(myvenv) C:\Users\user\Desktop\django>type C:\Users\user\AppData\Local\Programs\Python\Python313\Lib\site-packages\django\contrib\admin\options.py

import copy

import enum

import json

import re

import warnings

from functools import partial, update\_wrapper

from urllib.parse import parse\_qsl

from urllib.parse import quote as urlquote

from urllib.parse import urlparse

from django import forms

from django.conf import settings

from django.contrib import messages

from django.contrib.admin import helpers, widgets

from django.contrib.admin.checks import (

BaseModelAdminChecks,

InlineModelAdminChecks,

ModelAdminChecks,

)

from django.contrib.admin.exceptions import DisallowedModelAdminToField, NotRegistered

from django.contrib.admin.templatetags.admin\_urls import add\_preserved\_filters

from django.contrib.admin.utils import (

NestedObjects,

construct\_change\_message,

flatten\_fieldsets,

get\_deleted\_objects,

lookup\_spawns\_duplicates,

model\_format\_dict,

model\_ngettext,

quote,

unquote,

)

from django.contrib.admin.widgets import AutocompleteSelect, AutocompleteSelectMultiple

from django.contrib.auth import get\_permission\_codename

from django.core.exceptions import (

FieldDoesNotExist,

FieldError,

PermissionDenied,

ValidationError,

)

from django.core.paginator import Paginator

from django.db import models, router, transaction

from django.db.models.constants import LOOKUP\_SEP

from django.forms.formsets import DELETION\_FIELD\_NAME, all\_valid

from django.forms.models import (

BaseInlineFormSet,

inlineformset\_factory,

modelform\_defines\_fields,

modelform\_factory,

modelformset\_factory,

)

from django.forms.widgets import CheckboxSelectMultiple, SelectMultiple

from django.http import HttpResponseRedirect

from django.http.response import HttpResponseBase

from django.template.response import SimpleTemplateResponse, TemplateResponse

from django.urls import reverse

from django.utils.decorators import method\_decorator

from django.utils.deprecation import RemovedInDjango60Warning

from django.utils.html import format\_html

from django.utils.http import urlencode

from django.utils.safestring import mark\_safe

from django.utils.text import (

capfirst,

format\_lazy,

get\_text\_list,

smart\_split,

unescape\_string\_literal,

)

from django.utils.translation import gettext as \_

from django.utils.translation import ngettext

from django.views.decorators.csrf import csrf\_protect

from django.views.generic import RedirectView

IS\_POPUP\_VAR = "\_popup"

TO\_FIELD\_VAR = "\_to\_field"

IS\_FACETS\_VAR = "\_facets"

class ShowFacets(enum.Enum):

NEVER = "NEVER"

ALLOW = "ALLOW"

ALWAYS = "ALWAYS"

HORIZONTAL, VERTICAL = 1, 2

def get\_content\_type\_for\_model(obj):

# Since this module gets imported in the application's root package,

# it cannot import models from other applications at the module level.

from django.contrib.contenttypes.models import ContentType

return ContentType.objects.get\_for\_model(obj, for\_concrete\_model=False)

def get\_ul\_class(radio\_style):

return "radiolist" if radio\_style == VERTICAL else "radiolist inline"

class IncorrectLookupParameters(Exception):

pass

# Defaults for formfield\_overrides. ModelAdmin subclasses can change this

# by adding to ModelAdmin.formfield\_overrides.

FORMFIELD\_FOR\_DBFIELD\_DEFAULTS = {

models.DateTimeField: {

"form\_class": forms.SplitDateTimeField,

"widget": widgets.AdminSplitDateTime,

},

models.DateField: {"widget": widgets.AdminDateWidget},

models.TimeField: {"widget": widgets.AdminTimeWidget},

models.TextField: {"widget": widgets.AdminTextareaWidget},

models.URLField: {"widget": widgets.AdminURLFieldWidget},

models.IntegerField: {"widget": widgets.AdminIntegerFieldWidget},

models.BigIntegerField: {"widget": widgets.AdminBigIntegerFieldWidget},

models.CharField: {"widget": widgets.AdminTextInputWidget},

models.ImageField: {"widget": widgets.AdminFileWidget},

models.FileField: {"widget": widgets.AdminFileWidget},

models.EmailField: {"widget": widgets.AdminEmailInputWidget},

models.UUIDField: {"widget": widgets.AdminUUIDInputWidget},

}

csrf\_protect\_m = method\_decorator(csrf\_protect)

class BaseModelAdmin(metaclass=forms.MediaDefiningClass):

"""Functionality common to both ModelAdmin and InlineAdmin."""

autocomplete\_fields = ()

raw\_id\_fields = ()

fields = None

exclude = None

fieldsets = None

form = forms.ModelForm

filter\_vertical = ()

filter\_horizontal = ()

radio\_fields = {}

prepopulated\_fields = {}

formfield\_overrides = {}

readonly\_fields = ()

ordering = None

sortable\_by = None

view\_on\_site = True

show\_full\_result\_count = True

checks\_class = BaseModelAdminChecks

def check(self, \*\*kwargs):

return self.checks\_class().check(self, \*\*kwargs)

def \_\_init\_\_(self):

# Merge FORMFIELD\_FOR\_DBFIELD\_DEFAULTS with the formfield\_overrides

# rather than simply overwriting.

overrides = copy.deepcopy(FORMFIELD\_FOR\_DBFIELD\_DEFAULTS)

for k, v in self.formfield\_overrides.items():

overrides.setdefault(k, {}).update(v)

self.formfield\_overrides = overrides

def formfield\_for\_dbfield(self, db\_field, request, \*\*kwargs):

"""

Hook for specifying the form Field instance for a given database Field

instance.

If kwargs are given, they're passed to the form Field's constructor.

"""

# If the field specifies choices, we don't need to look for special

# admin widgets - we just need to use a select widget of some kind.

if db\_field.choices:

return self.formfield\_for\_choice\_field(db\_field, request, \*\*kwargs)

# ForeignKey or ManyToManyFields

if isinstance(db\_field, (models.ForeignKey, models.ManyToManyField)):

# Combine the field kwargs with any options for formfield\_overrides.

# Make sure the passed in \*\*kwargs override anything in

# formfield\_overrides because \*\*kwargs is more specific, and should

# always win.

if db\_field.\_\_class\_\_ in self.formfield\_overrides:

kwargs = {\*\*self.formfield\_overrides[db\_field.\_\_class\_\_], \*\*kwargs}

# Get the correct formfield.

if isinstance(db\_field, models.ForeignKey):

formfield = self.formfield\_for\_foreignkey(db\_field, request, \*\*kwargs)

elif isinstance(db\_field, models.ManyToManyField):

formfield = self.formfield\_for\_manytomany(db\_field, request, \*\*kwargs)

# For non-raw\_id fields, wrap the widget with a wrapper that adds

# extra HTML -- the "add other" interface -- to the end of the

# rendered output. formfield can be None if it came from a

# OneToOneField with parent\_link=True or a M2M intermediary.

if formfield and db\_field.name not in self.raw\_id\_fields:

try:

related\_modeladmin = self.admin\_site.get\_model\_admin(

db\_field.remote\_field.model

)

except NotRegistered:

wrapper\_kwargs = {}

else:

wrapper\_kwargs = {

"can\_add\_related": related\_modeladmin.has\_add\_permission(

request

),

"can\_change\_related": related\_modeladmin.has\_change\_permission(

request

),

"can\_delete\_related": related\_modeladmin.has\_delete\_permission(

request

),

"can\_view\_related": related\_modeladmin.has\_view\_permission(

request

),

}

formfield.widget = widgets.RelatedFieldWidgetWrapper(

formfield.widget,

db\_field.remote\_field,

self.admin\_site,

\*\*wrapper\_kwargs,

)

return formfield

# If we've got overrides for the formfield defined, use 'em. \*\*kwargs

# passed to formfield\_for\_dbfield override the defaults.

for klass in db\_field.\_\_class\_\_.mro():

if klass in self.formfield\_overrides:

kwargs = {\*\*copy.deepcopy(self.formfield\_overrides[klass]), \*\*kwargs}

return db\_field.formfield(\*\*kwargs)

# For any other type of field, just call its formfield() method.

return db\_field.formfield(\*\*kwargs)

def formfield\_for\_choice\_field(self, db\_field, request, \*\*kwargs):

"""

Get a form Field for a database Field that has declared choices.

"""

# If the field is named as a radio\_field, use a RadioSelect

if db\_field.name in self.radio\_fields:

# Avoid stomping on custom widget/choices arguments.

if "widget" not in kwargs:

kwargs["widget"] = widgets.AdminRadioSelect(

attrs={

"class": get\_ul\_class(self.radio\_fields[db\_field.name]),

}

)

if "choices" not in kwargs:

kwargs["choices"] = db\_field.get\_choices(

include\_blank=db\_field.blank, blank\_choice=[("", \_("None"))]

)

return db\_field.formfield(\*\*kwargs)

def get\_field\_queryset(self, db, db\_field, request):

"""

If the ModelAdmin specifies ordering, the queryset should respect that

ordering. Otherwise don't specify the queryset, let the field decide

(return None in that case).

"""

try:

related\_admin = self.admin\_site.get\_model\_admin(db\_field.remote\_field.model)

except NotRegistered:

return None

else:

ordering = related\_admin.get\_ordering(request)

if ordering is not None and ordering != ():

return db\_field.remote\_field.model.\_default\_manager.using(db).order\_by(

\*ordering

)

return None

def formfield\_for\_foreignkey(self, db\_field, request, \*\*kwargs):

"""

Get a form Field for a ForeignKey.

"""

db = kwargs.get("using")

if "widget" not in kwargs:

if db\_field.name in self.get\_autocomplete\_fields(request):

kwargs["widget"] = AutocompleteSelect(

db\_field, self.admin\_site, using=db

)

elif db\_field.name in self.raw\_id\_fields:

kwargs["widget"] = widgets.ForeignKeyRawIdWidget(

db\_field.remote\_field, self.admin\_site, using=db

)

elif db\_field.name in self.radio\_fields:

kwargs["widget"] = widgets.AdminRadioSelect(

attrs={

"class": get\_ul\_class(self.radio\_fields[db\_field.name]),

}

)

kwargs["empty\_label"] = (

kwargs.get("empty\_label", \_("None")) if db\_field.blank else None

)

if "queryset" not in kwargs:

queryset = self.get\_field\_queryset(db, db\_field, request)

if queryset is not None:

kwargs["queryset"] = queryset

return db\_field.formfield(\*\*kwargs)

def formfield\_for\_manytomany(self, db\_field, request, \*\*kwargs):

"""

Get a form Field for a ManyToManyField.

"""

# If it uses an intermediary model that isn't auto created, don't show

# a field in admin.

if not db\_field.remote\_field.through.\_meta.auto\_created:

return None

db = kwargs.get("using")

if "widget" not in kwargs:

autocomplete\_fields = self.get\_autocomplete\_fields(request)

if db\_field.name in autocomplete\_fields:

kwargs["widget"] = AutocompleteSelectMultiple(

db\_field,

self.admin\_site,

using=db,

)

elif db\_field.name in self.raw\_id\_fields:

kwargs["widget"] = widgets.ManyToManyRawIdWidget(

db\_field.remote\_field,

self.admin\_site,

using=db,

)

elif db\_field.name in [\*self.filter\_vertical, \*self.filter\_horizontal]:

kwargs["widget"] = widgets.FilteredSelectMultiple(

db\_field.verbose\_name, db\_field.name in self.filter\_vertical

)

if "queryset" not in kwargs:

queryset = self.get\_field\_queryset(db, db\_field, request)

if queryset is not None:

kwargs["queryset"] = queryset

form\_field = db\_field.formfield(\*\*kwargs)

if (

isinstance(form\_field.widget, SelectMultiple)

and form\_field.widget.allow\_multiple\_selected

and not isinstance(

form\_field.widget, (CheckboxSelectMultiple, AutocompleteSelectMultiple)

)

):

msg = \_(

"Hold down ?ontrol?? or ?ommand??on a Mac, to select more than one."

)

help\_text = form\_field.help\_text

form\_field.help\_text = (

format\_lazy("{} {}", help\_text, msg) if help\_text else msg

)

return form\_field

def get\_autocomplete\_fields(self, request):

"""

Return a list of ForeignKey and/or ManyToMany fields which should use

an autocomplete widget.

"""

return self.autocomplete\_fields

def get\_view\_on\_site\_url(self, obj=None):

if obj is None or not self.view\_on\_site:

return None

if callable(self.view\_on\_site):

return self.view\_on\_site(obj)

elif hasattr(obj, "get\_absolute\_url"):

# use the ContentType lookup if view\_on\_site is True

return reverse(

"admin:view\_on\_site",

kwargs={

"content\_type\_id": get\_content\_type\_for\_model(obj).pk,

"object\_id": obj.pk,

},

current\_app=self.admin\_site.name,

)

def get\_empty\_value\_display(self):

"""

Return the empty\_value\_display set on ModelAdmin or AdminSite.

"""

try:

return mark\_safe(self.empty\_value\_display)

except AttributeError:

return mark\_safe(self.admin\_site.empty\_value\_display)

def get\_exclude(self, request, obj=None):

"""

Hook for specifying exclude.

"""

return self.exclude

def get\_fields(self, request, obj=None):

"""

Hook for specifying fields.

"""

if self.fields:

return self.fields

# \_get\_form\_for\_get\_fields() is implemented in subclasses.

form = self.\_get\_form\_for\_get\_fields(request, obj)

return [\*form.base\_fields, \*self.get\_readonly\_fields(request, obj)]

def get\_fieldsets(self, request, obj=None):

"""

Hook for specifying fieldsets.

"""

if self.fieldsets:

return self.fieldsets

return [(None, {"fields": self.get\_fields(request, obj)})]

def get\_inlines(self, request, obj):

"""Hook for specifying custom inlines."""

return self.inlines

def get\_ordering(self, request):

"""

Hook for specifying field ordering.

"""

return self.ordering or () # otherwise we might try to \*None, which is bad ;)

def get\_readonly\_fields(self, request, obj=None):

"""

Hook for specifying custom readonly fields.

"""

return self.readonly\_fields

def get\_prepopulated\_fields(self, request, obj=None):

"""

Hook for specifying custom prepopulated fields.

"""

return self.prepopulated\_fields

def get\_queryset(self, request):

"""

Return a QuerySet of all model instances that can be edited by the

admin site. This is used by changelist\_view.

"""

qs = self.model.\_default\_manager.get\_queryset()

# TODO: this should be handled by some parameter to the ChangeList.

ordering = self.get\_ordering(request)

if ordering:

qs = qs.order\_by(\*ordering)

return qs

def get\_sortable\_by(self, request):

"""Hook for specifying which fields can be sorted in the changelist."""

return (

self.sortable\_by

if self.sortable\_by is not None

else self.get\_list\_display(request)

)

# RemovedInDjango60Warning: when the deprecation ends, replace with:

# def lookup\_allowed(self, lookup, value, request):

def lookup\_allowed(self, lookup, value, request=None):

from django.contrib.admin.filters import SimpleListFilter

model = self.model

# Check FKey lookups that are allowed, so that popups produced by

# ForeignKeyRawIdWidget, on the basis of ForeignKey.limit\_choices\_to,

# are allowed to work.

for fk\_lookup in model.\_meta.related\_fkey\_lookups:

# As ``limit\_choices\_to`` can be a callable, invoke it here.

if callable(fk\_lookup):

fk\_lookup = fk\_lookup()

if (lookup, value) in widgets.url\_params\_from\_lookup\_dict(

fk\_lookup

).items():

return True

relation\_parts = []

prev\_field = None

parts = lookup.split(LOOKUP\_SEP)

for part in parts:

try:

field = model.\_meta.get\_field(part)

except FieldDoesNotExist:

# Lookups on nonexistent fields are ok, since they're ignored

# later.

break

if not prev\_field or (

prev\_field.is\_relation

and field not in model.\_meta.parents.values()

and field is not model.\_meta.auto\_field

and (

model.\_meta.auto\_field is None

or part not in getattr(prev\_field, "to\_fields", [])

)

and (field.is\_relation or not field.primary\_key)

):

relation\_parts.append(part)

if not getattr(field, "path\_infos", None):

# This is not a relational field, so further parts

# must be transforms.

break

prev\_field = field

model = field.path\_infos[-1].to\_opts.model

if len(relation\_parts) <= 1:

# Either a local field filter, or no fields at all.

return True

valid\_lookups = {self.date\_hierarchy}

# RemovedInDjango60Warning: when the deprecation ends, replace with:

# for filter\_item in self.get\_list\_filter(request):

list\_filter = (

self.get\_list\_filter(request) if request is not None else self.list\_filter

)

for filter\_item in list\_filter:

if isinstance(filter\_item, type) and issubclass(

filter\_item, SimpleListFilter

):

valid\_lookups.add(filter\_item.parameter\_name)

elif isinstance(filter\_item, (list, tuple)):

valid\_lookups.add(filter\_item[0])

else:

valid\_lookups.add(filter\_item)

# Is it a valid relational lookup?

return not {

LOOKUP\_SEP.join(relation\_parts),

LOOKUP\_SEP.join(relation\_parts + [part]),

}.isdisjoint(valid\_lookups)

def to\_field\_allowed(self, request, to\_field):

"""

Return True if the model associated with this admin should be

allowed to be referenced by the specified field.

"""

try:

field = self.opts.get\_field(to\_field)

except FieldDoesNotExist:

return False

# Always allow referencing the primary key since it's already possible

# to get this information from the change view URL.

if field.primary\_key:

return True

# Allow reverse relationships to models defining m2m fields if they

# target the specified field.

for many\_to\_many in self.opts.many\_to\_many:

if many\_to\_many.m2m\_target\_field\_name() == to\_field:

return True

# Make sure at least one of the models registered for this site

# references this field through a FK or a M2M relationship.

registered\_models = set()

for model, admin in self.admin\_site.\_registry.items():

registered\_models.add(model)

for inline in admin.inlines:

registered\_models.add(inline.model)

related\_objects = (

f

for f in self.opts.get\_fields(include\_hidden=True)

if (f.auto\_created and not f.concrete)

)

for related\_object in related\_objects:

related\_model = related\_object.related\_model

remote\_field = related\_object.field.remote\_field

if (

any(issubclass(model, related\_model) for model in registered\_models)

and hasattr(remote\_field, "get\_related\_field")

and remote\_field.get\_related\_field() == field

):

return True

return False

def has\_add\_permission(self, request):

"""

Return True if the given request has permission to add an object.

Can be overridden by the user in subclasses.

"""

opts = self.opts

codename = get\_permission\_codename("add", opts)

return request.user.has\_perm("%s.%s" % (opts.app\_label, codename))

def has\_change\_permission(self, request, obj=None):

"""

Return True if the given request has permission to change the given

Django model instance, the default implementation doesn't examine the

`obj` parameter.

Can be overridden by the user in subclasses. In such case it should

return True if the given request has permission to change the `obj`

model instance. If `obj` is None, this should return True if the given

request has permission to change \*any\* object of the given type.

"""

opts = self.opts

codename = get\_permission\_codename("change", opts)

return request.user.has\_perm("%s.%s" % (opts.app\_label, codename))

def has\_delete\_permission(self, request, obj=None):

"""

Return True if the given request has permission to delete the given

Django model instance, the default implementation doesn't examine the

`obj` parameter.

Can be overridden by the user in subclasses. In such case it should

return True if the given request has permission to delete the `obj`

model instance. If `obj` is None, this should return True if the given

request has permission to delete \*any\* object of the given type.

"""

opts = self.opts

codename = get\_permission\_codename("delete", opts)

return request.user.has\_perm("%s.%s" % (opts.app\_label, codename))

def has\_view\_permission(self, request, obj=None):

"""

Return True if the given request has permission to view the given

Django model instance. The default implementation doesn't examine the

`obj` parameter.

If overridden by the user in subclasses, it should return True if the

given request has permission to view the `obj` model instance. If `obj`

is None, it should return True if the request has permission to view

any object of the given type.

"""

opts = self.opts

codename\_view = get\_permission\_codename("view", opts)

codename\_change = get\_permission\_codename("change", opts)

return request.user.has\_perm(

"%s.%s" % (opts.app\_label, codename\_view)

) or request.user.has\_perm("%s.%s" % (opts.app\_label, codename\_change))

def has\_view\_or\_change\_permission(self, request, obj=None):

return self.has\_view\_permission(request, obj) or self.has\_change\_permission(

request, obj

)

def has\_module\_permission(self, request):

"""

Return True if the given request has any permission in the given

app label.

Can be overridden by the user in subclasses. In such case it should

return True if the given request has permission to view the module on

the admin index page and access the module's index page. Overriding it

does not restrict access to the add, change or delete views. Use

`ModelAdmin.has\_(add|change|delete)\_permission` for that.

"""

return request.user.has\_module\_perms(self.opts.app\_label)

class ModelAdmin(BaseModelAdmin):

"""Encapsulate all admin options and functionality for a given model."""

list\_display = ("\_\_str\_\_",)

list\_display\_links = ()

list\_filter = ()

list\_select\_related = False

list\_per\_page = 100

list\_max\_show\_all = 200

list\_editable = ()

search\_fields = ()

search\_help\_text = None

date\_hierarchy = None

save\_as = False

save\_as\_continue = True

save\_on\_top = False

paginator = Paginator

preserve\_filters = True

show\_facets = ShowFacets.ALLOW

inlines = ()

# Custom templates (designed to be over-ridden in subclasses)

add\_form\_template = None

change\_form\_template = None

change\_list\_template = None

delete\_confirmation\_template = None

delete\_selected\_confirmation\_template = None

object\_history\_template = None

popup\_response\_template = None

# Actions

actions = ()

action\_form = helpers.ActionForm

actions\_on\_top = True

actions\_on\_bottom = False

actions\_selection\_counter = True

checks\_class = ModelAdminChecks

def \_\_init\_\_(self, model, admin\_site):

self.model = model

self.opts = model.\_meta

self.admin\_site = admin\_site

super().\_\_init\_\_()

def \_\_str\_\_(self):

return "%s.%s" % (self.opts.app\_label, self.\_\_class\_\_.\_\_name\_\_)

def \_\_repr\_\_(self):

return (

f"<{self.\_\_class\_\_.\_\_qualname\_\_}: model={self.model.\_\_qualname\_\_} "

f"site={self.admin\_site!r}>"

)

def get\_inline\_instances(self, request, obj=None):

inline\_instances = []

for inline\_class in self.get\_inlines(request, obj):

inline = inline\_class(self.model, self.admin\_site)

if request:

if not (

inline.has\_view\_or\_change\_permission(request, obj)

or inline.has\_add\_permission(request, obj)

or inline.has\_delete\_permission(request, obj)

):

continue

if not inline.has\_add\_permission(request, obj):

inline.max\_num = 0

inline\_instances.append(inline)

return inline\_instances

def get\_urls(self):

from django.urls import path

def wrap(view):

def wrapper(\*args, \*\*kwargs):

return self.admin\_site.admin\_view(view)(\*args, \*\*kwargs)

wrapper.model\_admin = self

return update\_wrapper(wrapper, view)

info = self.opts.app\_label, self.opts.model\_name

return [

path("", wrap(self.changelist\_view), name="%s\_%s\_changelist" % info),

path("add/", wrap(self.add\_view), name="%s\_%s\_add" % info),

path(

"<path:object\_id>/history/",

wrap(self.history\_view),

name="%s\_%s\_history" % info,

),

path(

"<path:object\_id>/delete/",

wrap(self.delete\_view),

name="%s\_%s\_delete" % info,

),

path(

"<path:object\_id>/change/",

wrap(self.change\_view),

name="%s\_%s\_change" % info,

),

# For backwards compatibility (was the change url before 1.9)

path(

"<path:object\_id>/",

wrap(

RedirectView.as\_view(

pattern\_name="%s:%s\_%s\_change"

% ((self.admin\_site.name,) + info)

)

),

),

]

@property

def urls(self):

return self.get\_urls()

@property

def media(self):

extra = "" if settings.DEBUG else ".min"

js = [

"vendor/jquery/jquery%s.js" % extra,

"jquery.init.js",

"core.js",

"admin/RelatedObjectLookups.js",

"actions.js",

"urlify.js",

"prepopulate.js",

"vendor/xregexp/xregexp%s.js" % extra,

]

return forms.Media(js=["admin/js/%s" % url for url in js])

def get\_model\_perms(self, request):

"""

Return a dict of all perms for this model. This dict has the keys

``add``, ``change``, ``delete``, and ``view`` mapping to the True/False

for each of those actions.

"""

return {

"add": self.has\_add\_permission(request),

"change": self.has\_change\_permission(request),

"delete": self.has\_delete\_permission(request),

"view": self.has\_view\_permission(request),

}

def \_get\_form\_for\_get\_fields(self, request, obj):

return self.get\_form(request, obj, fields=None)

def get\_form(self, request, obj=None, change=False, \*\*kwargs):

"""

Return a Form class for use in the admin add view. This is used by

add\_view and change\_view.

"""

if "fields" in kwargs:

fields = kwargs.pop("fields")

else:

fields = flatten\_fieldsets(self.get\_fieldsets(request, obj))

excluded = self.get\_exclude(request, obj)

exclude = [] if excluded is None else list(excluded)

readonly\_fields = self.get\_readonly\_fields(request, obj)

exclude.extend(readonly\_fields)

# Exclude all fields if it's a change form and the user doesn't have

# the change permission.

if (

change

and hasattr(request, "user")

and not self.has\_change\_permission(request, obj)

):

exclude.extend(fields)

if excluded is None and hasattr(self.form, "\_meta") and self.form.\_meta.exclude:

# Take the custom ModelForm's Meta.exclude into account only if the

# ModelAdmin doesn't define its own.

exclude.extend(self.form.\_meta.exclude)

# if exclude is an empty list we pass None to be consistent with the

# default on modelform\_factory

exclude = exclude or None

# Remove declared form fields which are in readonly\_fields.

new\_attrs = dict.fromkeys(

f for f in readonly\_fields if f in self.form.declared\_fields

)

form = type(self.form.\_\_name\_\_, (self.form,), new\_attrs)

defaults = {

"form": form,

"fields": fields,

"exclude": exclude,

"formfield\_callback": partial(self.formfield\_for\_dbfield, request=request),

\*\*kwargs,

}

if defaults["fields"] is None and not modelform\_defines\_fields(

defaults["form"]

):

defaults["fields"] = forms.ALL\_FIELDS

try:

return modelform\_factory(self.model, \*\*defaults)

except FieldError as e:

raise FieldError(

"%s. Check fields/fieldsets/exclude attributes of class %s."

% (e, self.\_\_class\_\_.\_\_name\_\_)

)

def get\_changelist(self, request, \*\*kwargs):

"""

Return the ChangeList class for use on the changelist page.

"""

from django.contrib.admin.views.main import ChangeList

return ChangeList

def get\_changelist\_instance(self, request):

"""

Return a `ChangeList` instance based on `request`. May raise

`IncorrectLookupParameters`.

"""

list\_display = self.get\_list\_display(request)

list\_display\_links = self.get\_list\_display\_links(request, list\_display)

# Add the action checkboxes if any actions are available.

if self.get\_actions(request):

list\_display = ["action\_checkbox", \*list\_display]

sortable\_by = self.get\_sortable\_by(request)

ChangeList = self.get\_changelist(request)

return ChangeList(

request,

self.model,

list\_display,

list\_display\_links,

self.get\_list\_filter(request),

self.date\_hierarchy,

self.get\_search\_fields(request),

self.get\_list\_select\_related(request),

self.list\_per\_page,

self.list\_max\_show\_all,

self.list\_editable,

self,

sortable\_by,

self.search\_help\_text,

)

def get\_object(self, request, object\_id, from\_field=None):

"""

Return an instance matching the field and value provided, the primary

key is used if no field is provided. Return ``None`` if no match is

found or the object\_id fails validation.

"""

queryset = self.get\_queryset(request)

model = queryset.model

field = (

model.\_meta.pk if from\_field is None else model.\_meta.get\_field(from\_field)

)

try:

object\_id = field.to\_python(object\_id)

return queryset.get(\*\*{field.name: object\_id})

except (model.DoesNotExist, ValidationError, ValueError):

return None

def get\_changelist\_form(self, request, \*\*kwargs):

"""

Return a Form class for use in the Formset on the changelist page.

"""

defaults = {

"formfield\_callback": partial(self.formfield\_for\_dbfield, request=request),

\*\*kwargs,

}

if defaults.get("fields") is None and not modelform\_defines\_fields(

defaults.get("form")

):

defaults["fields"] = forms.ALL\_FIELDS

return modelform\_factory(self.model, \*\*defaults)

def get\_changelist\_formset(self, request, \*\*kwargs):

"""

Return a FormSet class for use on the changelist page if list\_editable

is used.

"""

defaults = {

"formfield\_callback": partial(self.formfield\_for\_dbfield, request=request),

\*\*kwargs,

}

return modelformset\_factory(

self.model,

self.get\_changelist\_form(request),

extra=0,

fields=self.list\_editable,

\*\*defaults,

)

def get\_formsets\_with\_inlines(self, request, obj=None):

"""

Yield formsets and the corresponding inlines.

"""

for inline in self.get\_inline\_instances(request, obj):

yield inline.get\_formset(request, obj), inline

def get\_paginator(

self, request, queryset, per\_page, orphans=0, allow\_empty\_first\_page=True

):

return self.paginator(queryset, per\_page, orphans, allow\_empty\_first\_page)

def log\_addition(self, request, obj, message):

"""

Log that an object has been successfully added.

The default implementation creates an admin LogEntry object.

"""

from django.contrib.admin.models import ADDITION, LogEntry

return LogEntry.objects.log\_actions(

user\_id=request.user.pk,

queryset=[obj],

action\_flag=ADDITION,

change\_message=message,

single\_object=True,

)

def log\_change(self, request, obj, message):

"""

Log that an object has been successfully changed.

The default implementation creates an admin LogEntry object.

"""

from django.contrib.admin.models import CHANGE, LogEntry

return LogEntry.objects.log\_actions(

user\_id=request.user.pk,

queryset=[obj],

action\_flag=CHANGE,

change\_message=message,

single\_object=True,

)

def log\_deletion(self, request, obj, object\_repr):

"""

Log that an object will be deleted. Note that this method must be

called before the deletion.

The default implementation creates an admin LogEntry object.

"""

warnings.warn(

"ModelAdmin.log\_deletion() is deprecated. Use log\_deletions() instead.",

RemovedInDjango60Warning,

stacklevel=2,

)

from django.contrib.admin.models import DELETION, LogEntry

return LogEntry.objects.log\_action(

user\_id=request.user.pk,

content\_type\_id=get\_content\_type\_for\_model(obj).pk,

object\_id=obj.pk,

object\_repr=object\_repr,

action\_flag=DELETION,

)

def log\_deletions(self, request, queryset):

"""

Log that objects will be deleted. Note that this method must be called

before the deletion.

The default implementation creates admin LogEntry objects.

"""

from django.contrib.admin.models import DELETION, LogEntry

# RemovedInDjango60Warning.

if type(self).log\_deletion != ModelAdmin.log\_deletion:

warnings.warn(

"The usage of log\_deletion() is deprecated. Implement log\_deletions() "

"instead.",

RemovedInDjango60Warning,

stacklevel=2,

)

return [self.log\_deletion(request, obj, str(obj)) for obj in queryset]

return LogEntry.objects.log\_actions(

user\_id=request.user.pk,

queryset=queryset,

action\_flag=DELETION,

)

def action\_checkbox(self, obj):

"""

A list\_display column containing a checkbox widget.

"""

attrs = {

"class": "action-select",

"aria-label": format\_html(

\_("Select this object for an action - {}"), str(obj)

),

}

checkbox = forms.CheckboxInput(attrs, lambda value: False)

return checkbox.render(helpers.ACTION\_CHECKBOX\_NAME, str(obj.pk))

@staticmethod

def \_get\_action\_description(func, name):

try:

return func.short\_description

except AttributeError:

return capfirst(name.replace("\_", " "))

def \_get\_base\_actions(self):

"""Return the list of actions, prior to any request-based filtering."""

actions = []

base\_actions = (self.get\_action(action) for action in self.actions or [])

# get\_action might have returned None, so filter any of those out.

base\_actions = [action for action in base\_actions if action]

base\_action\_names = {name for \_, name, \_ in base\_actions}

# Gather actions from the admin site first

for name, func in self.admin\_site.actions:

if name in base\_action\_names:

continue

description = self.\_get\_action\_description(func, name)

actions.append((func, name, description))

# Add actions from this ModelAdmin.

actions.extend(base\_actions)

return actions

def \_filter\_actions\_by\_permissions(self, request, actions):

"""Filter out any actions that the user doesn't have access to."""

filtered\_actions = []

for action in actions:

callable = action[0]

if not hasattr(callable, "allowed\_permissions"):

filtered\_actions.append(action)

continue

permission\_checks = (

getattr(self, "has\_%s\_permission" % permission)

for permission in callable.allowed\_permissions

)

if any(has\_permission(request) for has\_permission in permission\_checks):

filtered\_actions.append(action)

return filtered\_actions

def get\_actions(self, request):

"""

Return a dictionary mapping the names of all actions for this

ModelAdmin to a tuple of (callable, name, description) for each action.

"""

# If self.actions is set to None that means actions are disabled on

# this page.

if self.actions is None or IS\_POPUP\_VAR in request.GET:

return {}

actions = self.\_filter\_actions\_by\_permissions(request, self.\_get\_base\_actions())

return {name: (func, name, desc) for func, name, desc in actions}

def get\_action\_choices(self, request, default\_choices=models.BLANK\_CHOICE\_DASH):

"""

Return a list of choices for use in a form object. Each choice is a

tuple (name, description).

"""

choices = [] + default\_choices

for func, name, description in self.get\_actions(request).values():

choice = (name, description % model\_format\_dict(self.opts))

choices.append(choice)

return choices

def get\_action(self, action):

"""

Return a given action from a parameter, which can either be a callable,

or the name of a method on the ModelAdmin. Return is a tuple of

(callable, name, description).

"""

# If the action is a callable, just use it.

if callable(action):

func = action

action = action.\_\_name\_\_

# Next, look for a method. Grab it off self.\_\_class\_\_ to get an unbound

# method instead of a bound one; this ensures that the calling

# conventions are the same for functions and methods.

elif hasattr(self.\_\_class\_\_, action):

func = getattr(self.\_\_class\_\_, action)

# Finally, look for a named method on the admin site

else:

try:

func = self.admin\_site.get\_action(action)

except KeyError:

return None

description = self.\_get\_action\_description(func, action)

return func, action, description

def get\_list\_display(self, request):

"""

Return a sequence containing the fields to be displayed on the

changelist.

"""

return self.list\_display

def get\_list\_display\_links(self, request, list\_display):

"""

Return a sequence containing the fields to be displayed as links

on the changelist. The list\_display parameter is the list of fields

returned by get\_list\_display().

"""

if (

self.list\_display\_links

or self.list\_display\_links is None

or not list\_display

):

return self.list\_display\_links

else:

# Use only the first item in list\_display as link

return list(list\_display)[:1]

def get\_list\_filter(self, request):

"""

Return a sequence containing the fields to be displayed as filters in

the right sidebar of the changelist page.

"""

return self.list\_filter

def get\_list\_select\_related(self, request):

"""

Return a list of fields to add to the select\_related() part of the

changelist items query.

"""

return self.list\_select\_related

def get\_search\_fields(self, request):

"""

Return a sequence containing the fields to be searched whenever

somebody submits a search query.

"""

return self.search\_fields

def get\_search\_results(self, request, queryset, search\_term):

"""

Return a tuple containing a queryset to implement the search

and a boolean indicating if the results may contain duplicates.

"""

# Apply keyword searches.

def construct\_search(field\_name):

if field\_name.startswith("^"):

return "%s\_\_istartswith" % field\_name.removeprefix("^")

elif field\_name.startswith("="):

return "%s\_\_iexact" % field\_name.removeprefix("=")

elif field\_name.startswith("@"):

return "%s\_\_search" % field\_name.removeprefix("@")

# Use field\_name if it includes a lookup.

opts = queryset.model.\_meta

lookup\_fields = field\_name.split(LOOKUP\_SEP)

# Go through the fields, following all relations.

prev\_field = None

for path\_part in lookup\_fields:

if path\_part == "pk":

path\_part = opts.pk.name

try:

field = opts.get\_field(path\_part)

except FieldDoesNotExist:

# Use valid query lookups.

if prev\_field and prev\_field.get\_lookup(path\_part):

return field\_name

else:

prev\_field = field

if hasattr(field, "path\_infos"):

# Update opts to follow the relation.

opts = field.path\_infos[-1].to\_opts

# Otherwise, use the field with icontains.

return "%s\_\_icontains" % field\_name

may\_have\_duplicates = False

search\_fields = self.get\_search\_fields(request)

if search\_fields and search\_term:

orm\_lookups = [

construct\_search(str(search\_field)) for search\_field in search\_fields

]

term\_queries = []

for bit in smart\_split(search\_term):

if bit.startswith(('"', "'")) and bit[0] == bit[-1]:

bit = unescape\_string\_literal(bit)

or\_queries = models.Q.create(

[(orm\_lookup, bit) for orm\_lookup in orm\_lookups],

connector=models.Q.OR,

)

term\_queries.append(or\_queries)

queryset = queryset.filter(models.Q.create(term\_queries))

may\_have\_duplicates |= any(

lookup\_spawns\_duplicates(self.opts, search\_spec)

for search\_spec in orm\_lookups

)

return queryset, may\_have\_duplicates

def get\_preserved\_filters(self, request):

"""

Return the preserved filters querystring.

"""

match = request.resolver\_match

if self.preserve\_filters and match:

current\_url = "%s:%s" % (match.app\_name, match.url\_name)

changelist\_url = "admin:%s\_%s\_changelist" % (

self.opts.app\_label,

self.opts.model\_name,

)

if current\_url == changelist\_url:

preserved\_filters = request.GET.urlencode()

else:

preserved\_filters = request.GET.get("\_changelist\_filters")

if preserved\_filters:

return urlencode({"\_changelist\_filters": preserved\_filters})

return ""

def construct\_change\_message(self, request, form, formsets, add=False):

"""

Construct a JSON structure describing changes from a changed object.

"""

return construct\_change\_message(form, formsets, add)

def message\_user(

self, request, message, level=messages.INFO, extra\_tags="", fail\_silently=False

):

"""

Send a message to the user. The default implementation

posts a message using the django.contrib.messages backend.

Exposes almost the same API as messages.add\_message(), but accepts the

positional arguments in a different order to maintain backwards

compatibility. For convenience, it accepts the `level` argument as

a string rather than the usual level number.

"""

if not isinstance(level, int):

# attempt to get the level if passed a string

try:

level = getattr(messages.constants, level.upper())

except AttributeError:

levels = messages.constants.DEFAULT\_TAGS.values()

levels\_repr = ", ".join("`%s`" % level for level in levels)

raise ValueError(

"Bad message level string: `%s`. Possible values are: %s"

% (level, levels\_repr)

)

messages.add\_message(

request, level, message, extra\_tags=extra\_tags, fail\_silently=fail\_silently

)

def save\_form(self, request, form, change):

"""

Given a ModelForm return an unsaved instance. ``change`` is True if

the object is being changed, and False if it's being added.

"""

return form.save(commit=False)

def save\_model(self, request, obj, form, change):

"""

Given a model instance save it to the database.

"""

obj.save()

def delete\_model(self, request, obj):

"""

Given a model instance delete it from the database.

"""

obj.delete()

def delete\_queryset(self, request, queryset):

"""Given a queryset, delete it from the database."""

queryset.delete()

def save\_formset(self, request, form, formset, change):

"""

Given an inline formset save it to the database.

"""

formset.save()

def save\_related(self, request, form, formsets, change):

"""

Given the ``HttpRequest``, the parent ``ModelForm`` instance, the

list of inline formsets and a boolean value based on whether the

parent is being added or changed, save the related objects to the

database. Note that at this point save\_form() and save\_model() have

already been called.

"""

form.save\_m2m()

for formset in formsets:

self.save\_formset(request, form, formset, change=change)

def render\_change\_form(

self, request, context, add=False, change=False, form\_url="", obj=None

):

app\_label = self.opts.app\_label

preserved\_filters = self.get\_preserved\_filters(request)

form\_url = add\_preserved\_filters(

{"preserved\_filters": preserved\_filters, "opts": self.opts}, form\_url

)

view\_on\_site\_url = self.get\_view\_on\_site\_url(obj)

has\_editable\_inline\_admin\_formsets = False

for inline in context["inline\_admin\_formsets"]:

if (

inline.has\_add\_permission

or inline.has\_change\_permission

or inline.has\_delete\_permission

):

has\_editable\_inline\_admin\_formsets = True

break

context.update(

{

"add": add,

"change": change,

"has\_view\_permission": self.has\_view\_permission(request, obj),

"has\_add\_permission": self.has\_add\_permission(request),

"has\_change\_permission": self.has\_change\_permission(request, obj),

"has\_delete\_permission": self.has\_delete\_permission(request, obj),

"has\_editable\_inline\_admin\_formsets": (

has\_editable\_inline\_admin\_formsets

),

"has\_file\_field": context["adminform"].form.is\_multipart()

or any(

admin\_formset.formset.is\_multipart()

for admin\_formset in context["inline\_admin\_formsets"]

),

"has\_absolute\_url": view\_on\_site\_url is not None,

"absolute\_url": view\_on\_site\_url,

"form\_url": form\_url,

"opts": self.opts,

"content\_type\_id": get\_content\_type\_for\_model(self.model).pk,

"save\_as": self.save\_as,

"save\_on\_top": self.save\_on\_top,

"to\_field\_var": TO\_FIELD\_VAR,

"is\_popup\_var": IS\_POPUP\_VAR,

"app\_label": app\_label,

}

)

if add and self.add\_form\_template is not None:

form\_template = self.add\_form\_template

else:

form\_template = self.change\_form\_template

request.current\_app = self.admin\_site.name

return TemplateResponse(

request,

form\_template

or [

"admin/%s/%s/change\_form.html" % (app\_label, self.opts.model\_name),

"admin/%s/change\_form.html" % app\_label,

"admin/change\_form.html",

],

context,

)

def \_get\_preserved\_qsl(self, request, preserved\_filters):

query\_string = urlparse(request.build\_absolute\_uri()).query

return parse\_qsl(query\_string.replace(preserved\_filters, ""))

def response\_add(self, request, obj, post\_url\_continue=None):

"""

Determine the HttpResponse for the add\_view stage.

"""

opts = obj.\_meta

preserved\_filters = self.get\_preserved\_filters(request)

preserved\_qsl = self.\_get\_preserved\_qsl(request, preserved\_filters)

obj\_url = reverse(

"admin:%s\_%s\_change" % (opts.app\_label, opts.model\_name),

args=(quote(obj.pk),),

current\_app=self.admin\_site.name,

)

# Add a link to the object's change form if the user can edit the obj.

if self.has\_change\_permission(request, obj):

obj\_repr = format\_html('<a href="{}">{}</a>', urlquote(obj\_url), obj)

else:

obj\_repr = str(obj)

msg\_dict = {

"name": opts.verbose\_name,

"obj": obj\_repr,

}

# Here, we distinguish between different save types by checking for

# the presence of keys in request.POST.

if IS\_POPUP\_VAR in request.POST:

to\_field = request.POST.get(TO\_FIELD\_VAR)

if to\_field:

attr = str(to\_field)

else:

attr = obj.\_meta.pk.attname

value = obj.serializable\_value(attr)

popup\_response\_data = json.dumps(

{

"value": str(value),

"obj": str(obj),

}

)

return TemplateResponse(

request,

self.popup\_response\_template

or [

"admin/%s/%s/popup\_response.html"

% (opts.app\_label, opts.model\_name),

"admin/%s/popup\_response.html" % opts.app\_label,

"admin/popup\_response.html",

],

{

"popup\_response\_data": popup\_response\_data,

},

)

elif "\_continue" in request.POST or (

# Redirecting after "Save as new".

"\_saveasnew" in request.POST

and self.save\_as\_continue

and self.has\_change\_permission(request, obj)

):

msg = \_("The {name} ?obj}??was added successfully.")

if self.has\_change\_permission(request, obj):

msg += " " + \_("You may edit it again below.")

self.message\_user(request, format\_html(msg, \*\*msg\_dict), messages.SUCCESS)

if post\_url\_continue is None:

post\_url\_continue = obj\_url

post\_url\_continue = add\_preserved\_filters(

{

"preserved\_filters": preserved\_filters,

"preserved\_qsl": preserved\_qsl,

"opts": opts,

},

post\_url\_continue,

)

return HttpResponseRedirect(post\_url\_continue)

elif "\_addanother" in request.POST:

msg = format\_html(

\_(

"The {name} ?obj}??was added successfully. You may add another "

"{name} below."

),

\*\*msg\_dict,

)

self.message\_user(request, msg, messages.SUCCESS)

redirect\_url = request.path

redirect\_url = add\_preserved\_filters(

{

"preserved\_filters": preserved\_filters,

"preserved\_qsl": preserved\_qsl,

"opts": opts,

},

redirect\_url,

)

return HttpResponseRedirect(redirect\_url)

else:

msg = format\_html(

\_("The {name} ?obj}??was added successfully."), \*\*msg\_dict

)

self.message\_user(request, msg, messages.SUCCESS)

return self.response\_post\_save\_add(request, obj)

def response\_change(self, request, obj):

"""

Determine the HttpResponse for the change\_view stage.

"""

if IS\_POPUP\_VAR in request.POST:

opts = obj.\_meta

to\_field = request.POST.get(TO\_FIELD\_VAR)

attr = str(to\_field) if to\_field else opts.pk.attname

value = request.resolver\_match.kwargs["object\_id"]

new\_value = obj.serializable\_value(attr)

popup\_response\_data = json.dumps(

{

"action": "change",

"value": str(value),

"obj": str(obj),

"new\_value": str(new\_value),

}

)

return TemplateResponse(

request,

self.popup\_response\_template

or [

"admin/%s/%s/popup\_response.html"

% (opts.app\_label, opts.model\_name),

"admin/%s/popup\_response.html" % opts.app\_label,

"admin/popup\_response.html",

],

{

"popup\_response\_data": popup\_response\_data,

},

)

opts = self.opts

preserved\_filters = self.get\_preserved\_filters(request)

preserved\_qsl = self.\_get\_preserved\_qsl(request, preserved\_filters)

msg\_dict = {

"name": opts.verbose\_name,

"obj": format\_html('<a href="{}">{}</a>', urlquote(request.path), obj),

}

if "\_continue" in request.POST:

msg = format\_html(

\_(

"The {name} ?obj}??was changed successfully. You may edit it "

"again below."

),

\*\*msg\_dict,

)

self.message\_user(request, msg, messages.SUCCESS)

redirect\_url = request.path

redirect\_url = add\_preserved\_filters(

{

"preserved\_filters": preserved\_filters,

"preserved\_qsl": preserved\_qsl,

"opts": opts,

},

redirect\_url,

)

return HttpResponseRedirect(redirect\_url)

elif "\_addanother" in request.POST:

msg = format\_html(

\_(

"The {name} ?obj}??was changed successfully. You may add another "

"{name} below."

),

\*\*msg\_dict,

)

self.message\_user(request, msg, messages.SUCCESS)

redirect\_url = reverse(

"admin:%s\_%s\_add" % (opts.app\_label, opts.model\_name),

current\_app=self.admin\_site.name,

)

redirect\_url = add\_preserved\_filters(

{

"preserved\_filters": preserved\_filters,

"preserved\_qsl": preserved\_qsl,

"opts": opts,

},

redirect\_url,

)

return HttpResponseRedirect(redirect\_url)

else:

msg = format\_html(

\_("The {name} ?obj}??was changed successfully."), \*\*msg\_dict

)

self.message\_user(request, msg, messages.SUCCESS)

return self.response\_post\_save\_change(request, obj)

def \_response\_post\_save(self, request, obj):

if self.has\_view\_or\_change\_permission(request):

post\_url = reverse(

"admin:%s\_%s\_changelist" % (self.opts.app\_label, self.opts.model\_name),

current\_app=self.admin\_site.name,

)

preserved\_filters = self.get\_preserved\_filters(request)

post\_url = add\_preserved\_filters(

{"preserved\_filters": preserved\_filters, "opts": self.opts}, post\_url

)

else:

post\_url = reverse("admin:index", current\_app=self.admin\_site.name)

return HttpResponseRedirect(post\_url)

def response\_post\_save\_add(self, request, obj):

"""

Figure out where to redirect after the 'Save' button has been pressed

when adding a new object.

"""

return self.\_response\_post\_save(request, obj)

def response\_post\_save\_change(self, request, obj):

"""

Figure out where to redirect after the 'Save' button has been pressed

when editing an existing object.

"""

return self.\_response\_post\_save(request, obj)

def response\_action(self, request, queryset):

"""

Handle an admin action. This is called if a request is POSTed to the

changelist; it returns an HttpResponse if the action was handled, and

None otherwise.

"""

# There can be multiple action forms on the page (at the top

# and bottom of the change list, for example). Get the action

# whose button was pushed.

try:

action\_index = int(request.POST.get("index", 0))

except ValueError:

action\_index = 0

# Construct the action form.

data = request.POST.copy()

data.pop(helpers.ACTION\_CHECKBOX\_NAME, None)

data.pop("index", None)

# Use the action whose button was pushed

try:

data.update({"action": data.getlist("action")[action\_index]})

except IndexError:

# If we didn't get an action from the chosen form that's invalid

# POST data, so by deleting action it'll fail the validation check

# below. So no need to do anything here

pass

action\_form = self.action\_form(data, auto\_id=None)

action\_form.fields["action"].choices = self.get\_action\_choices(request)

# If the form's valid we can handle the action.

if action\_form.is\_valid():

action = action\_form.cleaned\_data["action"]

select\_across = action\_form.cleaned\_data["select\_across"]

func = self.get\_actions(request)[action][0]

# Get the list of selected PKs. If nothing's selected, we can't

# perform an action on it, so bail. Except we want to perform

# the action explicitly on all objects.

selected = request.POST.getlist(helpers.ACTION\_CHECKBOX\_NAME)

if not selected and not select\_across:

# Reminder that something needs to be selected or nothing will happen

msg = \_(

"Items must be selected in order to perform "

"actions on them. No items have been changed."

)

self.message\_user(request, msg, messages.WARNING)

return None

if not select\_across:

# Perform the action only on the selected objects

queryset = queryset.filter(pk\_\_in=selected)

response = func(self, request, queryset)

# Actions may return an HttpResponse-like object, which will be

# used as the response from the POST. If not, we'll be a good

# little HTTP citizen and redirect back to the changelist page.

if isinstance(response, HttpResponseBase):

return response

else:

return HttpResponseRedirect(request.get\_full\_path())

else:

msg = \_("No action selected.")

self.message\_user(request, msg, messages.WARNING)

return None

def response\_delete(self, request, obj\_display, obj\_id):

"""

Determine the HttpResponse for the delete\_view stage.

"""

if IS\_POPUP\_VAR in request.POST:

popup\_response\_data = json.dumps(

{

"action": "delete",

"value": str(obj\_id),

}

)

return TemplateResponse(

request,

self.popup\_response\_template

or [

"admin/%s/%s/popup\_response.html"

% (self.opts.app\_label, self.opts.model\_name),

"admin/%s/popup\_response.html" % self.opts.app\_label,

"admin/popup\_response.html",

],

{

"popup\_response\_data": popup\_response\_data,

},

)

self.message\_user(

request,

\_("The %(name)s ??(obj)s??was deleted successfully.")

% {

"name": self.opts.verbose\_name,

"obj": obj\_display,

},

messages.SUCCESS,

)

if self.has\_change\_permission(request, None):

post\_url = reverse(

"admin:%s\_%s\_changelist" % (self.opts.app\_label, self.opts.model\_name),

current\_app=self.admin\_site.name,

)

preserved\_filters = self.get\_preserved\_filters(request)

post\_url = add\_preserved\_filters(

{"preserved\_filters": preserved\_filters, "opts": self.opts}, post\_url

)

else:

post\_url = reverse("admin:index", current\_app=self.admin\_site.name)

return HttpResponseRedirect(post\_url)

def render\_delete\_form(self, request, context):

app\_label = self.opts.app\_label

request.current\_app = self.admin\_site.name

context.update(

to\_field\_var=TO\_FIELD\_VAR,

is\_popup\_var=IS\_POPUP\_VAR,

media=self.media,

)

return TemplateResponse(

request,

self.delete\_confirmation\_template

or [

"admin/{}/{}/delete\_confirmation.html".format(

app\_label, self.opts.model\_name

),

"admin/{}/delete\_confirmation.html".format(app\_label),

"admin/delete\_confirmation.html",

],

context,

)

def get\_inline\_formsets(self, request, formsets, inline\_instances, obj=None):

# Edit permissions on parent model are required for editable inlines.

can\_edit\_parent = (

self.has\_change\_permission(request, obj)

if obj

else self.has\_add\_permission(request)

)

inline\_admin\_formsets = []

for inline, formset in zip(inline\_instances, formsets):

fieldsets = list(inline.get\_fieldsets(request, obj))

readonly = list(inline.get\_readonly\_fields(request, obj))

if can\_edit\_parent:

has\_add\_permission = inline.has\_add\_permission(request, obj)

has\_change\_permission = inline.has\_change\_permission(request, obj)

has\_delete\_permission = inline.has\_delete\_permission(request, obj)

else:

# Disable all edit-permissions, and override formset settings.

has\_add\_permission = has\_change\_permission = has\_delete\_permission = (

False

)

formset.extra = formset.max\_num = 0

has\_view\_permission = inline.has\_view\_permission(request, obj)

prepopulated = dict(inline.get\_prepopulated\_fields(request, obj))

inline\_admin\_formset = helpers.InlineAdminFormSet(

inline,

formset,

fieldsets,

prepopulated,

readonly,

model\_admin=self,

has\_add\_permission=has\_add\_permission,

has\_change\_permission=has\_change\_permission,

has\_delete\_permission=has\_delete\_permission,

has\_view\_permission=has\_view\_permission,

)

inline\_admin\_formsets.append(inline\_admin\_formset)

return inline\_admin\_formsets

def get\_changeform\_initial\_data(self, request):

"""

Get the initial form data from the request's GET params.

"""

initial = dict(request.GET.items())

for k in initial:

try:

f = self.opts.get\_field(k)

except FieldDoesNotExist:

continue

# We have to special-case M2Ms as a list of comma-separated PKs.

if isinstance(f, models.ManyToManyField):

initial[k] = initial[k].split(",")

return initial

def \_get\_obj\_does\_not\_exist\_redirect(self, request, opts, object\_id):

"""

Create a message informing the user that the object doesn't exist

and return a redirect to the admin index page.

"""

msg = \_("%(name)s with ID ??(key)s??doesn? exist. Perhaps it was deleted?") % {

"name": opts.verbose\_name,

"key": unquote(object\_id),

}

self.message\_user(request, msg, messages.WARNING)

url = reverse("admin:index", current\_app=self.admin\_site.name)

return HttpResponseRedirect(url)

@csrf\_protect\_m

def changeform\_view(self, request, object\_id=None, form\_url="", extra\_context=None):

with transaction.atomic(using=router.db\_for\_write(self.model)):

return self.\_changeform\_view(request, object\_id, form\_url, extra\_context)

def \_changeform\_view(self, request, object\_id, form\_url, extra\_context):

to\_field = request.POST.get(TO\_FIELD\_VAR, request.GET.get(TO\_FIELD\_VAR))

if to\_field and not self.to\_field\_allowed(request, to\_field):

raise DisallowedModelAdminToField(

"The field %s cannot be referenced." % to\_field

)

if request.method == "POST" and "\_saveasnew" in request.POST:

object\_id = None

add = object\_id is None

if add:

if not self.has\_add\_permission(request):

raise PermissionDenied

obj = None

else:

obj = self.get\_object(request, unquote(object\_id), to\_field)

if request.method == "POST":

if not self.has\_change\_permission(request, obj):

raise PermissionDenied

else:

if not self.has\_view\_or\_change\_permission(request, obj):

raise PermissionDenied

if obj is None:

return self.\_get\_obj\_does\_not\_exist\_redirect(

request, self.opts, object\_id

)

fieldsets = self.get\_fieldsets(request, obj)

ModelForm = self.get\_form(

request, obj, change=not add, fields=flatten\_fieldsets(fieldsets)

)

if request.method == "POST":

form = ModelForm(request.POST, request.FILES, instance=obj)

formsets, inline\_instances = self.\_create\_formsets(

request,

form.instance,

change=not add,

)

form\_validated = form.is\_valid()

if form\_validated:

new\_object = self.save\_form(request, form, change=not add)

else:

new\_object = form.instance

if all\_valid(formsets) and form\_validated:

self.save\_model(request, new\_object, form, not add)

self.save\_related(request, form, formsets, not add)

change\_message = self.construct\_change\_message(

request, form, formsets, add

)

if add:

self.log\_addition(request, new\_object, change\_message)

return self.response\_add(request, new\_object)

else:

self.log\_change(request, new\_object, change\_message)

return self.response\_change(request, new\_object)

else:

form\_validated = False

else:

if add:

initial = self.get\_changeform\_initial\_data(request)

form = ModelForm(initial=initial)

formsets, inline\_instances = self.\_create\_formsets(

request, form.instance, change=False

)

else:

form = ModelForm(instance=obj)

formsets, inline\_instances = self.\_create\_formsets(

request, obj, change=True

)

if not add and not self.has\_change\_permission(request, obj):

readonly\_fields = flatten\_fieldsets(fieldsets)

else:

readonly\_fields = self.get\_readonly\_fields(request, obj)

admin\_form = helpers.AdminForm(

form,

list(fieldsets),

# Clear prepopulated fields on a view-only form to avoid a crash.

(

self.get\_prepopulated\_fields(request, obj)

if add or self.has\_change\_permission(request, obj)

else {}

),

readonly\_fields,

model\_admin=self,

)

media = self.media + admin\_form.media

inline\_formsets = self.get\_inline\_formsets(

request, formsets, inline\_instances, obj

)

for inline\_formset in inline\_formsets:

media += inline\_formset.media

if add:

title = \_("Add %s")

elif self.has\_change\_permission(request, obj):

title = \_("Change %s")

else:

title = \_("View %s")

context = {

\*\*self.admin\_site.each\_context(request),

"title": title % self.opts.verbose\_name,

"subtitle": str(obj) if obj else None,

"adminform": admin\_form,

"object\_id": object\_id,

"original": obj,

"is\_popup": IS\_POPUP\_VAR in request.POST or IS\_POPUP\_VAR in request.GET,

"to\_field": to\_field,

"media": media,

"inline\_admin\_formsets": inline\_formsets,

"errors": helpers.AdminErrorList(form, formsets),

"preserved\_filters": self.get\_preserved\_filters(request),

}

# Hide the "Save" and "Save and continue" buttons if "Save as New" was

# previously chosen to prevent the interface from getting confusing.

if (

request.method == "POST"

and not form\_validated

and "\_saveasnew" in request.POST

):

context["show\_save"] = False

context["show\_save\_and\_continue"] = False

# Use the change template instead of the add template.

add = False

context.update(extra\_context or {})

return self.render\_change\_form(

request, context, add=add, change=not add, obj=obj, form\_url=form\_url

)

def add\_view(self, request, form\_url="", extra\_context=None):

return self.changeform\_view(request, None, form\_url, extra\_context)

def change\_view(self, request, object\_id, form\_url="", extra\_context=None):

return self.changeform\_view(request, object\_id, form\_url, extra\_context)

def \_get\_edited\_object\_pks(self, request, prefix):

"""Return POST data values of list\_editable primary keys."""

pk\_pattern = re.compile(

r"{}-\d+-{}$".format(re.escape(prefix), self.opts.pk.name)

)

return [value for key, value in request.POST.items() if pk\_pattern.match(key)]

def \_get\_list\_editable\_queryset(self, request, prefix):

"""

Based on POST data, return a queryset of the objects that were edited

via list\_editable.

"""

object\_pks = self.\_get\_edited\_object\_pks(request, prefix)

queryset = self.get\_queryset(request)

validate = queryset.model.\_meta.pk.to\_python

try:

for pk in object\_pks:

validate(pk)

except ValidationError:

# Disable the optimization if the POST data was tampered with.

return queryset

return queryset.filter(pk\_\_in=object\_pks)

@csrf\_protect\_m

def changelist\_view(self, request, extra\_context=None):

"""

The 'change list' admin view for this model.

"""

from django.contrib.admin.views.main import ERROR\_FLAG

app\_label = self.opts.app\_label

if not self.has\_view\_or\_change\_permission(request):

raise PermissionDenied

try:

cl = self.get\_changelist\_instance(request)

except IncorrectLookupParameters:

# Wacky lookup parameters were given, so redirect to the main

# changelist page, without parameters, and pass an 'invalid=1'

# parameter via the query string. If wacky parameters were given

# and the 'invalid=1' parameter was already in the query string,

# something is screwed up with the database, so display an error

# page.

if ERROR\_FLAG in request.GET:

return SimpleTemplateResponse(

"admin/invalid\_setup.html",

{

"title": \_("Database error"),

},

)

return HttpResponseRedirect(request.path + "?" + ERROR\_FLAG + "=1")

# If the request was POSTed, this might be a bulk action or a bulk

# edit. Try to look up an action or confirmation first, but if this

# isn't an action the POST will fall through to the bulk edit check,

# below.

action\_failed = False

selected = request.POST.getlist(helpers.ACTION\_CHECKBOX\_NAME)

actions = self.get\_actions(request)

# Actions with no confirmation

if (

actions

and request.method == "POST"

and "index" in request.POST

and "\_save" not in request.POST

):

if selected:

response = self.response\_action(

request, queryset=cl.get\_queryset(request)

)

if response:

return response

else:

action\_failed = True

else:

msg = \_(

"Items must be selected in order to perform "

"actions on them. No items have been changed."

)

self.message\_user(request, msg, messages.WARNING)

action\_failed = True

# Actions with confirmation

if (

actions

and request.method == "POST"

and helpers.ACTION\_CHECKBOX\_NAME in request.POST

and "index" not in request.POST

and "\_save" not in request.POST

):

if selected:

response = self.response\_action(

request, queryset=cl.get\_queryset(request)

)

if response:

return response

else:

action\_failed = True

if action\_failed:

# Redirect back to the changelist page to avoid resubmitting the

# form if the user refreshes the browser or uses the "No, take

# me back" button on the action confirmation page.

return HttpResponseRedirect(request.get\_full\_path())

# If we're allowing changelist editing, we need to construct a formset

# for the changelist given all the fields to be edited. Then we'll

# use the formset to validate/process POSTed data.

formset = cl.formset = None

# Handle POSTed bulk-edit data.

if request.method == "POST" and cl.list\_editable and "\_save" in request.POST:

if not self.has\_change\_permission(request):

raise PermissionDenied

FormSet = self.get\_changelist\_formset(request)

modified\_objects = self.\_get\_list\_editable\_queryset(

request, FormSet.get\_default\_prefix()

)

formset = cl.formset = FormSet(

request.POST, request.FILES, queryset=modified\_objects

)

if formset.is\_valid():

changecount = 0

with transaction.atomic(using=router.db\_for\_write(self.model)):

for form in formset.forms:

if form.has\_changed():

obj = self.save\_form(request, form, change=True)

self.save\_model(request, obj, form, change=True)

self.save\_related(request, form, formsets=[], change=True)

change\_msg = self.construct\_change\_message(

request, form, None

)

self.log\_change(request, obj, change\_msg)

changecount += 1

if changecount:

msg = ngettext(

"%(count)s %(name)s was changed successfully.",

"%(count)s %(name)s were changed successfully.",

changecount,

) % {

"count": changecount,

"name": model\_ngettext(self.opts, changecount),

}

self.message\_user(request, msg, messages.SUCCESS)

return HttpResponseRedirect(request.get\_full\_path())

# Handle GET -- construct a formset for display.

elif cl.list\_editable and self.has\_change\_permission(request):

FormSet = self.get\_changelist\_formset(request)

formset = cl.formset = FormSet(queryset=cl.result\_list)

# Build the list of media to be used by the formset.

if formset:

media = self.media + formset.media

else:

media = self.media

# Build the action form and populate it with available actions.

if actions:

action\_form = self.action\_form(auto\_id=None)

action\_form.fields["action"].choices = self.get\_action\_choices(request)

media += action\_form.media

else:

action\_form = None

selection\_note\_all = ngettext(

"%(total\_count)s selected", "All %(total\_count)s selected", cl.result\_count

)

context = {

\*\*self.admin\_site.each\_context(request),

"module\_name": str(self.opts.verbose\_name\_plural),

"selection\_note": \_("0 of %(cnt)s selected") % {"cnt": len(cl.result\_list)},

"selection\_note\_all": selection\_note\_all % {"total\_count": cl.result\_count},

"title": cl.title,

"subtitle": None,

"is\_popup": cl.is\_popup,

"to\_field": cl.to\_field,

"cl": cl,

"media": media,

"has\_add\_permission": self.has\_add\_permission(request),

"opts": cl.opts,

"action\_form": action\_form,

"actions\_on\_top": self.actions\_on\_top,

"actions\_on\_bottom": self.actions\_on\_bottom,

"actions\_selection\_counter": self.actions\_selection\_counter,

"preserved\_filters": self.get\_preserved\_filters(request),

\*\*(extra\_context or {}),

}

request.current\_app = self.admin\_site.name

return TemplateResponse(

request,

self.change\_list\_template

or [

"admin/%s/%s/change\_list.html" % (app\_label, self.opts.model\_name),

"admin/%s/change\_list.html" % app\_label,

"admin/change\_list.html",

],

context,

)

def get\_deleted\_objects(self, objs, request):

"""

Hook for customizing the delete process for the delete view and the

"delete selected" action.

"""

return get\_deleted\_objects(objs, request, self.admin\_site)

@csrf\_protect\_m

def delete\_view(self, request, object\_id, extra\_context=None):

with transaction.atomic(using=router.db\_for\_write(self.model)):

return self.\_delete\_view(request, object\_id, extra\_context)

def \_delete\_view(self, request, object\_id, extra\_context):

"The 'delete' admin view for this model."

app\_label = self.opts.app\_label

to\_field = request.POST.get(TO\_FIELD\_VAR, request.GET.get(TO\_FIELD\_VAR))

if to\_field and not self.to\_field\_allowed(request, to\_field):

raise DisallowedModelAdminToField(

"The field %s cannot be referenced." % to\_field

)

obj = self.get\_object(request, unquote(object\_id), to\_field)

if not self.has\_delete\_permission(request, obj):

raise PermissionDenied

if obj is None:

return self.\_get\_obj\_does\_not\_exist\_redirect(request, self.opts, object\_id)

# Populate deleted\_objects, a data structure of all related objects that

# will also be deleted.

(

deleted\_objects,

model\_count,

perms\_needed,

protected,

) = self.get\_deleted\_objects([obj], request)

if request.POST and not protected: # The user has confirmed the deletion.

if perms\_needed:

raise PermissionDenied

obj\_display = str(obj)

attr = str(to\_field) if to\_field else self.opts.pk.attname

obj\_id = obj.serializable\_value(attr)

self.log\_deletions(request, [obj])

self.delete\_model(request, obj)

return self.response\_delete(request, obj\_display, obj\_id)

object\_name = str(self.opts.verbose\_name)

if perms\_needed or protected:

title = \_("Cannot delete %(name)s") % {"name": object\_name}

else:

title = \_("Are you sure?")

context = {

\*\*self.admin\_site.each\_context(request),

"title": title,

"subtitle": None,

"object\_name": object\_name,

"object": obj,

"deleted\_objects": deleted\_objects,

"model\_count": dict(model\_count).items(),

"perms\_lacking": perms\_needed,

"protected": protected,

"opts": self.opts,

"app\_label": app\_label,

"preserved\_filters": self.get\_preserved\_filters(request),

"is\_popup": IS\_POPUP\_VAR in request.POST or IS\_POPUP\_VAR in request.GET,

"to\_field": to\_field,

\*\*(extra\_context or {}),

}

return self.render\_delete\_form(request, context)

def history\_view(self, request, object\_id, extra\_context=None):

"The 'history' admin view for this model."

from django.contrib.admin.models import LogEntry

from django.contrib.admin.views.main import PAGE\_VAR

# First check if the user can see this history.

model = self.model

obj = self.get\_object(request, unquote(object\_id))

if obj is None:

return self.\_get\_obj\_does\_not\_exist\_redirect(

request, model.\_meta, object\_id

)

if not self.has\_view\_or\_change\_permission(request, obj):

raise PermissionDenied

# Then get the history for this object.

app\_label = self.opts.app\_label

action\_list = (

LogEntry.objects.filter(

object\_id=unquote(object\_id),

content\_type=get\_content\_type\_for\_model(model),

)

.select\_related()

.order\_by("action\_time")

)

paginator = self.get\_paginator(request, action\_list, 100)

page\_number = request.GET.get(PAGE\_VAR, 1)

page\_obj = paginator.get\_page(page\_number)

page\_range = paginator.get\_elided\_page\_range(page\_obj.number)

context = {

\*\*self.admin\_site.each\_context(request),

"title": \_("Change history: %s") % obj,

"subtitle": None,

"action\_list": page\_obj,

"page\_range": page\_range,

"page\_var": PAGE\_VAR,

"pagination\_required": paginator.count > 100,

"module\_name": str(capfirst(self.opts.verbose\_name\_plural)),

"object": obj,

"opts": self.opts,

"preserved\_filters": self.get\_preserved\_filters(request),

\*\*(extra\_context or {}),

}

request.current\_app = self.admin\_site.name

return TemplateResponse(

request,

self.object\_history\_template

or [

"admin/%s/%s/object\_history.html" % (app\_label, self.opts.model\_name),

"admin/%s/object\_history.html" % app\_label,

"admin/object\_history.html",

],

context,

)

def get\_formset\_kwargs(self, request, obj, inline, prefix):

formset\_params = {

"instance": obj,

"prefix": prefix,

"queryset": inline.get\_queryset(request),

}

if request.method == "POST":

formset\_params.update(

{

"data": request.POST.copy(),

"files": request.FILES,

"save\_as\_new": "\_saveasnew" in request.POST,

}

)

return formset\_params

def \_create\_formsets(self, request, obj, change):

"Helper function to generate formsets for add/change\_view."

formsets = []

inline\_instances = []

prefixes = {}

get\_formsets\_args = [request]

if change:

get\_formsets\_args.append(obj)

for FormSet, inline in self.get\_formsets\_with\_inlines(\*get\_formsets\_args):

prefix = FormSet.get\_default\_prefix()

prefixes[prefix] = prefixes.get(prefix, 0) + 1

if prefixes[prefix] != 1 or not prefix:

prefix = "%s-%s" % (prefix, prefixes[prefix])

formset\_params = self.get\_formset\_kwargs(request, obj, inline, prefix)

formset = FormSet(\*\*formset\_params)

def user\_deleted\_form(request, obj, formset, index, inline):

"""Return whether or not the user deleted the form."""

return (

inline.has\_delete\_permission(request, obj)

and "{}-{}-DELETE".format(formset.prefix, index) in request.POST

)

# Bypass validation of each view-only inline form (since the form's

# data won't be in request.POST), unless the form was deleted.

if not inline.has\_change\_permission(request, obj if change else None):

for index, form in enumerate(formset.initial\_forms):

if user\_deleted\_form(request, obj, formset, index, inline):

continue

form.\_errors = {}

form.cleaned\_data = form.initial

formsets.append(formset)

inline\_instances.append(inline)

return formsets, inline\_instances

class InlineModelAdmin(BaseModelAdmin):

"""

Options for inline editing of ``model`` instances.

Provide ``fk\_name`` to specify the attribute name of the ``ForeignKey``

from ``model`` to its parent. This is required if ``model`` has more than

one ``ForeignKey`` to its parent.

"""

model = None

fk\_name = None

formset = BaseInlineFormSet

extra = 3

min\_num = None

max\_num = None

template = None

verbose\_name = None

verbose\_name\_plural = None

can\_delete = True

show\_change\_link = False

checks\_class = InlineModelAdminChecks

classes = None

def \_\_init\_\_(self, parent\_model, admin\_site):

self.admin\_site = admin\_site

self.parent\_model = parent\_model

self.opts = self.model.\_meta

self.has\_registered\_model = admin\_site.is\_registered(self.model)

super().\_\_init\_\_()

if self.verbose\_name\_plural is None:

if self.verbose\_name is None:

self.verbose\_name\_plural = self.opts.verbose\_name\_plural

else:

self.verbose\_name\_plural = format\_lazy("{}s", self.verbose\_name)

if self.verbose\_name is None:

self.verbose\_name = self.opts.verbose\_name

@property

def media(self):

extra = "" if settings.DEBUG else ".min"

js = ["vendor/jquery/jquery%s.js" % extra, "jquery.init.js", "inlines.js"]

if self.filter\_vertical or self.filter\_horizontal:

js.extend(["SelectBox.js", "SelectFilter2.js"])

return forms.Media(js=["admin/js/%s" % url for url in js])

def get\_extra(self, request, obj=None, \*\*kwargs):

"""Hook for customizing the number of extra inline forms."""

return self.extra

def get\_min\_num(self, request, obj=None, \*\*kwargs):

"""Hook for customizing the min number of inline forms."""

return self.min\_num

def get\_max\_num(self, request, obj=None, \*\*kwargs):

"""Hook for customizing the max number of extra inline forms."""

return self.max\_num

def get\_formset(self, request, obj=None, \*\*kwargs):

"""Return a BaseInlineFormSet class for use in admin add/change views."""

if "fields" in kwargs:

fields = kwargs.pop("fields")

else:

fields = flatten\_fieldsets(self.get\_fieldsets(request, obj))

excluded = self.get\_exclude(request, obj)

exclude = [] if excluded is None else list(excluded)

exclude.extend(self.get\_readonly\_fields(request, obj))

if excluded is None and hasattr(self.form, "\_meta") and self.form.\_meta.exclude:

# Take the custom ModelForm's Meta.exclude into account only if the

# InlineModelAdmin doesn't define its own.

exclude.extend(self.form.\_meta.exclude)

# If exclude is an empty list we use None, since that's the actual

# default.

exclude = exclude or None

can\_delete = self.can\_delete and self.has\_delete\_permission(request, obj)

defaults = {

"form": self.form,

"formset": self.formset,

"fk\_name": self.fk\_name,

"fields": fields,

"exclude": exclude,

"formfield\_callback": partial(self.formfield\_for\_dbfield, request=request),

"extra": self.get\_extra(request, obj, \*\*kwargs),

"min\_num": self.get\_min\_num(request, obj, \*\*kwargs),

"max\_num": self.get\_max\_num(request, obj, \*\*kwargs),

"can\_delete": can\_delete,

\*\*kwargs,

}

base\_model\_form = defaults["form"]

can\_change = self.has\_change\_permission(request, obj) if request else True

can\_add = self.has\_add\_permission(request, obj) if request else True

class DeleteProtectedModelForm(base\_model\_form):

def hand\_clean\_DELETE(self):

"""

We don't validate the 'DELETE' field itself because on

templates it's not rendered using the field information, but

just using a generic "deletion\_field" of the InlineModelAdmin.

"""

if self.cleaned\_data.get(DELETION\_FIELD\_NAME, False):

using = router.db\_for\_write(self.\_meta.model)

collector = NestedObjects(using=using)

if self.instance.\_state.adding:

return

collector.collect([self.instance])

if collector.protected:

objs = []

for p in collector.protected:

objs.append(

# Translators: Model verbose name and instance

# representation, suitable to be an item in a

# list.

\_("%(class\_name)s %(instance)s")

% {"class\_name": p.\_meta.verbose\_name, "instance": p}

)

params = {

"class\_name": self.\_meta.model.\_meta.verbose\_name,

"instance": self.instance,

"related\_objects": get\_text\_list(objs, \_("and")),

}

msg = \_(

"Deleting %(class\_name)s %(instance)s would require "

"deleting the following protected related objects: "

"%(related\_objects)s"

)

raise ValidationError(

msg, code="deleting\_protected", params=params

)

def is\_valid(self):

result = super().is\_valid()

self.hand\_clean\_DELETE()

return result

def has\_changed(self):

# Protect against unauthorized edits.

if not can\_change and not self.instance.\_state.adding:

return False

if not can\_add and self.instance.\_state.adding:

return False

return super().has\_changed()

defaults["form"] = DeleteProtectedModelForm

if defaults["fields"] is None and not modelform\_defines\_fields(

defaults["form"]

):

defaults["fields"] = forms.ALL\_FIELDS

return inlineformset\_factory(self.parent\_model, self.model, \*\*defaults)

def \_get\_form\_for\_get\_fields(self, request, obj=None):

return self.get\_formset(request, obj, fields=None).form

def get\_queryset(self, request):

queryset = super().get\_queryset(request)

if not self.has\_view\_or\_change\_permission(request):

queryset = queryset.none()

return queryset

def \_has\_any\_perms\_for\_target\_model(self, request, perms):

"""

This method is called only when the ModelAdmin's model is for an

ManyToManyField's implicit through model (if self.opts.auto\_created).

Return True if the user has any of the given permissions ('add',

'change', etc.) for the model that points to the through model.

"""

opts = self.opts

# Find the target model of an auto-created many-to-many relationship.

for field in opts.fields:

if field.remote\_field and field.remote\_field.model != self.parent\_model:

opts = field.remote\_field.model.\_meta

break

return any(

request.user.has\_perm(

"%s.%s" % (opts.app\_label, get\_permission\_codename(perm, opts))

)

for perm in perms

)

def has\_add\_permission(self, request, obj):

if self.opts.auto\_created:

# Auto-created intermediate models don't have their own

# permissions. The user needs to have the change permission for the

# related model in order to be able to do anything with the

# intermediate model.

return self.\_has\_any\_perms\_for\_target\_model(request, ["change"])

return super().has\_add\_permission(request)

def has\_change\_permission(self, request, obj=None):

if self.opts.auto\_created:

# Same comment as has\_add\_permission().

return self.\_has\_any\_perms\_for\_target\_model(request, ["change"])

return super().has\_change\_permission(request)

def has\_delete\_permission(self, request, obj=None):

if self.opts.auto\_created:

# Same comment as has\_add\_permission().

return self.\_has\_any\_perms\_for\_target\_model(request, ["change"])

return super().has\_delete\_permission(request, obj)

def has\_view\_permission(self, request, obj=None):

if self.opts.auto\_created:

# Same comment as has\_add\_permission(). The 'change' permission

# also implies the 'view' permission.

return self.\_has\_any\_perms\_for\_target\_model(request, ["view", "change"])

return super().has\_view\_permission(request)

class StackedInline(InlineModelAdmin):

template = "admin/edit\_inline/stacked.html"

class TabularInline(InlineModelAdmin):

template = "admin/edit\_inline/tabular.html"