" name="form-INITIAL\_FORMS" value="0" id="id\_form-INITIAL\_FORMS"><input type="hidden" name="form-

,→MIN\_NUM\_FORMS" value="0" id="id\_form-MIN\_NUM\_FORMS"><input type="hidden" name="form-MAX\_NUM\_FORMS

,→" value="1000" id="id\_form-MAX\_NUM\_FORMS">

<div><label for="id\_form-0-name">Name:</label><input id="id\_form-0-name" type="text" name="form-0-

,→name" maxlength="100"></div>

<div><label for="id\_form-0-title">Title:</label><select name="form-0-title" id="id\_form-0-title">

<option value="" selected>---------</option>

<option value="MR">Mr.</option>

<option value="MRS">Mrs.</option>

<option value="MS">Ms.</option>

</select><input type="hidden" name="form-0-id" id="id\_form-0-id"></div>

**Note: modelformset\_factory()** uses **formset\_factory()** to generate formsets. This means that a model formset is an extension of a basic formset that knows how to interact with a particular model.

**Note:** When using **multi-table inheritance**, forms generated by a formset factory will contain a parent link field (by default **<parent\_model\_name>\_ptr)** instead of an id field.

# Changing the queryset

By default, when you create a **formset** from a model, the **formset** will use a queryset that includes all objects in the model (e.g., **Author.objects.all()**). You can override this behavior by using the queryset argument:

>>> formset = AuthorFormSet(queryset=Author.objects.filter(name\_\_startswith="O"))

Alternatively, you can create a subclass that sets self.queryset in \_\_init\_\_:

from django.forms import BaseModelFormSet

from myapp.models import Author

class BaseAuthorFormSet(BaseModelFormSet):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.queryset = Author.objects.filter(name\_\_startswith="O")

Then, pass your **BaseAuthorFormSet** class to the factory function:

>>> AuthorFormSet = modelformset\_factory(

... Author, fields=["name", "title"], formset=BaseAuthorFormSet

... )

If you want to return a formset that doesn’t include any preexisting instances of the model, you can specify

an empty QuerySet:

>>> AuthorFormSet(queryset=Author.objects.none())

# Changing the form - - -

By default, when you use **modelformset\_factory**, a model form will be created using **modelform\_factory().**

Often, it can be useful to specify a custom model form. For example, you can create a custom model form

that has custom validation:

class AuthorForm(forms.ModelForm):

class Meta:

model = Author

fields = [“name”, “title”]

def clean\_name(self):

# custom validation for the name field

…

Then, pass your model form to the factory function:

AuthorFormSet = modelformset\_factory(Author, form=AuthorForm)

It is not always necessary to define a custom model form. The modelformset\_factory function has several

arguments which are passed through to modelform\_factory, which are described below

## Specifying widgets to use in the form with widgets

Using the widgets parameter, you can specify a dictionary of values to customize the ModelForm’s widget class for a particular field. This works the same way as the widgets dictionary on the inner Meta class of a ModelForm works:

>>> AuthorFormSet = modelformset\_factory(

... Author,

... fields=["name", "title"],

... widgets={"name": Textarea(attrs={"cols": 80, "rows": 20})},

... )

## Enabling localization for fields with localized\_fields

Using the localized\_fields parameter, you can enable localization for fields in the form.

>>> AuthorFormSet = modelformset\_factory(

... Author, fields=['name', 'title', 'birth\_date'],

... localized\_fields=['birth\_date'])

If **localized\_fields** is set to the special value **'\_\_all\_\_'**, all fields will be localized.

## Providing initial values

As with regular **formsets**, it’s possible to specify initial data for forms in the **formset** by specifying an initial parameter when instantiating the model **formset** class returned by **modelformset\_factory().** However, with model **formsets**, the initial values only apply to extra forms, those that aren’t attached to an existing model instance. If the length of initial exceeds the number of extra forms, the excess initial data is ignored. If the extra forms with initial data aren’t changed by the user, they won’t be validated or saved.

### Saving objects in the formset

As with a ModelForm, you can save the data as a model object. This is done with the formset’s **save()** method:

# Create a formset instance with POST data.

>>> formset = AuthorFormSet(request.POST)

# Assuming all is valid, save the data.

>>> instances = formset.save()

The **save()** method returns the instances that have been saved to the database. If a given instance’s data didn’t change in the bound data, the instance won’t be saved to the database and won’t be included in the return value (instances, in the above example).

When fields are missing from the form (for example because they have been excluded), these fields will not be set by the **save()** method. You can find more information about this restriction, which also holds for regular **ModelForms**, in Selecting the fields to use.

Pass **commit=False** to return the unsaved model instances:

# don't save to the database

>>> instances = formset.save(commit=False)

>>> for instance in instances:

... # do something with instance

... instance.save()

...

This gives you the ability to attach data to the instances before saving them to the database. If your formset contains a **ManyToManyField**, you’ll also need to call **formset.save\_m2m()** to ensure the many-to-many

relationships are saved properly.

After calling save(), your model formset will have three new attributes containing the formset’s changes:

models.BaseModelFormSet.changed\_objects

models.BaseModelFormSet.deleted\_objects

models.BaseModelFormSet.new\_objects

### Limiting the number of editable objects

As with regular formsets, you can use the max\_num and extra parameters to **modelformset\_factory()** to limit the number of extra forms displayed.

**max\_num** does not prevent existing objects from being displayed:

>>> Author.objects.order\_by("name")

<QuerySet [<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]>

>>> AuthorFormSet = modelformset\_factory(Author, fields=["name"], max\_num=1)

>>> formset = AuthorFormSet(queryset=Author.objects.order\_by("name"))

>>> [x.name for x in formset.get\_queryset()]

['Charles Baudelaire', 'Paul Verlaine', 'Walt Whitman']

Also, extra=0 doesn’t prevent creation of new model instances as you can add additional forms with JavaScript or send additional POST data. See Preventing new objects creation on how to do this. If the value of **max\_num** is greater than the number of existing related objects, up to extra additional blank forms will be added to the **formset**, so long as the total number of forms does not exceed **max\_num**:

>>> AuthorFormSet = modelformset\_factory(Author, fields=["name"], max\_num=4, extra=2)

>>> formset = AuthorFormSet(queryset=Author.objects.order\_by("name"))

>>> for form in formset:

... print(form)

...

<div><label for="id\_form-0-name">Name:</label><input id="id\_form-0-name" type="text" name="form-0-

,→name" value="Charles Baudelaire" maxlength="100"><input type="hidden" name="form-0-id" value="1"␣

,→id="id\_form-0-id"></div>

<div><label for="id\_form-1-name">Name:</label><input id="id\_form-1-name" type="text" name="form-1-

,→name" value="Paul Verlaine" maxlength="100"><input type="hidden" name="form-1-id" value="3" id=

,→"id\_form-1-id"></div>

<div><label for="id\_form-2-name">Name:</label><input id="id\_form-2-name" type="text" name="form-2-

,→name" value="Walt Whitman" maxlength="100"><input type="hidden" name="form-2-id" value="2" id=

,→"id\_form-2-id"></div>

<div><label for="id\_form-3-name">Name:</label><input id="id\_form-3-name" type="text" name="form-3-

,→name" maxlength="100"><input type="hidden" name="form-3-id" id="id\_form-3-id"></div>

A **max\_num** value of None (the default) puts a high limit on the number of forms displayed (1000). In practice this is equivalent to no limit.

## Preventing new objects creation

Using the edit\_only parameter, you can prevent creation of any new objects:

>>> AuthorFormSet = modelformset\_factory(

... Author,

... fields=["name", "title"],

... edit\_only=True,

... )

Here, the formset will only edit existing Author instances. No other objects will be created or edited.

### Using a model formset in a view

Model formsets are very similar to formsets. Let’s say we want to present a formset to edit Author model

instances:

from django.forms import modelformset\_factory

from django.shortcuts import render

from myapp.models import Author

def manage\_authors(request):

AuthorFormSet = modelformset\_factory(Author, fields=["name", "title"])

if request.method == "POST":

formset = AuthorFormSet(request.POST, request.FILES)

if formset.is\_valid():

formset.save()

# do something.

else:

formset = AuthorFormSet()

return render(request, "manage\_authors.html", {"formset": formset})

As you can see, the view logic of a model formset isn’t drastically different than that of a “normal” formset. The only difference is that we call **formset.save()** to save the data into the database. (This was described above, in Saving objects in the formset.)

### Overriding clean() on a ModelFormSet

Just like with ModelForms, by default the clean() method of a ModelFormSet will validate that none of

the items in the formset violate the unique constraints on your model (either unique, unique\_together

or unique\_for\_date|month|year). If you want to override the clean() method on a ModelFormSet and

maintain this validation, you must call the parent class’s clean method:

from django.forms import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):

def clean(self):

super().clean()

# example custom validation across forms in the formset

for form in self.forms:

# your custom formset validation

...

Also note that by the time you reach this step, individual model instances have already been created for each Form. Modifying a value in form.cleaned\_data is not sufficient to affect the saved value. If you wish to modify a value in ModelFormSet.clean() you must modify form.instance:

from django.forms import BaseModelFormSet

class MyModelFormSet(BaseModelFormSet):

def clean(self):

super().clean()

for form in self.forms:

name = form.cleaned\_data["name"].upper()

form.cleaned\_data["name"] = name

# update the instance value.

form.instance.name = name

## Using a custom queryset

As stated earlier, you can override the default queryset used by the model formset:

from django.forms import modelformset\_factory

from django.shortcuts import render

from myapp.models import Author

def manage\_authors(request):

AuthorFormSet = modelformset\_factory(Author, fields=["name", "title"])

queryset = Author.objects.filter(name\_\_startswith="O")

if request.method == "POST":

formset = AuthorFormSet(

request.POST,

request.FILES,

queryset=queryset,

)

if formset.is\_valid():

formset.save()

# Do something.

else:

formset = AuthorFormSet(queryset=queryset)

return render(request, "manage\_authors.html", {"formset": formset})

Note that we pass the queryset argument in both the POST and GET cases in this example.

## Using the formset in the template

There are three ways to render a formset in a Django template.

First, you can let the formset do most of the work:

<form method="post">

{{ formset }}

</form>

Second, you can manually render the formset, but let the form deal with itself:

<form method="post">

{{ formset.management\_form }}

{% for form in formset %}

{{ form }}

{% endfor %}

</form>

When you manually render the forms yourself, be sure to render the management form as shown above.

Third, you can manually render each field:

<form method="post">

{{ formset.management\_form }}

{% for form in formset %}

{% for field in form %}

{{ field.label\_tag }} {{ field }}

{% endfor %}

{% endfor %}

</form>

If you opt to use this third method and you don’t iterate over the fields with a {% for %} loop, you’ll need to render the primary key field. For example, if you were rendering the name and age fields of a model:

<form method="post">

{{ formset.management\_form }}

{% for form in formset %}

{{ form.id }}

<ul>

<li>{{ form.name }}</li>

<li>{{ form.age }}</li>

</ul>

{% endfor %}

</form>

Notice how we need to explicitly render **{{ form.id }}.** This ensures that the model formset, in the **POST** case, will work correctly. (This example assumes a primary key named id. If you’ve explicitly defined your own primary key that isn’t called id, make sure it gets rendered.)

## Inline formsets

### class models.BaseInlineFormSet

Inline formsets is a small abstraction layer on top of model formsets. These simplify the case of working with

related objects via a foreign key. Suppose you have these two models:

from django.db import models

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

title = models.CharField(max\_length=100)

If you want to create a formset that allows you to edit books belonging to a particular author, you could do

this:

>>> from django.forms import inlineformset\_factory

>>> BookFormSet = inlineformset\_factory(Author, Book, fields=["title"])

>>> author = Author.objects.get(name="Mike Royko")

>>> formset = BookFormSet(instance=author)

**BookFormSet’s** prefix is **'book\_set'** (<**model name>\_set** ). If Book’s **ForeignKey** to Author has a **related\_name**, that’s used instead.

Note: **inlineformset\_factory()** uses **modelformset\_factory()** and marks **can\_delete=True**.

# Overriding methods on an InlineFormSet

When overriding methods on InlineFormSet, you should subclass BaseInlineFormSet rather than

BaseModelFormSet.

For example, if you want to override clean():

from django.forms import BaseInlineFormSet

class CustomInlineFormSet(BaseInlineFormSet):

def clean(self):

super().clean()

# example custom validation across forms in the formset

for form in self.forms:

# your custom formset validation

...

See also Overriding clean() on a **ModelFormSet**.

Then when you create your inline formset, pass in the optional argument formset:

>>> from django.forms import inlineformset\_factory

>>> BookFormSet = inlineformset\_factory(

... Author, Book, fields=["title"], formset=CustomInlineFormSet

... )

>>> author = Author.objects.get(name="Mike Royko")

>>> formset = BookFormSet(instance=author)

### More than one foreign key to the same model

If your model contains more than one foreign key to the same model, you’ll need to resolve the ambiguity

manually using fk\_name. For example, consider the following model:

class Friendship(models.Model):

from\_friend = models.ForeignKey(

Friend,

on\_delete=models.CASCADE,

related\_name="from\_friends",

)

to\_friend = models.ForeignKey(

Friend,

on\_delete=models.CASCADE,

related\_name="friends",

)

length\_in\_months = models.IntegerField()

To resolve this, you can use fk\_name to inlineformset\_factory():

>>> FriendshipFormSet = inlineformset\_factory(

... Friend, Friendship, fk\_name="from\_friend", fields=["to\_friend", "length\_in\_months"]

... )

## Using an inline formset in a view

You may want to provide a view that allows a user to edit the related objects of a model. Here’s how you can

do that:

def manage\_books(request, author\_id):

author = Author.objects.get(pk=author\_id)

BookInlineFormSet = inlineformset\_factory(Author, Book, fields=["title"])

if request.method == "POST":

formset = BookInlineFormSet(request.POST, request.FILES, instance=author)

if formset.is\_valid():

formset.save()

# Do something. Should generally end with a redirect. For example:

return HttpResponseRedirect(author.get\_absolute\_url())

else:

formset = BookInlineFormSet(instance=author)

return render(request, "manage\_books.html", {"formset": formset})

Notice how we pass instance in both the POST and GET cases.

### Specifying widgets to use in the inline form

**inlineformset\_factory** uses **modelformset\_factory** and passes most of its arguments to **modelformset\_factory**. This means you can use the widgets parameter in much the same way as passing it to **modelformset\_factory**.

------------------------- END -----------------------------