# URLs Documentation

## File: devsearch/urls.py

from django.contrib import admin  
from django.urls import path, include  
from django.conf import settings  
from django.conf.urls.static import static  
from django.contrib.auth import views as auth\_views  
  
  
  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('projects/', include('projects.urls')),  
 path('', include('users.urls')),  
 path('jobs/', include('jobs.urls')),  
  
 path('reset\_password/', auth\_views.PasswordResetView.as\_view(template\_name='reset\_password.html'), name='reset\_password'),  
 path('reset\_password\_sent/', auth\_views.PasswordResetDoneView.as\_view(template\_name='reset\_password\_sent.html'), name="password\_reset\_done"),  
 path('reset/<uidb64>/<token>/', auth\_views.PasswordResetConfirmView.as\_view(template\_name='password\_reset\_confirm.html'), name="password\_reset\_confirm"),  
 path('reset\_password\_complete/', auth\_views.PasswordResetCompleteView.as\_view(template\_name='password\_reset\_complete.html'), name="password\_reset\_complete")  
]  
urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)  
urlpatterns += static(settings.STATIC\_URL, document\_root=settings.STATIC\_ROOT)  
urlpatterns += [  
 path('captcha/', include('captcha.urls')),  
]

## File: jobs/urls.py

from django.urls import path  
from . import views  
  
  
urlpatterns = [  
 path('', views.jobs, name="jobs"),  
 path('job/<str:pk>', views.job, name="job"),  
 path('create-job/', views.createJob, name="create-job"),  
 path('delete-job/<str:pk>', views.deleteJob, name="delete-job"),  
 path('update-job/<str:pk>', views.updateJob, name='update-job'),  
  
 path('addclick/<str:pk>', views.addClick, name="add-click"),  
 path('clikcs/<str:pk>', views.clicks, name="clicks"),  
 path('assignjob/<str:pk>/<str:sk>', views.assignJob, name="assign-job"),  
 path('deactivate/<str:pk>', views.changeJobStatus, name="change-status"),  
 path('create-contract/<str:pk>/<str:sk>', views.createContract, name="create-contract"),  
 path('contract/<str:pk>', views.contract, name='contract')  
 ]

## File: projects/urls.py

from django.contrib import admin  
from django.urls import path  
from . import views  
  
  
  
urlpatterns = [  
  
 path('', views.projects, name="projects"),  
 path('project/<str:pk>/', views.project, name="project"),  
 path('create-project', views.createProject, name="create-project"),  
 path('update-project/<str:pk>/', views.updateProject, name="update-project"),  
 path('delete-project/<str:pk>', views.deleteProject, name="delete-project")  
]

## File: users/urls.py

from django.contrib import admin  
from django.urls import path, re\_path  
from . import views  
  
  
  
urlpatterns = [  
 path('login/', views.loginUser, name="login"),  
 path('logout/', views.logoutUser, name="logout"),  
 path('register/', views.registerUser, name="register"),  
 path('', views.profiles, name="profiles"),  
 path('user/<str:pk>', views.user\_profile, name="user\_profile"),  
 path('account/', views.userAccount, name="account"),  
 path('edit-account/', views.editAccount, name="edit-account"),  
 path('create-skill/', views.createSkill, name="create-skill"),  
 path('delete-skill/<str:pk>', views.deleteSkill, name='delete-skill'),  
 path('edit-skill/<str:pk>', views.editSkill, name='edit-skill'),  
 path('inbox/', views.inbox, name='inbox'),  
 path('message/<str:pk>', views.viewMessage, name='view-message'),  
 path('create-message/<str:pk>', views.createMessage, name='create-message'),  
 path('no\_user/<str:pk>', views.noUser, name='no-user'),  
 path('set-profile-type/', views.setProfileType, name='set-type'),  
 path('companies/', views.companies, name='companies'),  
 path('user-agreement/<str:pk>', views.userAgreement, name='user-agreement')  
 # path('activate/<uidb64>/<token>', views.activate, name='activate')  
  
]

## File: venv/lib/python3.12/site-packages/captcha/tests/urls.py

from django.urls import include, re\_path  
  
from .views import (  
 test,  
 test\_custom\_error\_message,  
 test\_custom\_generator,  
 test\_id\_prefix,  
 test\_model\_form,  
 test\_non\_required,  
 test\_per\_form\_format,  
)  
  
  
urlpatterns = [  
 re\_path(r"test/$", test, name="captcha-test"),  
 re\_path(r"test-modelform/$", test\_model\_form, name="captcha-test-model-form"),  
 re\_path(  
 r"test2/$", test\_custom\_error\_message, name="captcha-test-custom-error-message"  
 ),  
 re\_path(r"test3/$", test\_per\_form\_format, name="test\_per\_form\_format"),  
 re\_path(r"custom-generator/$", test\_custom\_generator, name="test\_custom\_generator"),  
 re\_path(  
 r"test-non-required/$", test\_non\_required, name="captcha-test-non-required"  
 ),  
 re\_path(r"test-id-prefix/$", test\_id\_prefix, name="captcha-test-id-prefix"),  
 re\_path(r"", include("captcha.urls")),  
]

## File: venv/lib/python3.12/site-packages/captcha/urls.py

from django.urls import re\_path  
  
from captcha import views  
  
  
urlpatterns = [  
 re\_path(  
 r"image/(?P<key>\w+)/$",  
 views.captcha\_image,  
 name="captcha-image",  
 kwargs={"scale": 1},  
 ),  
 re\_path(  
 r"image/(?P<key>\w+)@2/$",  
 views.captcha\_image,  
 name="captcha-image-2x",  
 kwargs={"scale": 2},  
 ),  
 re\_path(r"audio/(?P<key>\w+).wav$", views.captcha\_audio, name="captcha-audio"),  
 re\_path(r"refresh/$", views.captcha\_refresh, name="captcha-refresh"),  
]

## File: venv/lib/python3.12/site-packages/django/contrib/admin/templatetags/admin\_urls.py

from urllib.parse import parse\_qsl, unquote, urlparse, urlunparse  
  
from django import template  
from django.contrib.admin.utils import quote  
from django.urls import Resolver404, get\_script\_prefix, resolve  
from django.utils.http import urlencode  
  
register = template.Library()  
  
  
@register.filter  
def admin\_urlname(value, arg):  
 return "admin:%s\_%s\_%s" % (value.app\_label, value.model\_name, arg)  
  
  
@register.filter  
def admin\_urlquote(value):  
 return quote(value)  
  
  
@register.simple\_tag(takes\_context=True)  
def add\_preserved\_filters(context, url, popup=False, to\_field=None):  
 opts = context.get("opts")  
 preserved\_filters = context.get("preserved\_filters")  
  
 parsed\_url = list(urlparse(url))  
 parsed\_qs = dict(parse\_qsl(parsed\_url[4]))  
 merged\_qs = {}  
  
 if opts and preserved\_filters:  
 preserved\_filters = dict(parse\_qsl(preserved\_filters))  
  
 match\_url = "/%s" % unquote(url).partition(get\_script\_prefix())[2]  
 try:  
 match = resolve(match\_url)  
 except Resolver404:  
 pass  
 else:  
 current\_url = "%s:%s" % (match.app\_name, match.url\_name)  
 changelist\_url = "admin:%s\_%s\_changelist" % (  
 opts.app\_label,  
 opts.model\_name,  
 )  
 if (  
 changelist\_url == current\_url  
 and "\_changelist\_filters" in preserved\_filters  
 ):  
 preserved\_filters = dict(  
 parse\_qsl(preserved\_filters["\_changelist\_filters"])  
 )  
  
 merged\_qs.update(preserved\_filters)  
  
 if popup:  
 from django.contrib.admin.options import IS\_POPUP\_VAR  
  
 merged\_qs[IS\_POPUP\_VAR] = 1  
 if to\_field:  
 from django.contrib.admin.options import TO\_FIELD\_VAR  
  
 merged\_qs[TO\_FIELD\_VAR] = to\_field  
  
 merged\_qs.update(parsed\_qs)  
  
 parsed\_url[4] = urlencode(merged\_qs)  
 return urlunparse(parsed\_url)

## File: venv/lib/python3.12/site-packages/django/contrib/admindocs/urls.py

from django.contrib.admindocs import views  
from django.urls import path, re\_path  
  
urlpatterns = [  
 path(  
 "",  
 views.BaseAdminDocsView.as\_view(template\_name="admin\_doc/index.html"),  
 name="django-admindocs-docroot",  
 ),  
 path(  
 "bookmarklets/",  
 views.BookmarkletsView.as\_view(),  
 name="django-admindocs-bookmarklets",  
 ),  
 path(  
 "tags/",  
 views.TemplateTagIndexView.as\_view(),  
 name="django-admindocs-tags",  
 ),  
 path(  
 "filters/",  
 views.TemplateFilterIndexView.as\_view(),  
 name="django-admindocs-filters",  
 ),  
 path(  
 "views/",  
 views.ViewIndexView.as\_view(),  
 name="django-admindocs-views-index",  
 ),  
 path(  
 "views/<view>/",  
 views.ViewDetailView.as\_view(),  
 name="django-admindocs-views-detail",  
 ),  
 path(  
 "models/",  
 views.ModelIndexView.as\_view(),  
 name="django-admindocs-models-index",  
 ),  
 re\_path(  
 r"^models/(?P<app\_label>[^\.]+)\.(?P<model\_name>[^/]+)/$",  
 views.ModelDetailView.as\_view(),  
 name="django-admindocs-models-detail",  
 ),  
 path(  
 "templates/<path:template>/",  
 views.TemplateDetailView.as\_view(),  
 name="django-admindocs-templates",  
 ),  
]

## File: venv/lib/python3.12/site-packages/django/contrib/auth/urls.py

# The views used below are normally mapped in the AdminSite instance.  
# This URLs file is used to provide a reliable view deployment for test purposes.  
# It is also provided as a convenience to those who want to deploy these URLs  
# elsewhere.  
  
from django.contrib.auth import views  
from django.urls import path  
  
urlpatterns = [  
 path("login/", views.LoginView.as\_view(), name="login"),  
 path("logout/", views.LogoutView.as\_view(), name="logout"),  
 path(  
 "password\_change/", views.PasswordChangeView.as\_view(), name="password\_change"  
 ),  
 path(  
 "password\_change/done/",  
 views.PasswordChangeDoneView.as\_view(),  
 name="password\_change\_done",  
 ),  
 path("password\_reset/", views.PasswordResetView.as\_view(), name="password\_reset"),  
 path(  
 "password\_reset/done/",  
 views.PasswordResetDoneView.as\_view(),  
 name="password\_reset\_done",  
 ),  
 path(  
 "reset/<uidb64>/<token>/",  
 views.PasswordResetConfirmView.as\_view(),  
 name="password\_reset\_confirm",  
 ),  
 path(  
 "reset/done/",  
 views.PasswordResetCompleteView.as\_view(),  
 name="password\_reset\_complete",  
 ),  
]

## File: venv/lib/python3.12/site-packages/django/contrib/flatpages/urls.py

from django.contrib.flatpages import views  
from django.urls import path  
  
urlpatterns = [  
 path("<path:url>", views.flatpage, name="django.contrib.flatpages.views.flatpage"),  
]

## File: venv/lib/python3.12/site-packages/django/contrib/staticfiles/urls.py

from django.conf import settings  
from django.conf.urls.static import static  
from django.contrib.staticfiles.views import serve  
  
urlpatterns = []  
  
  
def staticfiles\_urlpatterns(prefix=None):  
 """  
 Helper function to return a URL pattern for serving static files.  
 """  
 if prefix is None:  
 prefix = settings.STATIC\_URL  
 return static(prefix, view=serve)  
  
  
# Only append if urlpatterns are empty  
if settings.DEBUG and not urlpatterns:  
 urlpatterns += staticfiles\_urlpatterns()

## File: venv/lib/python3.12/site-packages/django/core/checks/urls.py

from collections import Counter  
  
from django.conf import settings  
  
from . import Error, Tags, Warning, register  
  
  
@register(Tags.urls)  
def check\_url\_config(app\_configs, \*\*kwargs):  
 if getattr(settings, "ROOT\_URLCONF", None):  
 from django.urls import get\_resolver  
  
 resolver = get\_resolver()  
 return check\_resolver(resolver)  
 return []  
  
  
def check\_resolver(resolver):  
 """  
 Recursively check the resolver.  
 """  
 check\_method = getattr(resolver, "check", None)  
 if check\_method is not None:  
 return check\_method()  
 elif not hasattr(resolver, "resolve"):  
 return get\_warning\_for\_invalid\_pattern(resolver)  
 else:  
 return []  
  
  
@register(Tags.urls)  
def check\_url\_namespaces\_unique(app\_configs, \*\*kwargs):  
 """  
 Warn if URL namespaces used in applications aren't unique.  
 """  
 if not getattr(settings, "ROOT\_URLCONF", None):  
 return []  
  
 from django.urls import get\_resolver  
  
 resolver = get\_resolver()  
 all\_namespaces = \_load\_all\_namespaces(resolver)  
 counter = Counter(all\_namespaces)  
 non\_unique\_namespaces = [n for n, count in counter.items() if count > 1]  
 errors = []  
 for namespace in non\_unique\_namespaces:  
 errors.append(  
 Warning(  
 "URL namespace '{}' isn't unique. You may not be able to reverse "  
 "all URLs in this namespace".format(namespace),  
 id="urls.W005",  
 )  
 )  
 return errors  
  
  
def \_load\_all\_namespaces(resolver, parents=()):  
 """  
 Recursively load all namespaces from URL patterns.  
 """  
 url\_patterns = getattr(resolver, "url\_patterns", [])  
 namespaces = [  
 ":".join(parents + (url.namespace,))  
 for url in url\_patterns  
 if getattr(url, "namespace", None) is not None  
 ]  
 for pattern in url\_patterns:  
 namespace = getattr(pattern, "namespace", None)  
 current = parents  
 if namespace is not None:  
 current += (namespace,)  
 namespaces.extend(\_load\_all\_namespaces(pattern, current))  
 return namespaces  
  
  
def get\_warning\_for\_invalid\_pattern(pattern):  
 """  
 Return a list containing a warning that the pattern is invalid.  
  
 describe\_pattern() cannot be used here, because we cannot rely on the  
 urlpattern having regex or name attributes.  
 """  
 if isinstance(pattern, str):  
 hint = (  
 "Try removing the string '{}'. The list of urlpatterns should not "  
 "have a prefix string as the first element.".format(pattern)  
 )  
 elif isinstance(pattern, tuple):  
 hint = "Try using path() instead of a tuple."  
 else:  
 hint = None  
  
 return [  
 Error(  
 "Your URL pattern {!r} is invalid. Ensure that urlpatterns is a list "  
 "of path() and/or re\_path() instances.".format(pattern),  
 hint=hint,  
 id="urls.E004",  
 )  
 ]  
  
  
@register(Tags.urls)  
def check\_url\_settings(app\_configs, \*\*kwargs):  
 errors = []  
 for name in ("STATIC\_URL", "MEDIA\_URL"):  
 value = getattr(settings, name)  
 if value and not value.endswith("/"):  
 errors.append(E006(name))  
 return errors  
  
  
def E006(name):  
 return Error(  
 "The {} setting must end with a slash.".format(name),  
 id="urls.E006",  
 )

## File: venv/lib/python3.12/site-packages/pip/\_internal/utils/urls.py

import os  
import string  
import urllib.parse  
import urllib.request  
from typing import Optional  
  
from .compat import WINDOWS  
  
  
def get\_url\_scheme(url: str) -> Optional[str]:  
 if ":" not in url:  
 return None  
 return url.split(":", 1)[0].lower()  
  
  
def path\_to\_url(path: str) -> str:  
 """  
 Convert a path to a file: URL. The path will be made absolute and have  
 quoted path parts.  
 """  
 path = os.path.normpath(os.path.abspath(path))  
 url = urllib.parse.urljoin("file:", urllib.request.pathname2url(path))  
 return url  
  
  
def url\_to\_path(url: str) -> str:  
 """  
 Convert a file: URL to a path.  
 """  
 assert url.startswith(  
 "file:"  
 ), f"You can only turn file: urls into filenames (not {url!r})"  
  
 \_, netloc, path, \_, \_ = urllib.parse.urlsplit(url)  
  
 if not netloc or netloc == "localhost":  
 # According to RFC 8089, same as empty authority.  
 netloc = ""  
 elif WINDOWS:  
 # If we have a UNC path, prepend UNC share notation.  
 netloc = "\\\\" + netloc  
 else:  
 raise ValueError(  
 f"non-local file URIs are not supported on this platform: {url!r}"  
 )  
  
 path = urllib.request.url2pathname(netloc + path)  
  
 # On Windows, urlsplit parses the path as something like "/C:/Users/foo".  
 # This creates issues for path-related functions like io.open(), so we try  
 # to detect and strip the leading slash.  
 if (  
 WINDOWS  
 and not netloc # Not UNC.  
 and len(path) >= 3  
 and path[0] == "/" # Leading slash to strip.  
 and path[1] in string.ascii\_letters # Drive letter.  
 and path[2:4] in (":", ":/") # Colon + end of string, or colon + absolute path.  
 ):  
 path = path[1:]  
  
 return path

## File: venv/lib/python3.12/site-packages/rest\_framework/urls.py

"""  
Login and logout views for the browsable API.  
  
Add these to your root URLconf if you're using the browsable API and  
your API requires authentication:  
  
 urlpatterns = [  
 ...  
 path('auth/', include('rest\_framework.urls'))  
 ]  
  
You should make sure your authentication settings include `SessionAuthentication`.  
"""  
from django.contrib.auth import views  
from django.urls import path  
  
app\_name = 'rest\_framework'  
urlpatterns = [  
 path('login/', views.LoginView.as\_view(template\_name='rest\_framework/login.html'), name='login'),  
 path('logout/', views.LogoutView.as\_view(), name='logout'),  
]

## File: venv/lib/python3.12/site-packages/rest\_framework/utils/urls.py

from urllib import parse  
  
from django.utils.encoding import force\_str  
  
  
def replace\_query\_param(url, key, val):  
 """  
 Given a URL and a key/val pair, set or replace an item in the query  
 parameters of the URL, and return the new URL.  
 """  
 (scheme, netloc, path, query, fragment) = parse.urlsplit(force\_str(url))  
 query\_dict = parse.parse\_qs(query, keep\_blank\_values=True)  
 query\_dict[force\_str(key)] = [force\_str(val)]  
 query = parse.urlencode(sorted(query\_dict.items()), doseq=True)  
 return parse.urlunsplit((scheme, netloc, path, query, fragment))  
  
  
def remove\_query\_param(url, key):  
 """  
 Given a URL and a key/val pair, remove an item in the query  
 parameters of the URL, and return the new URL.  
 """  
 (scheme, netloc, path, query, fragment) = parse.urlsplit(force\_str(url))  
 query\_dict = parse.parse\_qs(query, keep\_blank\_values=True)  
 query\_dict.pop(key, None)  
 query = parse.urlencode(sorted(query\_dict.items()), doseq=True)  
 return parse.urlunsplit((scheme, netloc, path, query, fragment))

# Views Documentation

## File: jobs/views.py

from django.shortcuts import render, redirect  
from django.http import HttpResponse  
from .models import Job, Contract  
from .forms import JobForm, ContractForm  
from projects.models import Tag  
from django.contrib.auth.decorators import login\_required  
from django.contrib import messages  
from users.models import Profile  
from users.views import createMessage  
from .utils import generate\_contract\_pdf, searchJobs, paginateProfiles  
from users.models import Message  
from users.forms import MessageForm  
# Create your views here.  
  
  
def jobs(request):  
 jobObj1, search\_query = searchJobs(request)  
 jobObj = tuple(x for x in jobObj1 if not x.is\_assigned and x.is\_active)  
 custom\_range, jobObj = paginateProfiles(request, jobObj, 6)  
 return render(request, 'jobs.html',  
 {'jobs': jobObj, 'search\_query': search\_query, 'custom\_range': custom\_range})  
   
  
  
   
  
def job(request, pk):  
 job = Job.objects.get(id=pk)  
 tags = job.tags.all()  
 context = {'job': job, 'tags': tags}  
 return render(request, 'job.html', context)  
  
  
  
  
  
@login\_required(login\_url="login")  
def createJob(request):  
 profile = request.user.profile  
 form = JobForm(request.POST)  
 if request.method == 'POST':  
 newtags = request.POST.get('newtags').replace(',', ' ').split()  
 form = JobForm(request.POST, request.FILES)  
 if form.is\_valid():  
 job = form.save(commit=False)  
 job.owner = profile  
 job.save()  
 form.save\_m2m()  
 for tag in newtags:  
 tag, created = Tag.objects.get\_or\_create(name=tag)  
 job.tags.add(tag)  
 return redirect('account')  
 else:  
 return HttpResponse('wrong')  
  
 context = {'form': form,}  
 return render(request, 'job\_form.html', context)  
  
@login\_required(login\_url="login")  
def createContract(request, pk, sk):  
 profile = request.user.profile  
 freelancer = Profile.objects.get(id=pk)  
 job = Job.objects.get(id=sk)  
 form = ContractForm(request.POST)  
 if request.method == 'POST':  
 form = ContractForm(request.POST, request.FILES)  
 if form.is\_valid():  
 contract = form.save(commit=False)  
 contract.client = job.owner  
 contract.freelancer = freelancer  
 contract.job = job  
 contract.save()  
 # return generate\_contract\_pdf(request, contract.id)  
 createTaskMessage(request, freelancer.id, contract.id)  
 return generate\_contract\_pdf(request, contract.id)  
 # return redirect('contract', contract.id)  
 else:  
 return HttpResponse('wrong')  
 else:  
 # If it's a GET request, pre-fill the form with job data  
 initial\_data = {  
 'job': job,  
 'freelancer': freelancer,  
 'client': job.owner,  
 'title': job.title,  
 'budget': job.budget,  
 'duration': job.duration,  
 'featured\_image': job.featured\_image,  
 'description': job.description,  
 }  
  
 form = ContractForm(initial=initial\_data)  
  
 context = {'form': form,}  
 return render(request, 'contract\_form.html', context)  
  
def createTaskMessage(request, pk, sk):  
 contract = Contract.objects.get(id=sk)  
 recipient = Profile.objects.get(id=pk)  
 sender = request.user.profile  
 form = MessageForm({  
 'subject': 'Contract Assignment',  
 'body': f'Dear {recipient.name}, I have created a contract that we have recently agreed for this <a href="http://freelance.wiut.uz/jobs/job/{contract.job.id}">task</a>. \n Budget is {contract.budget} soums and you can finish it in {contract.duration}. Please find and download the contract <a href="http://freelance.wiut.uz/jobs/contract/{contract.id}">here</a>. Once you sign, the task will be assigned to you'  
 }  
 )  
 if form.is\_valid:  
 form1=form.save(commit=False)  
 form1.recipient = recipient  
 form1.sender = sender  
 form1.save()  
 else:  
 messages.error(request, "Please fill all the fields accordingly")  
  
def contract(request, pk):  
 contract = Contract.objects.get(id=pk)  
 if request.method == "POST":  
 print(request.POST)  
 combined = request.POST['combined\_vars']  
 contract\_id, freelancer\_id = combined.split('\_')  
 print(contract\_id, freelancer\_id)  
 # freelancer\_id = request.POST['freelancer\_id']  
 contract = Contract.objects.get(id=contract\_id)  
 contract.job.assigned = Profile.objects.get(id=freelancer\_id)  
 contract.job.save()  
 return generate\_contract\_pdf(request, contract.id )  
 context ={'contract': contract}  
 return render(request, 'contract.html', context=context)  
  
  
@login\_required(login\_url="login")  
def updateJob(request, pk):  
 profile = request.user.profile  
 job = profile.job\_set.get(id=pk)  
 form = JobForm(instance=job)  
 if request.method == 'POST':  
 newtags = request.POST.get('newtags').replace(',', ' ').split()  
 form = JobForm(request.POST, request.FILES, instance=job)  
 if form.is\_valid():  
 job = form.save(commit=False)  
 job.owner = profile  
 job.save()  
 form.save\_m2m()  
 for tag in newtags:  
 tag, created = Tag.objects.get\_or\_create(name=tag)  
 job.tags.add(tag)  
 return redirect('account')  
 else:  
 return HttpResponse('wrong')  
 context = {'form': form, }  
 return render(request, 'job\_form.html', context)  
  
@login\_required(login\_url="login")  
def deleteJob(request, pk):  
 job = Job.objects.get(id=pk)  
 if request.method == 'POST':  
 job.delete()  
 return redirect('account')  
 context = {'job': job}  
 return render(request, 'delete-job.html', context)  
  
  
@login\_required(login\_url="login")  
def addClick(request, pk):  
 job = Job.objects.get(id=pk)  
 if request.method == 'POST':  
 profile = request.user.profile  
 if profile not in job.clicked.all():  
 job.click\_total = job.click\_total + 1  
 job.clicked.add(profile)  
 job.save()  
 messages.success(request, 'Your click has been added')  
 else:  
 messages.warning(request, 'You have already clicked')  
  
 return redirect('job', job.id)  
  
  
@login\_required(login\_url="login")  
def clicks(request, pk):  
 job = Job.objects.get(id=pk)  
 clicks = job.clicked.all()  
 context = {'clicks': clicks, 'jobid': pk, 'job': job}  
 return render(request, 'clicks.html', context)  
  
  
def assignJob(request, pk, sk):  
 job=Job.objects.get(id=pk)  
 profile = Profile.objects.get(id=sk)  
 name = profile.name  
 job.assigned = profile  
 job.is\_assigned = True  
 job.save()  
 messages.success(request, 'The job is now assigned to ', name)  
 return redirect('clicks', pk)  
  
  
login\_required(login\_url="login")  
def changeJobStatus(request, pk):  
 profile = request.user.profile  
 job = Job.objects.get(id=pk)  
 if job.is\_active:  
 job.is\_active = False  
 job.clicked.clear()  
 job.assigned = None  
 job.click\_total = 0  
 job.save()  
 else:  
 job.is\_active = True  
 job.save()  
 return redirect('account')

## File: projects/views.py

from django.shortcuts import render, redirect  
from django.http import HttpResponse  
from .models import Project, Tag  
from django.contrib import messages  
from .forms import ProjectForm, ReviewForm  
from .utils import searchProject, paginateProjects  
from django.contrib.auth.decorators import login\_required  
from django.core.paginator import Paginator, PageNotAnInteger, EmptyPage  
  
  
def projects(request):  
 projects, search\_query = searchProject(request)  
 custom\_range, projects = paginateProjects(request, projects, 6)  
 context = {'pl': projects, 'search\_query':search\_query, 'custom\_range':custom\_range}  
 return render(request, 'projects.html', context)  
  
  
def project(request, pk):  
 projectObj = Project.objects.get(id=pk)  
 form = ReviewForm()  
 if request.method == 'POST':  
 form = ReviewForm(request.POST)  
 review = form.save(commit=False)  
 review.project = projectObj  
 review.owner =request.user.profile  
 review.save()  
  
 projectObj.getVoteCount  
  
 messages.success(request, 'Review added')  
 return redirect('project', pk=projectObj.id)  
  
 tags = projectObj.tags.all()  
 return render(request, 'single-project.html', {'project': projectObj, 'tags': tags, 'form': form })  
  
@login\_required(login\_url="login")  
def createProject(request):  
 profile = request.user.profile  
 form = ProjectForm(request.POST)  
 if request.method == 'POST':  
 newtags = request.POST.get('newtags').replace(',', ' ').split()  
 if form.is\_valid():  
 form = ProjectForm(request.POST, request.FILES)  
 project = form.save(commit=False)  
 project.owner = profile  
 project.save()  
 form.save\_m2m()  
 for tag in newtags:  
 tag, created = Tag.objects.get\_or\_create(name=tag)  
 project.tags.add(tag)  
 return redirect('projects')  
 else:  
 return HttpResponse('wrong')  
  
 context = {'form': form,}  
 return render(request, 'project\_form.html', context)  
@login\_required(login\_url="login")  
def updateProject(request, pk):  
 profile = request.user.profile  
 projectObj = profile.project\_set.get(id=pk)  
 form = ProjectForm(instance=projectObj)  
  
 if request.method == 'POST':  
 newtags = request.POST.get('newtags').replace(',', ' ').split()  
  
  
  
 print(request.POST)  
 form = ProjectForm(request.POST, request.FILES, instance=projectObj)  
 if form.is\_valid():  
 projectObj = form.save(commit=False)  
 for tag in newtags:  
 tag, created = Tag.objects.get\_or\_create(name=tag)  
 projectObj.tags.add(tag)  
 projectObj.save()  
 return redirect('account', )  
 context = {'form': form, }  
 return render(request, 'project\_form.html', context)  
  
  
@login\_required(login\_url="login")  
def deleteProject(request, pk):  
 profile = request.user.profile  
 project = profile.project\_set.get(id=pk)  
 context = {'project': project, }  
 if request.method == 'POST':  
 project.delete()  
 return redirect('projects')  
 return render(request, 'delete\_object.html', context)

## File: users/views.py

from django.shortcuts import render, redirect  
from .models import Profile, Skill, Message  
from django.contrib.auth import login, authenticate, logout, get\_user\_model  
from django.contrib.auth.decorators import login\_required  
from .forms import CustomUserCreationForm, ProfileForm, SkillForm, MessageForm, CheckboxForm, LoginForm  
from django.contrib import messages  
from .utils import searchProfiles, paginateProfiles  
from django.contrib.auth.models import User  
from django.db.models import Q  
from django.core.mail import EmailMessage  
from .decorators import user\_is\_active  
from django.utils.encoding import force\_str  
from django.contrib.sites.shortcuts import get\_current\_site  
from django.utils.encoding import force\_bytes  
from django.utils.http import urlsafe\_base64\_encode  
from django.contrib.auth.tokens import default\_token\_generator as account\_activation\_token  
from django.template.loader import render\_to\_string  
from django.utils.http import urlsafe\_base64\_decode  
from django.utils.translation import gettext\_lazy as \_  
from django.http import HttpResponseRedirect  
  
# Create your views here.  
  
  
def loginUser(request):  
 page = 'login'  
 form = LoginForm()  
 context = {'page': page, 'form': form}  
 if request.user.is\_authenticated:  
 return redirect('profiles')  
  
 if request.method == 'POST':  
 form = LoginForm(request.POST)  
 # username = request.POST['username'].lower()  
 # password = request.POST['password']   
 # print(username, password)  
 # Validate the form: the captcha field will automatically  
 # check the input  
 if form.is\_valid():  
 username = request.POST['username'].lower()  
 password = request.POST['password']   
 print("username, password")  
 try:  
 user = User.objects.get(username=username)  
 except:  
 messages.error(request, "Username doesnot exist")  
  
 user = authenticate(request, username=username, password=password)  
 if user is not None:  
 login(request, user)  
 return redirect(request.GET['next'] if 'next' in request.GET else 'account')  
 else:  
 messages.error(request, "Incorrect username or password")  
 else:   
 messages.error(request, form.errors)  
 print(form.errors)  
  
 return render(request, 'login\_register.html', context)  
  
# def loginUser(request):  
# page = 'login'  
# context = {'page': page}  
  
# if request.user.is\_authenticated:  
# return redirect('profiles')  
  
# if request.method == 'POST':  
# username = request.POST['username'].lower()  
# password = request.POST['password']  
   
# try:  
# user = User.objects.get(username=username)  
# except:  
# messages.error(request, "Username doesnot exist")  
  
# user = authenticate(request, username=username, password=password)  
# if user is not None:  
# login(request, user)  
# print("redirect")  
# return redirect(request.GET['next'] if 'next' in request.GET else 'account')  
# else:  
# messages.error(request, "username doesnot exist")  
# return render(request, 'login\_register.html', context)  
  
  
def logoutUser(request):  
 logout(request)  
 return redirect('login')  
  
  
def registerUser(request):  
 page = 'register'  
 form = CustomUserCreationForm()  
 if request.method == 'POST':  
 form = CustomUserCreationForm(request.POST)  
 if form.is\_valid():  
 user = form.save(commit=False)  
 user.username = user.username.lower()  
 print(user.email)  
 user.save()  
 user.is\_active=False  
 messages.success(request, 'User account is created')  
 login(request, user)  
 return redirect('set-type')  
 else:  
 messages.error(request, "An error has occured")  
 context = {'page': page, 'form': form}  
 # return render(request, 'login\_register.html', context)  
 ext\_link = 'https://forms.gle/bp3AhN8BqZAJ3AcH6'  
 return HttpResponseRedirect(ext\_link)  
  
  
def setProfileType(request):  
 profile = request.user.profile  
 checkboxform = CheckboxForm()  
 if request.method == 'POST':  
 checkboxform = CheckboxForm(request.POST)  
 if 'type\_freelancer' in checkboxform.data:  
 profile.user\_type = 'Freelance'  
 profile.save()  
 else:  
 profile.user\_type = 'Client'  
 profile.save()  
 return redirect('edit-account')  
 context = {'checkboxform': checkboxform}  
 return render(request, 'profile\_type\_form.html', context)  
  
  
def profiles(request):  
 profileObj1, search\_query = searchProfiles(request)  
 profileObj = tuple(x for x in profileObj1 if x.user\_type == 'Freelance')  
 for profile in profileObj:  
 print(profile.name,profile.user\_type)  
 custom\_range, profileObj = paginateProfiles(request, profileObj, 9)  
 return render(request, 'profiles.html',  
 {'profiles': profileObj, 'search\_query': search\_query, 'custom\_range': custom\_range})  
  
  
def companies(request):  
 companiesObj1, search\_query = searchProfiles(request)  
 companiesObj = tuple(x for x in companiesObj1 if x.user\_type == 'Client')  
 custom\_range, profileObj = paginateProfiles(request, companiesObj, 9)  
 return render(request, 'companies.html',  
 {'profiles': companiesObj, 'search\_query': search\_query, 'custom\_range': custom\_range})  
  
  
def user\_profile(request, pk):  
 profileObj = Profile.objects.get(id=pk)  
  
 topskill = profileObj.skill\_set.exclude(description\_\_exact="")  
 otherskill = profileObj.skill\_set.filter(description="")  
  
 return render(request, 'user-profile.html', {'profile': profileObj, 'topskill': topskill,  
 'otherskill': otherskill})  
  
  
@login\_required()  
def userAccount(request):  
 profile = request.user.profile  
 skill = profile.skill\_set.all()  
 projects = profile.project\_set.all()  
 jobs = profile.job\_set.all()  
  
 try:  
 assigned = profile.assigned  
 except:  
 assigned = False  
  
 context = {'profile': profile, 'skills': skill, 'projects': projects, 'jobs': jobs, 'assigned\_job': assigned}  
 return render(request, 'account.html', context)  
  
# def activate(request, uidb64, token):  
# User = get\_user\_model()  
# try:  
# uid =force\_str(urlsafe\_base64\_decode(uidb64))  
# user = User.objects.get(pk=uid)  
# except:  
# user = None  
# if user is not None and account\_activation\_token(user, token):  
# user.is\_active = True  
# user.save()  
  
# messages.success(request, 'Account is activated')  
# return redirect("login")  
# else:  
# messages.error(request, "activation link invalid")  
# return redirect('/')  
  
# def activateEmail(request, user, to\_email):  
# mail\_subject = "Activate your user account."  
# message = render\_to\_string("template\_activate\_account.html",{  
# 'user': user.username,  
# 'domain': get\_current\_site(request).domain,  
# 'uid': urlsafe\_base64\_encode(force\_bytes(user.pk)),  
# 'token': account\_activation\_token.make\_token(user),  
# 'protocol': 'https' if request.is\_secure() else 'http'  
# })  
# email = EmailMessage(mail\_subject, message, to=[to\_email])  
# if email.send():  
# print("email is sent")  
# messages.success(request, f'Dear {user}, please go to your email {to\_email} inbox and click on received activation link \  
# to complete registration. <b>Note:<b> Check your spam folder')  
# else:  
# message.error(request, 'Something happend. Check the correctness of the email you have inserted')  
  
@login\_required(login\_url='login')  
def editAccount(request):  
 profile = request.user.profile  
 form = ProfileForm(instance=profile)  
 if request.method == 'POST':  
 print(request.POST)  
 form = ProfileForm(request.POST, request.FILES, instance=profile)  
 if form.is\_valid():  
 form.save()  
 # activateEmail(request, request.user, profile.email)  
 # profile.profile\_pic = request.FILES['profile\_pic']  
 # profile.save()  
 # user\_form = form.save(commit=False)  
 # user\_form.profile\_pic = request.FILES['profile\_pic']  
 # user\_form.save()  
 return redirect('account')  
 else:  
 messages.error(request, "An error has occured")  
 context = {'form': form}  
 return render(request, 'profile\_form.html', context)  
  
  
@login\_required(login\_url="login")  
def createSkill(request):  
 profile = request.user.profile  
 form = SkillForm()  
 if request.method == 'POST':  
 form = SkillForm(request.POST)  
 if form.is\_valid():  
 skill = form.save(commit=False)  
 skill.owner = profile  
 skill.save()  
 return redirect('account')  
 context = {'form': form}  
 return render(request, 'skill\_form.html', context)  
  
  
@login\_required(login\_url="login")  
def deleteSkill(request, pk):  
 skill = Skill.objects.get(id=pk)  
 skill.delete()  
 messages.success(request, 'Skill is deleted')  
 return redirect('account')  
  
  
def editSkill(request, pk):  
 skill = Skill.objects.get(id=pk)  
 form = SkillForm(instance=skill)  
 if request.method == 'POST':  
 form = SkillForm(request.POST, instance=skill)  
 if form.is\_valid():  
 form.save()  
 return redirect('account')  
 context = {'form': form}  
 return render(request, 'skill\_form.html', context)  
  
  
@login\_required(login\_url="login")  
def inbox(request):  
 profile = request.user.profile  
 messageRequests = profile.messages.all()  
 my\_messages = []  
  
 uniqueSenders = []  
 unreadSender = []  
 sent\_by\_me = []  
  
 for message in Message.objects.filter(sender=profile):  
 if message.recipient not in uniqueSenders:  
 uniqueSenders.append(message.recipient)  
  
 for message in messageRequests:  
 if message.sender not in uniqueSenders:  
 uniqueSenders.append(message.sender)  
  
 for message in messageRequests:  
 if not message.is\_read:  
 unreadSender.append(message.sender)  
  
 unreadCount = messageRequests.filter(is\_read=False).count()  
 context = {'unreadSenders': unreadSender, 'unReadCount': unreadCount, 'senders': uniqueSenders}  
 return render(request, 'inbox.html', context)  
  
  
@login\_required(login\_url="login")  
def viewMessage(request, pk):  
 sender = Profile.objects.get(id=pk)  
 recipient = request.user.profile  
 form = MessageForm()  
 if request.method == 'POST':  
 form = MessageForm(request.POST, request.FILES)  
  
 if form.is\_valid():  
 messageObj = form.save(commit=False)  
 messageObj.recipient = sender  
 messageObj.sender = recipient  
 if request.FILES:  
 messageObj.attached = request.FILES['attached']  
 if sender:  
 # messageObj.name = sender.name  
 messageObj.email = recipient.email  
  
 messageObj.save()  
 messages.success(request, 'Message is Sent')  
 return redirect(request.path, pk)  
  
 messageRequest = Message.objects.filter(  
 Q(sender=sender, recipient=recipient) | Q(sender=recipient, recipient=sender))  
 for item in messageRequest:  
 if not item.is\_read:  
 item.is\_read = True  
 item.save()  
 context = {'messageRequest': messageRequest, 'form': form}  
 return render(request, 'message.html', context)  
  
  
def createMessage(request, pk):  
 recipient = Profile.objects.get(id=pk)  
 sender = request.user.profile  
 form = MessageForm()  
 if request.method == 'POST':  
 form = MessageForm(request.POST, request.FILES)  
  
 if form.is\_valid():  
 messageObj = form.save(commit=False)  
 messageObj.recipient = recipient  
 if messageObj.attached:  
 messageObj.attached = request.FILES['attached']  
 messageObj.sender = sender  
 print(request.FILES)  
 if sender:  
 # messageObj.name = sender.name  
 messageObj.email = sender.email  
 messageObj.save()  
 messages.success(request, 'Message is Sent')  
 return redirect('user\_profile', pk)  
  
 context = {'form': form, 'recipient': pk}  
 return render(request, 'message\_form.html', context)  
  
def userAgreement(request, pk):  
 user\_type = pk  
   
 context = {'user\_type':user\_type }  
   
  
 return render(request, 'user\_agreement.html', context)  
  
def noUser(request, pk):  
 messageRequest = Message.objects.get(id=pk)  
 context = {'msg': messageRequest}  
 return render(request, 'no\_user.html', context)

## File: venv/lib/python3.12/site-packages/captcha/tests/views.py

from django import forms  
from django.contrib.auth.models import User  
from django.http import HttpResponse  
from django.template import engines  
  
from captcha.fields import CaptchaField  
  
  
TEST\_TEMPLATE = r"""  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">  
<html>  
 <head>  
 <meta http-equiv="Content-type" content="text/html; charset=utf-8">  
 <title>captcha test</title>  
 </head>  
 <body>  
 {% if passed %}  
 <p style="color:green">Form validated</p>  
 {% endif %}  
 {% if form.errors %}  
 {{form.errors}}  
 {% endif %}  
  
 <form action="{% url 'captcha-test' %}" method="post">  
 {{form.as\_p}}  
 <p><input type="submit" value="Continue &rarr;"></p>  
 </form>  
 </body>  
</html>  
"""  
  
  
def \_get\_template(template\_string):  
 return engines["django"].from\_string(template\_string)  
  
  
def \_test(request, form\_class):  
 passed = False  
 if request.POST:  
 form = form\_class(request.POST)  
 if form.is\_valid():  
 passed = True  
 else:  
 form = form\_class()  
  
 t = \_get\_template(TEST\_TEMPLATE)  
  
 return HttpResponse(  
 t.render(context=dict(passed=passed, form=form), request=request)  
 )  
  
  
def test(request):  
 class CaptchaTestForm(forms.Form):  
 subject = forms.CharField(max\_length=100)  
 sender = forms.EmailField()  
 captcha = CaptchaField(help\_text="asdasd")  
  
 return \_test(request, CaptchaTestForm)  
  
  
def test\_model\_form(request):  
 class CaptchaTestModelForm(forms.ModelForm):  
 subject = forms.CharField(max\_length=100)  
 sender = forms.EmailField()  
 captcha = CaptchaField(help\_text="asdasd")  
  
 class Meta:  
 model = User  
 fields = ("subject", "sender", "captcha")  
  
 return \_test(request, CaptchaTestModelForm)  
  
  
def test\_custom\_generator(request):  
 class CaptchaTestModelForm(forms.ModelForm):  
 subject = forms.CharField(max\_length=100)  
 sender = forms.EmailField()  
 captcha = CaptchaField(generator=lambda: ("111111", "111111"))  
  
 class Meta:  
 model = User  
 fields = ("subject", "sender", "captcha")  
  
 return \_test(request, CaptchaTestModelForm)  
  
  
def test\_custom\_error\_message(request):  
 class CaptchaTestErrorMessageForm(forms.Form):  
 captcha = CaptchaField(  
 help\_text="asdasd", error\_messages=dict(invalid="TEST CUSTOM ERROR MESSAGE")  
 )  
  
 return \_test(request, CaptchaTestErrorMessageForm)  
  
  
def test\_per\_form\_format(request):  
 class CaptchaTestFormatForm(forms.Form):  
 captcha = CaptchaField(  
 help\_text="asdasd",  
 error\_messages=dict(invalid="TEST CUSTOM ERROR MESSAGE"),  
 output\_format=(  
 "%(image)s testPerFieldCustomFormatString "  
 "%(hidden\_field)s %(text\_field)s"  
 ),  
 )  
  
 return \_test(request, CaptchaTestFormatForm)  
  
  
def test\_non\_required(request):  
 class CaptchaTestForm(forms.Form):  
 sender = forms.EmailField()  
 subject = forms.CharField(max\_length=100)  
 captcha = CaptchaField(help\_text="asdasd", required=False)  
  
 return \_test(request, CaptchaTestForm)  
  
  
def test\_id\_prefix(request):  
 class CaptchaTestForm(forms.Form):  
 sender = forms.EmailField()  
 subject = forms.CharField(max\_length=100)  
 captcha1 = CaptchaField(id\_prefix="form1")  
 captcha2 = CaptchaField(id\_prefix="form2")  
  
 return \_test(request, CaptchaTestForm)

## File: venv/lib/python3.12/site-packages/captcha/views.py

import json  
import os  
import random  
import subprocess  
import tempfile  
from io import BytesIO  
  
from PIL import Image, ImageDraw, ImageFont  
from ranged\_response import RangedFileResponse  
  
from django.core.exceptions import ImproperlyConfigured  
from django.http import Http404, HttpResponse  
  
from captcha.conf import settings  
from captcha.helpers import captcha\_audio\_url, captcha\_image\_url  
from captcha.models import CaptchaStore  
  
  
# Distance of the drawn text from the top of the captcha image  
DISTANCE\_FROM\_TOP = 4  
  
  
def getsize(font, text):  
 if hasattr(font, "getbbox"):  
 \_top, \_left, \_right, \_bottom = font.getbbox(text)  
 return \_right - \_left, \_bottom - \_top  
 elif hasattr(font, "getoffset"):  
 return tuple([x + y for x, y in zip(font.getsize(text), font.getoffset(text))])  
 else:  
 return font.getsize(text)  
  
  
def makeimg(size):  
 if settings.CAPTCHA\_BACKGROUND\_COLOR == "transparent":  
 image = Image.new("RGBA", size)  
 else:  
 image = Image.new("RGB", size, settings.CAPTCHA\_BACKGROUND\_COLOR)  
 return image  
  
  
def captcha\_image(request, key, scale=1):  
 if scale == 2 and not settings.CAPTCHA\_2X\_IMAGE:  
 raise Http404  
 try:  
 store = CaptchaStore.objects.get(hashkey=key)  
 except CaptchaStore.DoesNotExist:  
 # HTTP 410 Gone status so that crawlers don't index these expired urls.  
 return HttpResponse(status=410)  
  
 random.seed(key) # Do not generate different images for the same key  
  
 text = store.challenge  
  
 if isinstance(settings.CAPTCHA\_FONT\_PATH, str):  
 fontpath = settings.CAPTCHA\_FONT\_PATH  
 elif isinstance(settings.CAPTCHA\_FONT\_PATH, (list, tuple)):  
 fontpath = random.choice(settings.CAPTCHA\_FONT\_PATH)  
 else:  
 raise ImproperlyConfigured(  
 "settings.CAPTCHA\_FONT\_PATH needs to be a path to a font or list of paths to fonts"  
 )  
  
 if fontpath.lower().strip().endswith("ttf"):  
 font = ImageFont.truetype(fontpath, settings.CAPTCHA\_FONT\_SIZE \* scale)  
 else:  
 font = ImageFont.load(fontpath)  
  
 if settings.CAPTCHA\_IMAGE\_SIZE:  
 size = settings.CAPTCHA\_IMAGE\_SIZE  
 else:  
 size = getsize(font, text)  
 size = (size[0] \* 2, int(size[1] \* 1.4))  
  
 image = makeimg(size)  
 xpos = 2  
  
 charlist = []  
 for char in text:  
 if char in settings.CAPTCHA\_PUNCTUATION and len(charlist) >= 1:  
 charlist[-1] += char  
 else:  
 charlist.append(char)  
 for char in charlist:  
 fgimage = Image.new("RGB", size, settings.CAPTCHA\_FOREGROUND\_COLOR)  
 charimage = Image.new("L", getsize(font, " %s " % char), "#000000")  
 chardraw = ImageDraw.Draw(charimage)  
 chardraw.text((0, 0), " %s " % char, font=font, fill="#ffffff")  
 if settings.CAPTCHA\_LETTER\_ROTATION:  
 charimage = charimage.rotate(  
 random.randrange(\*settings.CAPTCHA\_LETTER\_ROTATION),  
 expand=0,  
 resample=Image.BICUBIC,  
 )  
 charimage = charimage.crop(charimage.getbbox())  
 maskimage = Image.new("L", size)  
  
 maskimage.paste(  
 charimage,  
 (  
 xpos,  
 DISTANCE\_FROM\_TOP,  
 xpos + charimage.size[0],  
 DISTANCE\_FROM\_TOP + charimage.size[1],  
 ),  
 )  
 size = maskimage.size  
 image = Image.composite(fgimage, image, maskimage)  
 xpos = xpos + 2 + charimage.size[0]  
  
 if settings.CAPTCHA\_IMAGE\_SIZE:  
 # centering captcha on the image  
 tmpimg = makeimg(size)  
 tmpimg.paste(  
 image,  
 (  
 int((size[0] - xpos) / 2),  
 int((size[1] - charimage.size[1]) / 2 - DISTANCE\_FROM\_TOP),  
 ),  
 )  
 image = tmpimg.crop((0, 0, size[0], size[1]))  
 else:  
 image = image.crop((0, 0, xpos + 1, size[1]))  
 draw = ImageDraw.Draw(image)  
  
 for f in settings.noise\_functions():  
 draw = f(draw, image)  
 for f in settings.filter\_functions():  
 image = f(image)  
  
 out = BytesIO()  
 image.save(out, "PNG")  
 out.seek(0)  
  
 response = HttpResponse(content\_type="image/png")  
 response.write(out.read())  
 response["Content-length"] = out.tell()  
  
 # At line :50 above we fixed the random seed so that we always generate the  
 # same image, see: https://github.com/mbi/django-simple-captcha/pull/194  
 # This is a problem though, because knowledge of the seed will let an attacker  
 # predict the next random (globally). We therefore reset the random here.  
 # Reported in https://github.com/mbi/django-simple-captcha/pull/221  
 random.seed()  
  
 return response  
  
  
def captcha\_audio(request, key):  
 if settings.CAPTCHA\_FLITE\_PATH:  
 try:  
 store = CaptchaStore.objects.get(hashkey=key)  
 except CaptchaStore.DoesNotExist:  
 # HTTP 410 Gone status so that crawlers don't index these expired urls.  
 return HttpResponse(status=410)  
  
 text = store.challenge  
 if "captcha.helpers.math\_challenge" == settings.CAPTCHA\_CHALLENGE\_FUNCT:  
 text = text.replace("\*", "times").replace("-", "minus").replace("+", "plus")  
 else:  
 text = ", ".join(list(text))  
 path = str(os.path.join(tempfile.gettempdir(), "%s.wav" % key))  
 subprocess.call([settings.CAPTCHA\_FLITE\_PATH, "-t", text, "-o", path])  
  
 # Add arbitrary noise if sox is installed  
 if settings.CAPTCHA\_SOX\_PATH:  
 arbnoisepath = str(  
 os.path.join(tempfile.gettempdir(), "%s\_arbitrary.wav") % key  
 )  
 mergedpath = str(os.path.join(tempfile.gettempdir(), "%s\_merged.wav") % key)  
 subprocess.call(  
 [  
 settings.CAPTCHA\_SOX\_PATH,  
 "-r",  
 "8000",  
 "-n",  
 arbnoisepath,  
 "synth",  
 "2",  
 "brownnoise",  
 "gain",  
 "-15",  
 ]  
 )  
 subprocess.call(  
 [  
 settings.CAPTCHA\_SOX\_PATH,  
 "-m",  
 arbnoisepath,  
 path,  
 "-t",  
 "wavpcm",  
 "-b",  
 "16",  
 mergedpath,  
 ]  
 )  
 os.remove(arbnoisepath)  
 os.remove(path)  
 os.rename(mergedpath, path)  
  
 if os.path.isfile(path):  
 response = RangedFileResponse(  
 request, open(path, "rb"), content\_type="audio/wav"  
 )  
 response["Content-Disposition"] = 'attachment; filename="{}.wav"'.format(  
 key  
 )  
 return response  
 raise Http404  
  
  
def captcha\_refresh(request):  
 """Return json with new captcha for ajax refresh request"""  
 if not request.headers.get("x-requested-with") == "XMLHttpRequest":  
 raise Http404  
  
 new\_key = CaptchaStore.pick()  
 to\_json\_response = {  
 "key": new\_key,  
 "image\_url": captcha\_image\_url(new\_key),  
 "audio\_url": captcha\_audio\_url(new\_key)  
 if settings.CAPTCHA\_FLITE\_PATH  
 else None,  
 }  
 return HttpResponse(json.dumps(to\_json\_response), content\_type="application/json")

## File: venv/lib/python3.12/site-packages/django/contrib/admindocs/views.py

import inspect  
from importlib import import\_module  
from inspect import cleandoc  
from pathlib import Path  
  
from django.apps import apps  
from django.contrib import admin  
from django.contrib.admin.views.decorators import staff\_member\_required  
from django.contrib.admindocs import utils  
from django.contrib.admindocs.utils import (  
 remove\_non\_capturing\_groups,  
 replace\_metacharacters,  
 replace\_named\_groups,  
 replace\_unnamed\_groups,  
)  
from django.core.exceptions import ImproperlyConfigured, ViewDoesNotExist  
from django.db import models  
from django.http import Http404  
from django.template.engine import Engine  
from django.urls import get\_mod\_func, get\_resolver, get\_urlconf  
from django.utils.\_os import safe\_join  
from django.utils.decorators import method\_decorator  
from django.utils.functional import cached\_property  
from django.utils.inspect import (  
 func\_accepts\_kwargs,  
 func\_accepts\_var\_args,  
 get\_func\_full\_args,  
 method\_has\_no\_args,  
)  
from django.utils.translation import gettext as \_  
from django.views.generic import TemplateView  
  
from .utils import get\_view\_name  
  
# Exclude methods starting with these strings from documentation  
MODEL\_METHODS\_EXCLUDE = ("\_", "add\_", "delete", "save", "set\_")  
  
  
class BaseAdminDocsView(TemplateView):  
 """  
 Base view for admindocs views.  
 """  
  
 @method\_decorator(staff\_member\_required)  
 def dispatch(self, request, \*args, \*\*kwargs):  
 if not utils.docutils\_is\_available:  
 # Display an error message for people without docutils  
 self.template\_name = "admin\_doc/missing\_docutils.html"  
 return self.render\_to\_response(admin.site.each\_context(request))  
 return super().dispatch(request, \*args, \*\*kwargs)  
  
 def get\_context\_data(self, \*\*kwargs):  
 return super().get\_context\_data(  
 \*\*{  
 \*\*kwargs,  
 \*\*admin.site.each\_context(self.request),  
 }  
 )  
  
  
class BookmarkletsView(BaseAdminDocsView):  
 template\_name = "admin\_doc/bookmarklets.html"  
  
  
class TemplateTagIndexView(BaseAdminDocsView):  
 template\_name = "admin\_doc/template\_tag\_index.html"  
  
 def get\_context\_data(self, \*\*kwargs):  
 tags = []  
 try:  
 engine = Engine.get\_default()  
 except ImproperlyConfigured:  
 # Non-trivial TEMPLATES settings aren't supported (#24125).  
 pass  
 else:  
 app\_libs = sorted(engine.template\_libraries.items())  
 builtin\_libs = [("", lib) for lib in engine.template\_builtins]  
 for module\_name, library in builtin\_libs + app\_libs:  
 for tag\_name, tag\_func in library.tags.items():  
 title, body, metadata = utils.parse\_docstring(tag\_func.\_\_doc\_\_)  
 title = title and utils.parse\_rst(  
 title, "tag", \_("tag:") + tag\_name  
 )  
 body = body and utils.parse\_rst(body, "tag", \_("tag:") + tag\_name)  
 for key in metadata:  
 metadata[key] = utils.parse\_rst(  
 metadata[key], "tag", \_("tag:") + tag\_name  
 )  
 tag\_library = module\_name.split(".")[-1]  
 tags.append(  
 {  
 "name": tag\_name,  
 "title": title,  
 "body": body,  
 "meta": metadata,  
 "library": tag\_library,  
 }  
 )  
 return super().get\_context\_data(\*\*{\*\*kwargs, "tags": tags})  
  
  
class TemplateFilterIndexView(BaseAdminDocsView):  
 template\_name = "admin\_doc/template\_filter\_index.html"  
  
 def get\_context\_data(self, \*\*kwargs):  
 filters = []  
 try:  
 engine = Engine.get\_default()  
 except ImproperlyConfigured:  
 # Non-trivial TEMPLATES settings aren't supported (#24125).  
 pass  
 else:  
 app\_libs = sorted(engine.template\_libraries.items())  
 builtin\_libs = [("", lib) for lib in engine.template\_builtins]  
 for module\_name, library in builtin\_libs + app\_libs:  
 for filter\_name, filter\_func in library.filters.items():  
 title, body, metadata = utils.parse\_docstring(filter\_func.\_\_doc\_\_)  
 title = title and utils.parse\_rst(  
 title, "filter", \_("filter:") + filter\_name  
 )  
 body = body and utils.parse\_rst(  
 body, "filter", \_("filter:") + filter\_name  
 )  
 for key in metadata:  
 metadata[key] = utils.parse\_rst(  
 metadata[key], "filter", \_("filter:") + filter\_name  
 )  
 tag\_library = module\_name.split(".")[-1]  
 filters.append(  
 {  
 "name": filter\_name,  
 "title": title,  
 "body": body,  
 "meta": metadata,  
 "library": tag\_library,  
 }  
 )  
 return super().get\_context\_data(\*\*{\*\*kwargs, "filters": filters})  
  
  
class ViewIndexView(BaseAdminDocsView):  
 template\_name = "admin\_doc/view\_index.html"  
  
 def get\_context\_data(self, \*\*kwargs):  
 views = []  
 url\_resolver = get\_resolver(get\_urlconf())  
 try:  
 view\_functions = extract\_views\_from\_urlpatterns(url\_resolver.url\_patterns)  
 except ImproperlyConfigured:  
 view\_functions = []  
 for func, regex, namespace, name in view\_functions:  
 views.append(  
 {  
 "full\_name": get\_view\_name(func),  
 "url": simplify\_regex(regex),  
 "url\_name": ":".join((namespace or []) + (name and [name] or [])),  
 "namespace": ":".join(namespace or []),  
 "name": name,  
 }  
 )  
 return super().get\_context\_data(\*\*{\*\*kwargs, "views": views})  
  
  
class ViewDetailView(BaseAdminDocsView):  
 template\_name = "admin\_doc/view\_detail.html"  
  
 @staticmethod  
 def \_get\_view\_func(view):  
 urlconf = get\_urlconf()  
 if get\_resolver(urlconf).\_is\_callback(view):  
 mod, func = get\_mod\_func(view)  
 try:  
 # Separate the module and function, e.g.  
 # 'mymodule.views.myview' -> 'mymodule.views', 'myview').  
 return getattr(import\_module(mod), func)  
 except ImportError:  
 # Import may fail because view contains a class name, e.g.  
 # 'mymodule.views.ViewContainer.my\_view', so mod takes the form  
 # 'mymodule.views.ViewContainer'. Parse it again to separate  
 # the module and class.  
 mod, klass = get\_mod\_func(mod)  
 return getattr(getattr(import\_module(mod), klass), func)  
  
 def get\_context\_data(self, \*\*kwargs):  
 view = self.kwargs["view"]  
 view\_func = self.\_get\_view\_func(view)  
 if view\_func is None:  
 raise Http404  
 title, body, metadata = utils.parse\_docstring(view\_func.\_\_doc\_\_)  
 title = title and utils.parse\_rst(title, "view", \_("view:") + view)  
 body = body and utils.parse\_rst(body, "view", \_("view:") + view)  
 for key in metadata:  
 metadata[key] = utils.parse\_rst(metadata[key], "model", \_("view:") + view)  
 return super().get\_context\_data(  
 \*\*{  
 \*\*kwargs,  
 "name": view,  
 "summary": title,  
 "body": body,  
 "meta": metadata,  
 }  
 )  
  
  
class ModelIndexView(BaseAdminDocsView):  
 template\_name = "admin\_doc/model\_index.html"  
  
 def get\_context\_data(self, \*\*kwargs):  
 m\_list = [m.\_meta for m in apps.get\_models()]  
 return super().get\_context\_data(\*\*{\*\*kwargs, "models": m\_list})  
  
  
class ModelDetailView(BaseAdminDocsView):  
 template\_name = "admin\_doc/model\_detail.html"  
  
 def get\_context\_data(self, \*\*kwargs):  
 model\_name = self.kwargs["model\_name"]  
 # Get the model class.  
 try:  
 app\_config = apps.get\_app\_config(self.kwargs["app\_label"])  
 except LookupError:  
 raise Http404(\_("App %(app\_label)r not found") % self.kwargs)  
 try:  
 model = app\_config.get\_model(model\_name)  
 except LookupError:  
 raise Http404(  
 \_("Model %(model\_name)r not found in app %(app\_label)r") % self.kwargs  
 )  
  
 opts = model.\_meta  
  
 title, body, metadata = utils.parse\_docstring(model.\_\_doc\_\_)  
 title = title and utils.parse\_rst(title, "model", \_("model:") + model\_name)  
 body = body and utils.parse\_rst(body, "model", \_("model:") + model\_name)  
  
 # Gather fields/field descriptions.  
 fields = []  
 for field in opts.fields:  
 # ForeignKey is a special case since the field will actually be a  
 # descriptor that returns the other object  
 if isinstance(field, models.ForeignKey):  
 data\_type = field.remote\_field.model.\_\_name\_\_  
 app\_label = field.remote\_field.model.\_meta.app\_label  
 verbose = utils.parse\_rst(  
 (  
 \_("the related `%(app\_label)s.%(data\_type)s` object")  
 % {  
 "app\_label": app\_label,  
 "data\_type": data\_type,  
 }  
 ),  
 "model",  
 \_("model:") + data\_type,  
 )  
 else:  
 data\_type = get\_readable\_field\_data\_type(field)  
 verbose = field.verbose\_name  
 fields.append(  
 {  
 "name": field.name,  
 "data\_type": data\_type,  
 "verbose": verbose or "",  
 "help\_text": field.help\_text,  
 }  
 )  
  
 # Gather many-to-many fields.  
 for field in opts.many\_to\_many:  
 data\_type = field.remote\_field.model.\_\_name\_\_  
 app\_label = field.remote\_field.model.\_meta.app\_label  
 verbose = \_("related `%(app\_label)s.%(object\_name)s` objects") % {  
 "app\_label": app\_label,  
 "object\_name": data\_type,  
 }  
 fields.append(  
 {  
 "name": "%s.all" % field.name,  
 "data\_type": "List",  
 "verbose": utils.parse\_rst(  
 \_("all %s") % verbose, "model", \_("model:") + opts.model\_name  
 ),  
 }  
 )  
 fields.append(  
 {  
 "name": "%s.count" % field.name,  
 "data\_type": "Integer",  
 "verbose": utils.parse\_rst(  
 \_("number of %s") % verbose,  
 "model",  
 \_("model:") + opts.model\_name,  
 ),  
 }  
 )  
  
 methods = []  
 # Gather model methods.  
 for func\_name, func in model.\_\_dict\_\_.items():  
 if inspect.isfunction(func) or isinstance(  
 func, (cached\_property, property)  
 ):  
 try:  
 for exclude in MODEL\_METHODS\_EXCLUDE:  
 if func\_name.startswith(exclude):  
 raise StopIteration  
 except StopIteration:  
 continue  
 verbose = func.\_\_doc\_\_  
 verbose = verbose and (  
 utils.parse\_rst(  
 cleandoc(verbose), "model", \_("model:") + opts.model\_name  
 )  
 )  
 # Show properties, cached\_properties, and methods without  
 # arguments as fields. Otherwise, show as a 'method with  
 # arguments'.  
 if isinstance(func, (cached\_property, property)):  
 fields.append(  
 {  
 "name": func\_name,  
 "data\_type": get\_return\_data\_type(func\_name),  
 "verbose": verbose or "",  
 }  
 )  
 elif (  
 method\_has\_no\_args(func)  
 and not func\_accepts\_kwargs(func)  
 and not func\_accepts\_var\_args(func)  
 ):  
 fields.append(  
 {  
 "name": func\_name,  
 "data\_type": get\_return\_data\_type(func\_name),  
 "verbose": verbose or "",  
 }  
 )  
 else:  
 arguments = get\_func\_full\_args(func)  
 # Join arguments with ', ' and in case of default value,  
 # join it with '='. Use repr() so that strings will be  
 # correctly displayed.  
 print\_arguments = ", ".join(  
 [  
 "=".join([arg\_el[0], \*map(repr, arg\_el[1:])])  
 for arg\_el in arguments  
 ]  
 )  
 methods.append(  
 {  
 "name": func\_name,  
 "arguments": print\_arguments,  
 "verbose": verbose or "",  
 }  
 )  
  
 # Gather related objects  
 for rel in opts.related\_objects:  
 verbose = \_("related `%(app\_label)s.%(object\_name)s` objects") % {  
 "app\_label": rel.related\_model.\_meta.app\_label,  
 "object\_name": rel.related\_model.\_meta.object\_name,  
 }  
 accessor = rel.get\_accessor\_name()  
 fields.append(  
 {  
 "name": "%s.all" % accessor,  
 "data\_type": "List",  
 "verbose": utils.parse\_rst(  
 \_("all %s") % verbose, "model", \_("model:") + opts.model\_name  
 ),  
 }  
 )  
 fields.append(  
 {  
 "name": "%s.count" % accessor,  
 "data\_type": "Integer",  
 "verbose": utils.parse\_rst(  
 \_("number of %s") % verbose,  
 "model",  
 \_("model:") + opts.model\_name,  
 ),  
 }  
 )  
 return super().get\_context\_data(  
 \*\*{  
 \*\*kwargs,  
 "name": opts.label,  
 "summary": title,  
 "description": body,  
 "fields": fields,  
 "methods": methods,  
 }  
 )  
  
  
class TemplateDetailView(BaseAdminDocsView):  
 template\_name = "admin\_doc/template\_detail.html"  
  
 def get\_context\_data(self, \*\*kwargs):  
 template = self.kwargs["template"]  
 templates = []  
 try:  
 default\_engine = Engine.get\_default()  
 except ImproperlyConfigured:  
 # Non-trivial TEMPLATES settings aren't supported (#24125).  
 pass  
 else:  
 # This doesn't account for template loaders (#24128).  
 for index, directory in enumerate(default\_engine.dirs):  
 template\_file = Path(safe\_join(directory, template))  
 if template\_file.exists():  
 template\_contents = template\_file.read\_text()  
 else:  
 template\_contents = ""  
 templates.append(  
 {  
 "file": template\_file,  
 "exists": template\_file.exists(),  
 "contents": template\_contents,  
 "order": index,  
 }  
 )  
 return super().get\_context\_data(  
 \*\*{  
 \*\*kwargs,  
 "name": template,  
 "templates": templates,  
 }  
 )  
  
  
####################  
# Helper functions #  
####################  
  
  
def get\_return\_data\_type(func\_name):  
 """Return a somewhat-helpful data type given a function name"""  
 if func\_name.startswith("get\_"):  
 if func\_name.endswith("\_list"):  
 return "List"  
 elif func\_name.endswith("\_count"):  
 return "Integer"  
 return ""  
  
  
def get\_readable\_field\_data\_type(field):  
 """  
 Return the description for a given field type, if it exists. Fields'  
 descriptions can contain format strings, which will be interpolated with  
 the values of field.\_\_dict\_\_ before being output.  
 """  
 return field.description % field.\_\_dict\_\_  
  
  
def extract\_views\_from\_urlpatterns(urlpatterns, base="", namespace=None):  
 """  
 Return a list of views from a list of urlpatterns.  
  
 Each object in the returned list is a four-tuple:  
 (view\_func, regex, namespace, name)  
 """  
 views = []  
 for p in urlpatterns:  
 if hasattr(p, "url\_patterns"):  
 try:  
 patterns = p.url\_patterns  
 except ImportError:  
 continue  
 views.extend(  
 extract\_views\_from\_urlpatterns(  
 patterns,  
 base + str(p.pattern),  
 (namespace or []) + (p.namespace and [p.namespace] or []),  
 )  
 )  
 elif hasattr(p, "callback"):  
 try:  
 views.append((p.callback, base + str(p.pattern), namespace, p.name))  
 except ViewDoesNotExist:  
 continue  
 else:  
 raise TypeError(\_("%s does not appear to be a urlpattern object") % p)  
 return views  
  
  
def simplify\_regex(pattern):  
 r"""  
 Clean up urlpattern regexes into something more readable by humans. For  
 example, turn "^(?P<sport\_slug>\w+)/athletes/(?P<athlete\_slug>\w+)/$"  
 into "/<sport\_slug>/athletes/<athlete\_slug>/".  
 """  
 pattern = remove\_non\_capturing\_groups(pattern)  
 pattern = replace\_named\_groups(pattern)  
 pattern = replace\_unnamed\_groups(pattern)  
 pattern = replace\_metacharacters(pattern)  
 if not pattern.startswith("/"):  
 pattern = "/" + pattern  
 return pattern

## File: venv/lib/python3.12/site-packages/django/contrib/auth/views.py

import warnings  
from urllib.parse import urlparse, urlunparse  
  
from django.conf import settings  
  
# Avoid shadowing the login() and logout() views below.  
from django.contrib.auth import REDIRECT\_FIELD\_NAME, get\_user\_model  
from django.contrib.auth import login as auth\_login  
from django.contrib.auth import logout as auth\_logout  
from django.contrib.auth import update\_session\_auth\_hash  
from django.contrib.auth.decorators import login\_required  
from django.contrib.auth.forms import (  
 AuthenticationForm,  
 PasswordChangeForm,  
 PasswordResetForm,  
 SetPasswordForm,  
)  
from django.contrib.auth.tokens import default\_token\_generator  
from django.contrib.sites.shortcuts import get\_current\_site  
from django.core.exceptions import ImproperlyConfigured, ValidationError  
from django.http import HttpResponseRedirect, QueryDict  
from django.shortcuts import resolve\_url  
from django.urls import reverse\_lazy  
from django.utils.decorators import method\_decorator  
from django.utils.deprecation import RemovedInDjango50Warning  
from django.utils.http import url\_has\_allowed\_host\_and\_scheme, urlsafe\_base64\_decode  
from django.utils.translation import gettext\_lazy as \_  
from django.views.decorators.cache import never\_cache  
from django.views.decorators.csrf import csrf\_protect  
from django.views.decorators.debug import sensitive\_post\_parameters  
from django.views.generic.base import TemplateView  
from django.views.generic.edit import FormView  
  
UserModel = get\_user\_model()  
  
  
class RedirectURLMixin:  
 next\_page = None  
 redirect\_field\_name = REDIRECT\_FIELD\_NAME  
 success\_url\_allowed\_hosts = set()  
  
 def get\_success\_url(self):  
 return self.get\_redirect\_url() or self.get\_default\_redirect\_url()  
  
 def get\_redirect\_url(self):  
 """Return the user-originating redirect URL if it's safe."""  
 redirect\_to = self.request.POST.get(  
 self.redirect\_field\_name, self.request.GET.get(self.redirect\_field\_name)  
 )  
 url\_is\_safe = url\_has\_allowed\_host\_and\_scheme(  
 url=redirect\_to,  
 allowed\_hosts=self.get\_success\_url\_allowed\_hosts(),  
 require\_https=self.request.is\_secure(),  
 )  
 return redirect\_to if url\_is\_safe else ""  
  
 def get\_success\_url\_allowed\_hosts(self):  
 return {self.request.get\_host(), \*self.success\_url\_allowed\_hosts}  
  
 def get\_default\_redirect\_url(self):  
 """Return the default redirect URL."""  
 if self.next\_page:  
 return resolve\_url(self.next\_page)  
 raise ImproperlyConfigured("No URL to redirect to. Provide a next\_page.")  
  
  
class LoginView(RedirectURLMixin, FormView):  
 """  
 Display the login form and handle the login action.  
 """  
  
 form\_class = AuthenticationForm  
 authentication\_form = None  
 template\_name = "registration/login.html"  
 redirect\_authenticated\_user = False  
 extra\_context = None  
  
 @method\_decorator(sensitive\_post\_parameters())  
 @method\_decorator(csrf\_protect)  
 @method\_decorator(never\_cache)  
 def dispatch(self, request, \*args, \*\*kwargs):  
 if self.redirect\_authenticated\_user and self.request.user.is\_authenticated:  
 redirect\_to = self.get\_success\_url()  
 if redirect\_to == self.request.path:  
 raise ValueError(  
 "Redirection loop for authenticated user detected. Check that "  
 "your LOGIN\_REDIRECT\_URL doesn't point to a login page."  
 )  
 return HttpResponseRedirect(redirect\_to)  
 return super().dispatch(request, \*args, \*\*kwargs)  
  
 def get\_default\_redirect\_url(self):  
 """Return the default redirect URL."""  
 if self.next\_page:  
 return resolve\_url(self.next\_page)  
 else:  
 return resolve\_url(settings.LOGIN\_REDIRECT\_URL)  
  
 def get\_form\_class(self):  
 return self.authentication\_form or self.form\_class  
  
 def get\_form\_kwargs(self):  
 kwargs = super().get\_form\_kwargs()  
 kwargs["request"] = self.request  
 return kwargs  
  
 def form\_valid(self, form):  
 """Security check complete. Log the user in."""  
 auth\_login(self.request, form.get\_user())  
 return HttpResponseRedirect(self.get\_success\_url())  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 current\_site = get\_current\_site(self.request)  
 context.update(  
 {  
 self.redirect\_field\_name: self.get\_redirect\_url(),  
 "site": current\_site,  
 "site\_name": current\_site.name,  
 \*\*(self.extra\_context or {}),  
 }  
 )  
 return context  
  
  
class LogoutView(RedirectURLMixin, TemplateView):  
 """  
 Log out the user and display the 'You are logged out' message.  
 """  
  
 # RemovedInDjango50Warning: when the deprecation ends, remove "get" and  
 # "head" from http\_method\_names.  
 http\_method\_names = ["get", "head", "post", "options"]  
 template\_name = "registration/logged\_out.html"  
 extra\_context = None  
  
 # RemovedInDjango50Warning: when the deprecation ends, move  
 # @method\_decorator(csrf\_protect) from post() to dispatch().  
 @method\_decorator(never\_cache)  
 def dispatch(self, request, \*args, \*\*kwargs):  
 if request.method.lower() == "get":  
 warnings.warn(  
 "Log out via GET requests is deprecated and will be removed in Django "  
 "5.0. Use POST requests for logging out.",  
 RemovedInDjango50Warning,  
 )  
 return super().dispatch(request, \*args, \*\*kwargs)  
  
 @method\_decorator(csrf\_protect)  
 def post(self, request, \*args, \*\*kwargs):  
 """Logout may be done via POST."""  
 auth\_logout(request)  
 redirect\_to = self.get\_success\_url()  
 if redirect\_to != request.get\_full\_path():  
 # Redirect to target page once the session has been cleared.  
 return HttpResponseRedirect(redirect\_to)  
 return super().get(request, \*args, \*\*kwargs)  
  
 # RemovedInDjango50Warning.  
 get = post  
  
 def get\_default\_redirect\_url(self):  
 """Return the default redirect URL."""  
 if self.next\_page:  
 return resolve\_url(self.next\_page)  
 elif settings.LOGOUT\_REDIRECT\_URL:  
 return resolve\_url(settings.LOGOUT\_REDIRECT\_URL)  
 else:  
 return self.request.path  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 current\_site = get\_current\_site(self.request)  
 context.update(  
 {  
 "site": current\_site,  
 "site\_name": current\_site.name,  
 "title": \_("Logged out"),  
 "subtitle": None,  
 \*\*(self.extra\_context or {}),  
 }  
 )  
 return context  
  
  
def logout\_then\_login(request, login\_url=None):  
 """  
 Log out the user if they are logged in. Then redirect to the login page.  
 """  
 login\_url = resolve\_url(login\_url or settings.LOGIN\_URL)  
 return LogoutView.as\_view(next\_page=login\_url)(request)  
  
  
def redirect\_to\_login(next, login\_url=None, redirect\_field\_name=REDIRECT\_FIELD\_NAME):  
 """  
 Redirect the user to the login page, passing the given 'next' page.  
 """  
 resolved\_url = resolve\_url(login\_url or settings.LOGIN\_URL)  
  
 login\_url\_parts = list(urlparse(resolved\_url))  
 if redirect\_field\_name:  
 querystring = QueryDict(login\_url\_parts[4], mutable=True)  
 querystring[redirect\_field\_name] = next  
 login\_url\_parts[4] = querystring.urlencode(safe="/")  
  
 return HttpResponseRedirect(urlunparse(login\_url\_parts))  
  
  
# Class-based password reset views  
# - PasswordResetView sends the mail  
# - PasswordResetDoneView shows a success message for the above  
# - PasswordResetConfirmView checks the link the user clicked and  
# prompts for a new password  
# - PasswordResetCompleteView shows a success message for the above  
  
  
class PasswordContextMixin:  
 extra\_context = None  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 context.update(  
 {"title": self.title, "subtitle": None, \*\*(self.extra\_context or {})}  
 )  
 return context  
  
  
class PasswordResetView(PasswordContextMixin, FormView):  
 email\_template\_name = "registration/password\_reset\_email.html"  
 extra\_email\_context = None  
 form\_class = PasswordResetForm  
 from\_email = None  
 html\_email\_template\_name = None  
 subject\_template\_name = "registration/password\_reset\_subject.txt"  
 success\_url = reverse\_lazy("password\_reset\_done")  
 template\_name = "registration/password\_reset\_form.html"  
 title = \_("Password reset")  
 token\_generator = default\_token\_generator  
  
 @method\_decorator(csrf\_protect)  
 def dispatch(self, \*args, \*\*kwargs):  
 return super().dispatch(\*args, \*\*kwargs)  
  
 def form\_valid(self, form):  
 opts = {  
 "use\_https": self.request.is\_secure(),  
 "token\_generator": self.token\_generator,  
 "from\_email": self.from\_email,  
 "email\_template\_name": self.email\_template\_name,  
 "subject\_template\_name": self.subject\_template\_name,  
 "request": self.request,  
 "html\_email\_template\_name": self.html\_email\_template\_name,  
 "extra\_email\_context": self.extra\_email\_context,  
 }  
 form.save(\*\*opts)  
 return super().form\_valid(form)  
  
  
INTERNAL\_RESET\_SESSION\_TOKEN = "\_password\_reset\_token"  
  
  
class PasswordResetDoneView(PasswordContextMixin, TemplateView):  
 template\_name = "registration/password\_reset\_done.html"  
 title = \_("Password reset sent")  
  
  
class PasswordResetConfirmView(PasswordContextMixin, FormView):  
 form\_class = SetPasswordForm  
 post\_reset\_login = False  
 post\_reset\_login\_backend = None  
 reset\_url\_token = "set-password"  
 success\_url = reverse\_lazy("password\_reset\_complete")  
 template\_name = "registration/password\_reset\_confirm.html"  
 title = \_("Enter new password")  
 token\_generator = default\_token\_generator  
  
 @method\_decorator(sensitive\_post\_parameters())  
 @method\_decorator(never\_cache)  
 def dispatch(self, \*args, \*\*kwargs):  
 if "uidb64" not in kwargs or "token" not in kwargs:  
 raise ImproperlyConfigured(  
 "The URL path must contain 'uidb64' and 'token' parameters."  
 )  
  
 self.validlink = False  
 self.user = self.get\_user(kwargs["uidb64"])  
  
 if self.user is not None:  
 token = kwargs["token"]  
 if token == self.reset\_url\_token:  
 session\_token = self.request.session.get(INTERNAL\_RESET\_SESSION\_TOKEN)  
 if self.token\_generator.check\_token(self.user, session\_token):  
 # If the token is valid, display the password reset form.  
 self.validlink = True  
 return super().dispatch(\*args, \*\*kwargs)  
 else:  
 if self.token\_generator.check\_token(self.user, token):  
 # Store the token in the session and redirect to the  
 # password reset form at a URL without the token. That  
 # avoids the possibility of leaking the token in the  
 # HTTP Referer header.  
 self.request.session[INTERNAL\_RESET\_SESSION\_TOKEN] = token  
 redirect\_url = self.request.path.replace(  
 token, self.reset\_url\_token  
 )  
 return HttpResponseRedirect(redirect\_url)  
  
 # Display the "Password reset unsuccessful" page.  
 return self.render\_to\_response(self.get\_context\_data())  
  
 def get\_user(self, uidb64):  
 try:  
 # urlsafe\_base64\_decode() decodes to bytestring  
 uid = urlsafe\_base64\_decode(uidb64).decode()  
 user = UserModel.\_default\_manager.get(pk=uid)  
 except (  
 TypeError,  
 ValueError,  
 OverflowError,  
 UserModel.DoesNotExist,  
 ValidationError,  
 ):  
 user = None  
 return user  
  
 def get\_form\_kwargs(self):  
 kwargs = super().get\_form\_kwargs()  
 kwargs["user"] = self.user  
 return kwargs  
  
 def form\_valid(self, form):  
 user = form.save()  
 del self.request.session[INTERNAL\_RESET\_SESSION\_TOKEN]  
 if self.post\_reset\_login:  
 auth\_login(self.request, user, self.post\_reset\_login\_backend)  
 return super().form\_valid(form)  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 if self.validlink:  
 context["validlink"] = True  
 else:  
 context.update(  
 {  
 "form": None,  
 "title": \_("Password reset unsuccessful"),  
 "validlink": False,  
 }  
 )  
 return context  
  
  
class PasswordResetCompleteView(PasswordContextMixin, TemplateView):  
 template\_name = "registration/password\_reset\_complete.html"  
 title = \_("Password reset complete")  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 context["login\_url"] = resolve\_url(settings.LOGIN\_URL)  
 return context  
  
  
class PasswordChangeView(PasswordContextMixin, FormView):  
 form\_class = PasswordChangeForm  
 success\_url = reverse\_lazy("password\_change\_done")  
 template\_name = "registration/password\_change\_form.html"  
 title = \_("Password change")  
  
 @method\_decorator(sensitive\_post\_parameters())  
 @method\_decorator(csrf\_protect)  
 @method\_decorator(login\_required)  
 def dispatch(self, \*args, \*\*kwargs):  
 return super().dispatch(\*args, \*\*kwargs)  
  
 def get\_form\_kwargs(self):  
 kwargs = super().get\_form\_kwargs()  
 kwargs["user"] = self.request.user  
 return kwargs  
  
 def form\_valid(self, form):  
 form.save()  
 # Updating the password logs out all other sessions for the user  
 # except the current one.  
 update\_session\_auth\_hash(self.request, form.user)  
 return super().form\_valid(form)  
  
  
class PasswordChangeDoneView(PasswordContextMixin, TemplateView):  
 template\_name = "registration/password\_change\_done.html"  
 title = \_("Password change successful")  
  
 @method\_decorator(login\_required)  
 def dispatch(self, \*args, \*\*kwargs):  
 return super().dispatch(\*args, \*\*kwargs)

## File: venv/lib/python3.12/site-packages/django/contrib/contenttypes/views.py

from django.apps import apps  
from django.contrib.contenttypes.models import ContentType  
from django.contrib.sites.shortcuts import get\_current\_site  
from django.core.exceptions import ObjectDoesNotExist  
from django.http import Http404, HttpResponseRedirect  
from django.utils.translation import gettext as \_  
  
  
def shortcut(request, content\_type\_id, object\_id):  
 """  
 Redirect to an object's page based on a content-type ID and an object ID.  
 """  
 # Look up the object, making sure it's got a get\_absolute\_url() function.  
 try:  
 content\_type = ContentType.objects.get(pk=content\_type\_id)  
 if not content\_type.model\_class():  
 raise Http404(  
 \_("Content type %(ct\_id)s object has no associated model")  
 % {"ct\_id": content\_type\_id}  
 )  
 obj = content\_type.get\_object\_for\_this\_type(pk=object\_id)  
 except (ObjectDoesNotExist, ValueError):  
 raise Http404(  
 \_("Content type %(ct\_id)s object %(obj\_id)s doesn’t exist")  
 % {"ct\_id": content\_type\_id, "obj\_id": object\_id}  
 )  
  
 try:  
 get\_absolute\_url = obj.get\_absolute\_url  
 except AttributeError:  
 raise Http404(  
 \_("%(ct\_name)s objects don’t have a get\_absolute\_url() method")  
 % {"ct\_name": content\_type.name}  
 )  
 absurl = get\_absolute\_url()  
  
 # Try to figure out the object's domain, so we can do a cross-site redirect  
 # if necessary.  
  
 # If the object actually defines a domain, we're done.  
 if absurl.startswith(("http://", "https://", "//")):  
 return HttpResponseRedirect(absurl)  
  
 # Otherwise, we need to introspect the object's relationships for a  
 # relation to the Site object  
 try:  
 object\_domain = get\_current\_site(request).domain  
 except ObjectDoesNotExist:  
 object\_domain = None  
  
 if apps.is\_installed("django.contrib.sites"):  
 Site = apps.get\_model("sites.Site")  
 opts = obj.\_meta  
  
 for field in opts.many\_to\_many:  
 # Look for a many-to-many relationship to Site.  
 if field.remote\_field.model is Site:  
 site\_qs = getattr(obj, field.name).all()  
 if object\_domain and site\_qs.filter(domain=object\_domain).exists():  
 # The current site's domain matches a site attached to the  
 # object.  
 break  
 # Caveat: In the case of multiple related Sites, this just  
 # selects the \*first\* one, which is arbitrary.  
 site = site\_qs.first()  
 if site:  
 object\_domain = site.domain  
 break  
 else:  
 # No many-to-many relationship to Site found. Look for a  
 # many-to-one relationship to Site.  
 for field in obj.\_meta.fields:  
 if field.remote\_field and field.remote\_field.model is Site:  
 try:  
 site = getattr(obj, field.name)  
 except Site.DoesNotExist:  
 continue  
 if site is not None:  
 object\_domain = site.domain  
 break  
  
 # If all that malarkey found an object domain, use it. Otherwise, fall back  
 # to whatever get\_absolute\_url() returned.  
 if object\_domain is not None:  
 protocol = request.scheme  
 return HttpResponseRedirect("%s://%s%s" % (protocol, object\_domain, absurl))  
 else:  
 return HttpResponseRedirect(absurl)

## File: venv/lib/python3.12/site-packages/django/contrib/flatpages/views.py

from django.conf import settings  
from django.contrib.flatpages.models import FlatPage  
from django.contrib.sites.shortcuts import get\_current\_site  
from django.http import Http404, HttpResponse, HttpResponsePermanentRedirect  
from django.shortcuts import get\_object\_or\_404  
from django.template import loader  
from django.utils.safestring import mark\_safe  
from django.views.decorators.csrf import csrf\_protect  
  
DEFAULT\_TEMPLATE = "flatpages/default.html"  
  
# This view is called from FlatpageFallbackMiddleware.process\_response  
# when a 404 is raised, which often means CsrfViewMiddleware.process\_view  
# has not been called even if CsrfViewMiddleware is installed. So we need  
# to use @csrf\_protect, in case the template needs {% csrf\_token %}.  
# However, we can't just wrap this view; if no matching flatpage exists,  
# or a redirect is required for authentication, the 404 needs to be returned  
# without any CSRF checks. Therefore, we only  
# CSRF protect the internal implementation.  
  
  
def flatpage(request, url):  
 """  
 Public interface to the flat page view.  
  
 Models: `flatpages.flatpages`  
 Templates: Uses the template defined by the ``template\_name`` field,  
 or :template:`flatpages/default.html` if template\_name is not defined.  
 Context:  
 flatpage  
 `flatpages.flatpages` object  
 """  
 if not url.startswith("/"):  
 url = "/" + url  
 site\_id = get\_current\_site(request).id  
 try:  
 f = get\_object\_or\_404(FlatPage, url=url, sites=site\_id)  
 except Http404:  
 if not url.endswith("/") and settings.APPEND\_SLASH:  
 url += "/"  
 f = get\_object\_or\_404(FlatPage, url=url, sites=site\_id)  
 return HttpResponsePermanentRedirect("%s/" % request.path)  
 else:  
 raise  
 return render\_flatpage(request, f)  
  
  
@csrf\_protect  
def render\_flatpage(request, f):  
 """  
 Internal interface to the flat page view.  
 """  
 # If registration is required for accessing this page, and the user isn't  
 # logged in, redirect to the login page.  
 if f.registration\_required and not request.user.is\_authenticated:  
 from django.contrib.auth.views import redirect\_to\_login  
  
 return redirect\_to\_login(request.path)  
 if f.template\_name:  
 template = loader.select\_template((f.template\_name, DEFAULT\_TEMPLATE))  
 else:  
 template = loader.get\_template(DEFAULT\_TEMPLATE)  
  
 # To avoid having to always use the "|safe" filter in flatpage templates,  
 # mark the title and content as already safe (since they are raw HTML  
 # content in the first place).  
 f.title = mark\_safe(f.title)  
 f.content = mark\_safe(f.content)  
  
 return HttpResponse(template.render({"flatpage": f}, request))

## File: venv/lib/python3.12/site-packages/django/contrib/gis/sitemaps/views.py

from django.apps import apps  
from django.contrib.gis.db.models import GeometryField  
from django.contrib.gis.db.models.functions import AsKML, Transform  
from django.contrib.gis.shortcuts import render\_to\_kml, render\_to\_kmz  
from django.core.exceptions import FieldDoesNotExist  
from django.db import DEFAULT\_DB\_ALIAS, connections  
from django.http import Http404  
  
  
def kml(request, label, model, field\_name=None, compress=False, using=DEFAULT\_DB\_ALIAS):  
 """  
 This view generates KML for the given app label, model, and field name.  
  
 The field name must be that of a geographic field.  
 """  
 placemarks = []  
 try:  
 klass = apps.get\_model(label, model)  
 except LookupError:  
 raise Http404(  
 'You must supply a valid app label and module name. Got "%s.%s"'  
 % (label, model)  
 )  
  
 if field\_name:  
 try:  
 field = klass.\_meta.get\_field(field\_name)  
 if not isinstance(field, GeometryField):  
 raise FieldDoesNotExist  
 except FieldDoesNotExist:  
 raise Http404("Invalid geometry field.")  
  
 connection = connections[using]  
  
 if connection.features.has\_AsKML\_function:  
 # Database will take care of transformation.  
 placemarks = klass.\_default\_manager.using(using).annotate(kml=AsKML(field\_name))  
 else:  
 # If the database offers no KML method, we use the `kml`  
 # attribute of the lazy geometry instead.  
 placemarks = []  
 if connection.features.has\_Transform\_function:  
 qs = klass.\_default\_manager.using(using).annotate(  
 \*\*{"%s\_4326" % field\_name: Transform(field\_name, 4326)}  
 )  
 field\_name += "\_4326"  
 else:  
 qs = klass.\_default\_manager.using(using).all()  
 for mod in qs:  
 mod.kml = getattr(mod, field\_name).kml  
 placemarks.append(mod)  
  
 # Getting the render function and rendering to the correct.  
 if compress:  
 render = render\_to\_kmz  
 else:  
 render = render\_to\_kml  
 return render("gis/kml/placemarks.kml", {"places": placemarks})  
  
  
def kmz(request, label, model, field\_name=None, using=DEFAULT\_DB\_ALIAS):  
 """  
 Return KMZ for the given app label, model, and field name.  
 """  
 return kml(request, label, model, field\_name, compress=True, using=using)

## File: venv/lib/python3.12/site-packages/django/contrib/gis/views.py

from django.http import Http404  
from django.utils.translation import gettext as \_  
  
  
def feed(request, url, feed\_dict=None):  
 """Provided for backwards compatibility."""  
 if not feed\_dict:  
 raise Http404(\_("No feeds are registered."))  
  
 slug = url.partition("/")[0]  
 try:  
 f = feed\_dict[slug]  
 except KeyError:  
 raise Http404(\_("Slug %r isn’t registered.") % slug)  
  
 instance = f()  
 instance.feed\_url = getattr(f, "feed\_url", None) or request.path  
 instance.title\_template = f.title\_template or ("feeds/%s\_title.html" % slug)  
 instance.description\_template = f.description\_template or (  
 "feeds/%s\_description.html" % slug  
 )  
 return instance(request)

## File: venv/lib/python3.12/site-packages/django/contrib/messages/views.py

from django.contrib import messages  
  
  
class SuccessMessageMixin:  
 """  
 Add a success message on successful form submission.  
 """  
  
 success\_message = ""  
  
 def form\_valid(self, form):  
 response = super().form\_valid(form)  
 success\_message = self.get\_success\_message(form.cleaned\_data)  
 if success\_message:  
 messages.success(self.request, success\_message)  
 return response  
  
 def get\_success\_message(self, cleaned\_data):  
 return self.success\_message % cleaned\_data

## File: venv/lib/python3.12/site-packages/django/contrib/sitemaps/views.py

import datetime  
import warnings  
from dataclasses import dataclass  
from functools import wraps  
  
from django.contrib.sites.shortcuts import get\_current\_site  
from django.core.paginator import EmptyPage, PageNotAnInteger  
from django.http import Http404  
from django.template.response import TemplateResponse  
from django.urls import reverse  
from django.utils import timezone  
from django.utils.deprecation import RemovedInDjango50Warning  
from django.utils.http import http\_date  
  
  
@dataclass  
class SitemapIndexItem:  
 location: str  
 last\_mod: bool = None  
  
 # RemovedInDjango50Warning  
 def \_\_str\_\_(self):  
 msg = (  
 "Calling `\_\_str\_\_` on SitemapIndexItem is deprecated, use the `location` "  
 "attribute instead."  
 )  
 warnings.warn(msg, RemovedInDjango50Warning, stacklevel=2)  
 return self.location  
  
  
def x\_robots\_tag(func):  
 @wraps(func)  
 def inner(request, \*args, \*\*kwargs):  
 response = func(request, \*args, \*\*kwargs)  
 response.headers["X-Robots-Tag"] = "noindex, noodp, noarchive"  
 return response  
  
 return inner  
  
  
def \_get\_latest\_lastmod(current\_lastmod, new\_lastmod):  
 """  
 Returns the latest `lastmod` where `lastmod` can be either a date or a  
 datetime.  
 """  
 if not isinstance(new\_lastmod, datetime.datetime):  
 new\_lastmod = datetime.datetime.combine(new\_lastmod, datetime.time.min)  
 if timezone.is\_naive(new\_lastmod):  
 new\_lastmod = timezone.make\_aware(new\_lastmod, datetime.timezone.utc)  
 return new\_lastmod if current\_lastmod is None else max(current\_lastmod, new\_lastmod)  
  
  
@x\_robots\_tag  
def index(  
 request,  
 sitemaps,  
 template\_name="sitemap\_index.xml",  
 content\_type="application/xml",  
 sitemap\_url\_name="django.contrib.sitemaps.views.sitemap",  
):  
 req\_protocol = request.scheme  
 req\_site = get\_current\_site(request)  
  
 sites = [] # all sections' sitemap URLs  
 all\_indexes\_lastmod = True  
 latest\_lastmod = None  
 for section, site in sitemaps.items():  
 # For each section label, add links of all pages of its sitemap  
 # (usually generated by the `sitemap` view).  
 if callable(site):  
 site = site()  
 protocol = req\_protocol if site.protocol is None else site.protocol  
 sitemap\_url = reverse(sitemap\_url\_name, kwargs={"section": section})  
 absolute\_url = "%s://%s%s" % (protocol, req\_site.domain, sitemap\_url)  
 site\_lastmod = site.get\_latest\_lastmod()  
 if all\_indexes\_lastmod:  
 if site\_lastmod is not None:  
 latest\_lastmod = \_get\_latest\_lastmod(latest\_lastmod, site\_lastmod)  
 else:  
 all\_indexes\_lastmod = False  
 sites.append(SitemapIndexItem(absolute\_url, site\_lastmod))  
 # Add links to all pages of the sitemap.  
 for page in range(2, site.paginator.num\_pages + 1):  
 sites.append(  
 SitemapIndexItem("%s?p=%s" % (absolute\_url, page), site\_lastmod)  
 )  
 # If lastmod is defined for all sites, set header so as  
 # ConditionalGetMiddleware is able to send 304 NOT MODIFIED  
 if all\_indexes\_lastmod and latest\_lastmod:  
 headers = {"Last-Modified": http\_date(latest\_lastmod.timestamp())}  
 else:  
 headers = None  
 return TemplateResponse(  
 request,  
 template\_name,  
 {"sitemaps": sites},  
 content\_type=content\_type,  
 headers=headers,  
 )  
  
  
@x\_robots\_tag  
def sitemap(  
 request,  
 sitemaps,  
 section=None,  
 template\_name="sitemap.xml",  
 content\_type="application/xml",  
):  
 req\_protocol = request.scheme  
 req\_site = get\_current\_site(request)  
  
 if section is not None:  
 if section not in sitemaps:  
 raise Http404("No sitemap available for section: %r" % section)  
 maps = [sitemaps[section]]  
 else:  
 maps = sitemaps.values()  
 page = request.GET.get("p", 1)  
  
 lastmod = None  
 all\_sites\_lastmod = True  
 urls = []  
 for site in maps:  
 try:  
 if callable(site):  
 site = site()  
 urls.extend(site.get\_urls(page=page, site=req\_site, protocol=req\_protocol))  
 if all\_sites\_lastmod:  
 site\_lastmod = getattr(site, "latest\_lastmod", None)  
 if site\_lastmod is not None:  
 lastmod = \_get\_latest\_lastmod(lastmod, site\_lastmod)  
 else:  
 all\_sites\_lastmod = False  
 except EmptyPage:  
 raise Http404("Page %s empty" % page)  
 except PageNotAnInteger:  
 raise Http404("No page '%s'" % page)  
 # If lastmod is defined for all sites, set header so as  
 # ConditionalGetMiddleware is able to send 304 NOT MODIFIED  
 if all\_sites\_lastmod:  
 headers = {"Last-Modified": http\_date(lastmod.timestamp())} if lastmod else None  
 else:  
 headers = None  
 return TemplateResponse(  
 request,  
 template\_name,  
 {"urlset": urls},  
 content\_type=content\_type,  
 headers=headers,  
 )

## File: venv/lib/python3.12/site-packages/django/contrib/staticfiles/views.py

"""  
Views and functions for serving static files. These are only to be used during  
development, and SHOULD NOT be used in a production setting.  
  
"""  
import os  
import posixpath  
  
from django.conf import settings  
from django.contrib.staticfiles import finders  
from django.http import Http404  
from django.views import static  
  
  
def serve(request, path, insecure=False, \*\*kwargs):  
 """  
 Serve static files below a given point in the directory structure or  
 from locations inferred from the staticfiles finders.  
  
 To use, put a URL pattern such as::  
  
 from django.contrib.staticfiles import views  
  
 path('<path:path>', views.serve)  
  
 in your URLconf.  
  
 It uses the django.views.static.serve() view to serve the found files.  
 """  
 if not settings.DEBUG and not insecure:  
 raise Http404  
 normalized\_path = posixpath.normpath(path).lstrip("/")  
 absolute\_path = finders.find(normalized\_path)  
 if not absolute\_path:  
 if path.endswith("/") or path == "":  
 raise Http404("Directory indexes are not allowed here.")  
 raise Http404("'%s' could not be found" % path)  
 document\_root, path = os.path.split(absolute\_path)  
 return static.serve(request, path, document\_root=document\_root, \*\*kwargs)

## File: venv/lib/python3.12/site-packages/django/contrib/syndication/views.py

from inspect import getattr\_static, unwrap  
  
from django.contrib.sites.shortcuts import get\_current\_site  
from django.core.exceptions import ImproperlyConfigured, ObjectDoesNotExist  
from django.http import Http404, HttpResponse  
from django.template import TemplateDoesNotExist, loader  
from django.utils import feedgenerator  
from django.utils.encoding import iri\_to\_uri  
from django.utils.html import escape  
from django.utils.http import http\_date  
from django.utils.timezone import get\_default\_timezone, is\_naive, make\_aware  
from django.utils.translation import get\_language  
  
  
def add\_domain(domain, url, secure=False):  
 protocol = "https" if secure else "http"  
 if url.startswith("//"):  
 # Support network-path reference (see #16753) - RSS requires a protocol  
 url = "%s:%s" % (protocol, url)  
 elif not url.startswith(("http://", "https://", "mailto:")):  
 url = iri\_to\_uri("%s://%s%s" % (protocol, domain, url))  
 return url  
  
  
class FeedDoesNotExist(ObjectDoesNotExist):  
 pass  
  
  
class Feed:  
 feed\_type = feedgenerator.DefaultFeed  
 title\_template = None  
 description\_template = None  
 language = None  
  
 def \_\_call\_\_(self, request, \*args, \*\*kwargs):  
 try:  
 obj = self.get\_object(request, \*args, \*\*kwargs)  
 except ObjectDoesNotExist:  
 raise Http404("Feed object does not exist.")  
 feedgen = self.get\_feed(obj, request)  
 response = HttpResponse(content\_type=feedgen.content\_type)  
 if hasattr(self, "item\_pubdate") or hasattr(self, "item\_updateddate"):  
 # if item\_pubdate or item\_updateddate is defined for the feed, set  
 # header so as ConditionalGetMiddleware is able to send 304 NOT MODIFIED  
 response.headers["Last-Modified"] = http\_date(  
 feedgen.latest\_post\_date().timestamp()  
 )  
 feedgen.write(response, "utf-8")  
 return response  
  
 def item\_title(self, item):  
 # Titles should be double escaped by default (see #6533)  
 return escape(str(item))  
  
 def item\_description(self, item):  
 return str(item)  
  
 def item\_link(self, item):  
 try:  
 return item.get\_absolute\_url()  
 except AttributeError:  
 raise ImproperlyConfigured(  
 "Give your %s class a get\_absolute\_url() method, or define an "  
 "item\_link() method in your Feed class." % item.\_\_class\_\_.\_\_name\_\_  
 )  
  
 def item\_enclosures(self, item):  
 enc\_url = self.\_get\_dynamic\_attr("item\_enclosure\_url", item)  
 if enc\_url:  
 enc = feedgenerator.Enclosure(  
 url=str(enc\_url),  
 length=str(self.\_get\_dynamic\_attr("item\_enclosure\_length", item)),  
 mime\_type=str(self.\_get\_dynamic\_attr("item\_enclosure\_mime\_type", item)),  
 )  
 return [enc]  
 return []  
  
 def \_get\_dynamic\_attr(self, attname, obj, default=None):  
 try:  
 attr = getattr(self, attname)  
 except AttributeError:  
 return default  
 if callable(attr):  
 # Check co\_argcount rather than try/excepting the function and  
 # catching the TypeError, because something inside the function  
 # may raise the TypeError. This technique is more accurate.  
 func = unwrap(attr)  
 try:  
 code = func.\_\_code\_\_  
 except AttributeError:  
 func = unwrap(attr.\_\_call\_\_)  
 code = func.\_\_code\_\_  
 # If function doesn't have arguments and it is not a static method,  
 # it was decorated without using @functools.wraps.  
 if not code.co\_argcount and not isinstance(  
 getattr\_static(self, func.\_\_name\_\_, None), staticmethod  
 ):  
 raise ImproperlyConfigured(  
 f"Feed method {attname!r} decorated by {func.\_\_name\_\_!r} needs to "  
 f"use @functools.wraps."  
 )  
 if code.co\_argcount == 2: # one argument is 'self'  
 return attr(obj)  
 else:  
 return attr()  
 return attr  
  
 def feed\_extra\_kwargs(self, obj):  
 """  
 Return an extra keyword arguments dictionary that is used when  
 initializing the feed generator.  
 """  
 return {}  
  
 def item\_extra\_kwargs(self, item):  
 """  
 Return an extra keyword arguments dictionary that is used with  
 the `add\_item` call of the feed generator.  
 """  
 return {}  
  
 def get\_object(self, request, \*args, \*\*kwargs):  
 return None  
  
 def get\_context\_data(self, \*\*kwargs):  
 """  
 Return a dictionary to use as extra context if either  
 ``self.description\_template`` or ``self.item\_template`` are used.  
  
 Default implementation preserves the old behavior  
 of using {'obj': item, 'site': current\_site} as the context.  
 """  
 return {"obj": kwargs.get("item"), "site": kwargs.get("site")}  
  
 def get\_feed(self, obj, request):  
 """  
 Return a feedgenerator.DefaultFeed object, fully populated, for  
 this feed. Raise FeedDoesNotExist for invalid parameters.  
 """  
 current\_site = get\_current\_site(request)  
  
 link = self.\_get\_dynamic\_attr("link", obj)  
 link = add\_domain(current\_site.domain, link, request.is\_secure())  
  
 feed = self.feed\_type(  
 title=self.\_get\_dynamic\_attr("title", obj),  
 subtitle=self.\_get\_dynamic\_attr("subtitle", obj),  
 link=link,  
 description=self.\_get\_dynamic\_attr("description", obj),  
 language=self.language or get\_language(),  
 feed\_url=add\_domain(  
 current\_site.domain,  
 self.\_get\_dynamic\_attr("feed\_url", obj) or request.path,  
 request.is\_secure(),  
 ),  
 author\_name=self.\_get\_dynamic\_attr("author\_name", obj),  
 author\_link=self.\_get\_dynamic\_attr("author\_link", obj),  
 author\_email=self.\_get\_dynamic\_attr("author\_email", obj),  
 categories=self.\_get\_dynamic\_attr("categories", obj),  
 feed\_copyright=self.\_get\_dynamic\_attr("feed\_copyright", obj),  
 feed\_guid=self.\_get\_dynamic\_attr("feed\_guid", obj),  
 ttl=self.\_get\_dynamic\_attr("ttl", obj),  
 \*\*self.feed\_extra\_kwargs(obj),  
 )  
  
 title\_tmp = None  
 if self.title\_template is not None:  
 try:  
 title\_tmp = loader.get\_template(self.title\_template)  
 except TemplateDoesNotExist:  
 pass  
  
 description\_tmp = None  
 if self.description\_template is not None:  
 try:  
 description\_tmp = loader.get\_template(self.description\_template)  
 except TemplateDoesNotExist:  
 pass  
  
 for item in self.\_get\_dynamic\_attr("items", obj):  
 context = self.get\_context\_data(  
 item=item, site=current\_site, obj=obj, request=request  
 )  
 if title\_tmp is not None:  
 title = title\_tmp.render(context, request)  
 else:  
 title = self.\_get\_dynamic\_attr("item\_title", item)  
 if description\_tmp is not None:  
 description = description\_tmp.render(context, request)  
 else:  
 description = self.\_get\_dynamic\_attr("item\_description", item)  
 link = add\_domain(  
 current\_site.domain,  
 self.\_get\_dynamic\_attr("item\_link", item),  
 request.is\_secure(),  
 )  
 enclosures = self.\_get\_dynamic\_attr("item\_enclosures", item)  
 author\_name = self.\_get\_dynamic\_attr("item\_author\_name", item)  
 if author\_name is not None:  
 author\_email = self.\_get\_dynamic\_attr("item\_author\_email", item)  
 author\_link = self.\_get\_dynamic\_attr("item\_author\_link", item)  
 else:  
 author\_email = author\_link = None  
  
 tz = get\_default\_timezone()  
  
 pubdate = self.\_get\_dynamic\_attr("item\_pubdate", item)  
 if pubdate and is\_naive(pubdate):  
 pubdate = make\_aware(pubdate, tz)  
  
 updateddate = self.\_get\_dynamic\_attr("item\_updateddate", item)  
 if updateddate and is\_naive(updateddate):  
 updateddate = make\_aware(updateddate, tz)  
  
 feed.add\_item(  
 title=title,  
 link=link,  
 description=description,  
 unique\_id=self.\_get\_dynamic\_attr("item\_guid", item, link),  
 unique\_id\_is\_permalink=self.\_get\_dynamic\_attr(  
 "item\_guid\_is\_permalink", item  
 ),  
 enclosures=enclosures,  
 pubdate=pubdate,  
 updateddate=updateddate,  
 author\_name=author\_name,  
 author\_email=author\_email,  
 author\_link=author\_link,  
 comments=self.\_get\_dynamic\_attr("item\_comments", item),  
 categories=self.\_get\_dynamic\_attr("item\_categories", item),  
 item\_copyright=self.\_get\_dynamic\_attr("item\_copyright", item),  
 \*\*self.item\_extra\_kwargs(item),  
 )  
 return feed

## File: venv/lib/python3.12/site-packages/rest\_framework/authtoken/views.py

from rest\_framework import parsers, renderers  
from rest\_framework.authtoken.models import Token  
from rest\_framework.authtoken.serializers import AuthTokenSerializer  
from rest\_framework.compat import coreapi, coreschema  
from rest\_framework.response import Response  
from rest\_framework.schemas import ManualSchema  
from rest\_framework.schemas import coreapi as coreapi\_schema  
from rest\_framework.views import APIView  
  
  
class ObtainAuthToken(APIView):  
 throttle\_classes = ()  
 permission\_classes = ()  
 parser\_classes = (parsers.FormParser, parsers.MultiPartParser, parsers.JSONParser,)  
 renderer\_classes = (renderers.JSONRenderer,)  
 serializer\_class = AuthTokenSerializer  
  
 if coreapi\_schema.is\_enabled():  
 schema = ManualSchema(  
 fields=[  
 coreapi.Field(  
 name="username",  
 required=True,  
 location='form',  
 schema=coreschema.String(  
 title="Username",  
 description="Valid username for authentication",  
 ),  
 ),  
 coreapi.Field(  
 name="password",  
 required=True,  
 location='form',  
 schema=coreschema.String(  
 title="Password",  
 description="Valid password for authentication",  
 ),  
 ),  
 ],  
 encoding="application/json",  
 )  
  
 def get\_serializer\_context(self):  
 return {  
 'request': self.request,  
 'format': self.format\_kwarg,  
 'view': self  
 }  
  
 def get\_serializer(self, \*args, \*\*kwargs):  
 kwargs['context'] = self.get\_serializer\_context()  
 return self.serializer\_class(\*args, \*\*kwargs)  
  
 def post(self, request, \*args, \*\*kwargs):  
 serializer = self.get\_serializer(data=request.data)  
 serializer.is\_valid(raise\_exception=True)  
 user = serializer.validated\_data['user']  
 token, created = Token.objects.get\_or\_create(user=user)  
 return Response({'token': token.key})  
  
  
obtain\_auth\_token = ObtainAuthToken.as\_view()

## File: venv/lib/python3.12/site-packages/rest\_framework/schemas/views.py

"""  
views.py # Houses `SchemaView`, `APIView` subclass.  
  
See schemas.\_\_init\_\_.py for package overview.  
"""  
from rest\_framework import exceptions, renderers  
from rest\_framework.response import Response  
from rest\_framework.schemas import coreapi  
from rest\_framework.settings import api\_settings  
from rest\_framework.views import APIView  
  
  
class SchemaView(APIView):  
 \_ignore\_model\_permissions = True  
 schema = None # exclude from schema  
 renderer\_classes = None  
 schema\_generator = None  
 public = False  
  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
 if self.renderer\_classes is None:  
 if coreapi.is\_enabled():  
 self.renderer\_classes = [  
 renderers.CoreAPIOpenAPIRenderer,  
 renderers.CoreJSONRenderer  
 ]  
 else:  
 self.renderer\_classes = [  
 renderers.OpenAPIRenderer,  
 renderers.JSONOpenAPIRenderer,  
 ]  
 if renderers.BrowsableAPIRenderer in api\_settings.DEFAULT\_RENDERER\_CLASSES:  
 self.renderer\_classes += [renderers.BrowsableAPIRenderer]  
  
 def get(self, request, \*args, \*\*kwargs):  
 schema = self.schema\_generator.get\_schema(request, self.public)  
 if schema is None:  
 raise exceptions.PermissionDenied()  
 return Response(schema)  
  
 def handle\_exception(self, exc):  
 # Schema renderers do not render exceptions, so re-perform content  
 # negotiation with default renderers.  
 self.renderer\_classes = api\_settings.DEFAULT\_RENDERER\_CLASSES  
 neg = self.perform\_content\_negotiation(self.request, force=True)  
 self.request.accepted\_renderer, self.request.accepted\_media\_type = neg  
 return super().handle\_exception(exc)

## File: venv/lib/python3.12/site-packages/rest\_framework/views.py

"""  
Provides an APIView class that is the base of all views in REST framework.  
"""  
from django.conf import settings  
from django.core.exceptions import PermissionDenied  
from django.db import connections, models  
from django.http import Http404  
from django.http.response import HttpResponseBase  
from django.utils.cache import cc\_delim\_re, patch\_vary\_headers  
from django.utils.encoding import smart\_str  
from django.views.decorators.csrf import csrf\_exempt  
from django.views.generic import View  
  
from rest\_framework import exceptions, status  
from rest\_framework.request import Request  
from rest\_framework.response import Response  
from rest\_framework.schemas import DefaultSchema  
from rest\_framework.settings import api\_settings  
from rest\_framework.utils import formatting  
  
  
def get\_view\_name(view):  
 """  
 Given a view instance, return a textual name to represent the view.  
 This name is used in the browsable API, and in OPTIONS responses.  
  
 This function is the default for the `VIEW\_NAME\_FUNCTION` setting.  
 """  
 # Name may be set by some Views, such as a ViewSet.  
 name = getattr(view, 'name', None)  
 if name is not None:  
 return name  
  
 name = view.\_\_class\_\_.\_\_name\_\_  
 name = formatting.remove\_trailing\_string(name, 'View')  
 name = formatting.remove\_trailing\_string(name, 'ViewSet')  
 name = formatting.camelcase\_to\_spaces(name)  
  
 # Suffix may be set by some Views, such as a ViewSet.  
 suffix = getattr(view, 'suffix', None)  
 if suffix:  
 name += ' ' + suffix  
  
 return name  
  
  
def get\_view\_description(view, html=False):  
 """  
 Given a view instance, return a textual description to represent the view.  
 This name is used in the browsable API, and in OPTIONS responses.  
  
 This function is the default for the `VIEW\_DESCRIPTION\_FUNCTION` setting.  
 """  
 # Description may be set by some Views, such as a ViewSet.  
 description = getattr(view, 'description', None)  
 if description is None:  
 description = view.\_\_class\_\_.\_\_doc\_\_ or ''  
  
 description = formatting.dedent(smart\_str(description))  
 if html:  
 return formatting.markup\_description(description)  
 return description  
  
  
def set\_rollback():  
 for db in connections.all():  
 if db.settings\_dict['ATOMIC\_REQUESTS'] and db.in\_atomic\_block:  
 db.set\_rollback(True)  
  
  
def exception\_handler(exc, context):  
 """  
 Returns the response that should be used for any given exception.  
  
 By default we handle the REST framework `APIException`, and also  
 Django's built-in `Http404` and `PermissionDenied` exceptions.  
  
 Any unhandled exceptions may return `None`, which will cause a 500 error  
 to be raised.  
 """  
 if isinstance(exc, Http404):  
 exc = exceptions.NotFound()  
 elif isinstance(exc, PermissionDenied):  
 exc = exceptions.PermissionDenied()  
  
 if isinstance(exc, exceptions.APIException):  
 headers = {}  
 if getattr(exc, 'auth\_header', None):  
 headers['WWW-Authenticate'] = exc.auth\_header  
 if getattr(exc, 'wait', None):  
 headers['Retry-After'] = '%d' % exc.wait  
  
 if isinstance(exc.detail, (list, dict)):  
 data = exc.detail  
 else:  
 data = {'detail': exc.detail}  
  
 set\_rollback()  
 return Response(data, status=exc.status\_code, headers=headers)  
  
 return None  
  
  
class APIView(View):  
  
 # The following policies may be set at either globally, or per-view.  
 renderer\_classes = api\_settings.DEFAULT\_RENDERER\_CLASSES  
 parser\_classes = api\_settings.DEFAULT\_PARSER\_CLASSES  
 authentication\_classes = api\_settings.DEFAULT\_AUTHENTICATION\_CLASSES  
 throttle\_classes = api\_settings.DEFAULT\_THROTTLE\_CLASSES  
 permission\_classes = api\_settings.DEFAULT\_PERMISSION\_CLASSES  
 content\_negotiation\_class = api\_settings.DEFAULT\_CONTENT\_NEGOTIATION\_CLASS  
 metadata\_class = api\_settings.DEFAULT\_METADATA\_CLASS  
 versioning\_class = api\_settings.DEFAULT\_VERSIONING\_CLASS  
  
 # Allow dependency injection of other settings to make testing easier.  
 settings = api\_settings  
  
 schema = DefaultSchema()  
  
 @classmethod  
 def as\_view(cls, \*\*initkwargs):  
 """  
 Store the original class on the view function.  
  
 This allows us to discover information about the view when we do URL  
 reverse lookups. Used for breadcrumb generation.  
 """  
 if isinstance(getattr(cls, 'queryset', None), models.query.QuerySet):  
 def force\_evaluation():  
 raise RuntimeError(  
 'Do not evaluate the `.queryset` attribute directly, '  
 'as the result will be cached and reused between requests. '  
 'Use `.all()` or call `.get\_queryset()` instead.'  
 )  
 cls.queryset.\_fetch\_all = force\_evaluation  
  
 view = super().as\_view(\*\*initkwargs)  
 view.cls = cls  
 view.initkwargs = initkwargs  
  
 # Note: session based authentication is explicitly CSRF validated,  
 # all other authentication is CSRF exempt.  
 return csrf\_exempt(view)  
  
 @property  
 def allowed\_methods(self):  
 """  
 Wrap Django's private `\_allowed\_methods` interface in a public property.  
 """  
 return self.\_allowed\_methods()  
  
 @property  
 def default\_response\_headers(self):  
 headers = {  
 'Allow': ', '.join(self.allowed\_methods),  
 }  
 if len(self.renderer\_classes) > 1:  
 headers['Vary'] = 'Accept'  
 return headers  
  
 def http\_method\_not\_allowed(self, request, \*args, \*\*kwargs):  
 """  
 If `request.method` does not correspond to a handler method,  
 determine what kind of exception to raise.  
 """  
 raise exceptions.MethodNotAllowed(request.method)  
  
 def permission\_denied(self, request, message=None, code=None):  
 """  
 If request is not permitted, determine what kind of exception to raise.  
 """  
 if request.authenticators and not request.successful\_authenticator:  
 raise exceptions.NotAuthenticated()  
 raise exceptions.PermissionDenied(detail=message, code=code)  
  
 def throttled(self, request, wait):  
 """  
 If request is throttled, determine what kind of exception to raise.  
 """  
 raise exceptions.Throttled(wait)  
  
 def get\_authenticate\_header(self, request):  
 """  
 If a request is unauthenticated, determine the WWW-Authenticate  
 header to use for 401 responses, if any.  
 """  
 authenticators = self.get\_authenticators()  
 if authenticators:  
 return authenticators[0].authenticate\_header(request)  
  
 def get\_parser\_context(self, http\_request):  
 """  
 Returns a dict that is passed through to Parser.parse(),  
 as the `parser\_context` keyword argument.  
 """  
 # Note: Additionally `request` and `encoding` will also be added  
 # to the context by the Request object.  
 return {  
 'view': self,  
 'args': getattr(self, 'args', ()),  
 'kwargs': getattr(self, 'kwargs', {})  
 }  
  
 def get\_renderer\_context(self):  
 """  
 Returns a dict that is passed through to Renderer.render(),  
 as the `renderer\_context` keyword argument.  
 """  
 # Note: Additionally 'response' will also be added to the context,  
 # by the Response object.  
 return {  
 'view': self,  
 'args': getattr(self, 'args', ()),  
 'kwargs': getattr(self, 'kwargs', {}),  
 'request': getattr(self, 'request', None)  
 }  
  
 def get\_exception\_handler\_context(self):  
 """  
 Returns a dict that is passed through to EXCEPTION\_HANDLER,  
 as the `context` argument.  
 """  
 return {  
 'view': self,  
 'args': getattr(self, 'args', ()),  
 'kwargs': getattr(self, 'kwargs', {}),  
 'request': getattr(self, 'request', None)  
 }  
  
 def get\_view\_name(self):  
 """  
 Return the view name, as used in OPTIONS responses and in the  
 browsable API.  
 """  
 func = self.settings.VIEW\_NAME\_FUNCTION  
 return func(self)  
  
 def get\_view\_description(self, html=False):  
 """  
 Return some descriptive text for the view, as used in OPTIONS responses  
 and in the browsable API.  
 """  
 func = self.settings.VIEW\_DESCRIPTION\_FUNCTION  
 return func(self, html)  
  
 # API policy instantiation methods  
  
 def get\_format\_suffix(self, \*\*kwargs):  
 """  
 Determine if the request includes a '.json' style format suffix  
 """  
 if self.settings.FORMAT\_SUFFIX\_KWARG:  
 return kwargs.get(self.settings.FORMAT\_SUFFIX\_KWARG)  
  
 def get\_renderers(self):  
 """  
 Instantiates and returns the list of renderers that this view can use.  
 """  
 return [renderer() for renderer in self.renderer\_classes]  
  
 def get\_parsers(self):  
 """  
 Instantiates and returns the list of parsers that this view can use.  
 """  
 return [parser() for parser in self.parser\_classes]  
  
 def get\_authenticators(self):  
 """  
 Instantiates and returns the list of authenticators that this view can use.  
 """  
 return [auth() for auth in self.authentication\_classes]  
  
 def get\_permissions(self):  
 """  
 Instantiates and returns the list of permissions that this view requires.  
 """  
 return [permission() for permission in self.permission\_classes]  
  
 def get\_throttles(self):  
 """  
 Instantiates and returns the list of throttles that this view uses.  
 """  
 return [throttle() for throttle in self.throttle\_classes]  
  
 def get\_content\_negotiator(self):  
 """  
 Instantiate and return the content negotiation class to use.  
 """  
 if not getattr(self, '\_negotiator', None):  
 self.\_negotiator = self.content\_negotiation\_class()  
 return self.\_negotiator  
  
 def get\_exception\_handler(self):  
 """  
 Returns the exception handler that this view uses.  
 """  
 return self.settings.EXCEPTION\_HANDLER  
  
 # API policy implementation methods  
  
 def perform\_content\_negotiation(self, request, force=False):  
 """  
 Determine which renderer and media type to use render the response.  
 """  
 renderers = self.get\_renderers()  
 conneg = self.get\_content\_negotiator()  
  
 try:  
 return conneg.select\_renderer(request, renderers, self.format\_kwarg)  
 except Exception:  
 if force:  
 return (renderers[0], renderers[0].media\_type)  
 raise  
  
 def perform\_authentication(self, request):  
 """  
 Perform authentication on the incoming request.  
  
 Note that if you override this and simply 'pass', then authentication  
 will instead be performed lazily, the first time either  
 `request.user` or `request.auth` is accessed.  
 """  
 request.user  
  
 def check\_permissions(self, request):  
 """  
 Check if the request should be permitted.  
 Raises an appropriate exception if the request is not permitted.  
 """  
 for permission in self.get\_permissions():  
 if not permission.has\_permission(request, self):  
 self.permission\_denied(  
 request,  
 message=getattr(permission, 'message', None),  
 code=getattr(permission, 'code', None)  
 )  
  
 def check\_object\_permissions(self, request, obj):  
 """  
 Check if the request should be permitted for a given object.  
 Raises an appropriate exception if the request is not permitted.  
 """  
 for permission in self.get\_permissions():  
 if not permission.has\_object\_permission(request, self, obj):  
 self.permission\_denied(  
 request,  
 message=getattr(permission, 'message', None),  
 code=getattr(permission, 'code', None)  
 )  
  
 def check\_throttles(self, request):  
 """  
 Check if request should be throttled.  
 Raises an appropriate exception if the request is throttled.  
 """  
 throttle\_durations = []  
 for throttle in self.get\_throttles():  
 if not throttle.allow\_request(request, self):  
 throttle\_durations.append(throttle.wait())  
  
 if throttle\_durations:  
 # Filter out `None` values which may happen in case of config / rate  
 # changes, see #1438  
 durations = [  
 duration for duration in throttle\_durations  
 if duration is not None  
 ]  
  
 duration = max(durations, default=None)  
 self.throttled(request, duration)  
  
 def determine\_version(self, request, \*args, \*\*kwargs):  
 """  
 If versioning is being used, then determine any API version for the  
 incoming request. Returns a two-tuple of (version, versioning\_scheme)  
 """  
 if self.versioning\_class is None:  
 return (None, None)  
 scheme = self.versioning\_class()  
 return (scheme.determine\_version(request, \*args, \*\*kwargs), scheme)  
  
 # Dispatch methods  
  
 def initialize\_request(self, request, \*args, \*\*kwargs):  
 """  
 Returns the initial request object.  
 """  
 parser\_context = self.get\_parser\_context(request)  
  
 return Request(  
 request,  
 parsers=self.get\_parsers(),  
 authenticators=self.get\_authenticators(),  
 negotiator=self.get\_content\_negotiator(),  
 parser\_context=parser\_context  
 )  
  
 def initial(self, request, \*args, \*\*kwargs):  
 """  
 Runs anything that needs to occur prior to calling the method handler.  
 """  
 self.format\_kwarg = self.get\_format\_suffix(\*\*kwargs)  
  
 # Perform content negotiation and store the accepted info on the request  
 neg = self.perform\_content\_negotiation(request)  
 request.accepted\_renderer, request.accepted\_media\_type = neg  
  
 # Determine the API version, if versioning is in use.  
 version, scheme = self.determine\_version(request, \*args, \*\*kwargs)  
 request.version, request.versioning\_scheme = version, scheme  
  
 # Ensure that the incoming request is permitted  
 self.perform\_authentication(request)  
 self.check\_permissions(request)  
 self.check\_throttles(request)  
  
 def finalize\_response(self, request, response, \*args, \*\*kwargs):  
 """  
 Returns the final response object.  
 """  
 # Make the error obvious if a proper response is not returned  
 assert isinstance(response, HttpResponseBase), (  
 'Expected a `Response`, `HttpResponse` or `HttpStreamingResponse` '  
 'to be returned from the view, but received a `%s`'  
 % type(response)  
 )  
  
 if isinstance(response, Response):  
 if not getattr(request, 'accepted\_renderer', None):  
 neg = self.perform\_content\_negotiation(request, force=True)  
 request.accepted\_renderer, request.accepted\_media\_type = neg  
  
 response.accepted\_renderer = request.accepted\_renderer  
 response.accepted\_media\_type = request.accepted\_media\_type  
 response.renderer\_context = self.get\_renderer\_context()  
  
 # Add new vary headers to the response instead of overwriting.  
 vary\_headers = self.headers.pop('Vary', None)  
 if vary\_headers is not None:  
 patch\_vary\_headers(response, cc\_delim\_re.split(vary\_headers))  
  
 for key, value in self.headers.items():  
 response[key] = value  
  
 return response  
  
 def handle\_exception(self, exc):  
 """  
 Handle any exception that occurs, by returning an appropriate response,  
 or re-raising the error.  
 """  
 if isinstance(exc, (exceptions.NotAuthenticated,  
 exceptions.AuthenticationFailed)):  
 # WWW-Authenticate header for 401 responses, else coerce to 403  
 auth\_header = self.get\_authenticate\_header(self.request)  
  
 if auth\_header:  
 exc.auth\_header = auth\_header  
 else:  
 exc.status\_code = status.HTTP\_403\_FORBIDDEN  
  
 exception\_handler = self.get\_exception\_handler()  
  
 context = self.get\_exception\_handler\_context()  
 response = exception\_handler(exc, context)  
  
 if response is None:  
 self.raise\_uncaught\_exception(exc)  
  
 response.exception = True  
 return response  
  
 def raise\_uncaught\_exception(self, exc):  
 if settings.DEBUG:  
 request = self.request  
 renderer\_format = getattr(request.accepted\_renderer, 'format')  
 use\_plaintext\_traceback = renderer\_format not in ('html', 'api', 'admin')  
 request.force\_plaintext\_errors(use\_plaintext\_traceback)  
 raise exc  
  
 # Note: Views are made CSRF exempt from within `as\_view` as to prevent  
 # accidental removal of this exemption in cases where `dispatch` needs to  
 # be overridden.  
 def dispatch(self, request, \*args, \*\*kwargs):  
 """  
 `.dispatch()` is pretty much the same as Django's regular dispatch,  
 but with extra hooks for startup, finalize, and exception handling.  
 """  
 self.args = args  
 self.kwargs = kwargs  
 request = self.initialize\_request(request, \*args, \*\*kwargs)  
 self.request = request  
 self.headers = self.default\_response\_headers # deprecate?  
  
 try:  
 self.initial(request, \*args, \*\*kwargs)  
  
 # Get the appropriate handler method  
 if request.method.lower() in self.http\_method\_names:  
 handler = getattr(self, request.method.lower(),  
 self.http\_method\_not\_allowed)  
 else:  
 handler = self.http\_method\_not\_allowed  
  
 response = handler(request, \*args, \*\*kwargs)  
  
 except Exception as exc:  
 response = self.handle\_exception(exc)  
  
 self.response = self.finalize\_response(request, response, \*args, \*\*kwargs)  
 return self.response  
  
 def options(self, request, \*args, \*\*kwargs):  
 """  
 Handler method for HTTP 'OPTIONS' request.  
 """  
 if self.metadata\_class is None:  
 return self.http\_method\_not\_allowed(request, \*args, \*\*kwargs)  
 data = self.metadata\_class().determine\_metadata(request, self)  
 return Response(data, status=status.HTTP\_200\_OK)

## File: venv/lib/python3.12/site-packages/rest\_framework\_simplejwt/views.py

from django.utils.module\_loading import import\_string  
from rest\_framework import generics, status  
from rest\_framework.request import Request  
from rest\_framework.response import Response  
from rest\_framework.serializers import Serializer  
  
from .authentication import AUTH\_HEADER\_TYPES  
from .exceptions import InvalidToken, TokenError  
from .settings import api\_settings  
  
  
class TokenViewBase(generics.GenericAPIView):  
 permission\_classes = ()  
 authentication\_classes = ()  
  
 serializer\_class = None  
 \_serializer\_class = ""  
  
 www\_authenticate\_realm = "api"  
  
 def get\_serializer\_class(self) -> Serializer:  
 """  
 If serializer\_class is set, use it directly. Otherwise get the class from settings.  
 """  
  
 if self.serializer\_class:  
 return self.serializer\_class  
 try:  
 return import\_string(self.\_serializer\_class)  
 except ImportError:  
 msg = "Could not import serializer '%s'" % self.\_serializer\_class  
 raise ImportError(msg)  
  
 def get\_authenticate\_header(self, request: Request) -> str:  
 return '{} realm="{}"'.format(  
 AUTH\_HEADER\_TYPES[0],  
 self.www\_authenticate\_realm,  
 )  
  
 def post(self, request: Request, \*args, \*\*kwargs) -> Response:  
 serializer = self.get\_serializer(data=request.data)  
  
 try:  
 serializer.is\_valid(raise\_exception=True)  
 except TokenError as e:  
 raise InvalidToken(e.args[0])  
  
 return Response(serializer.validated\_data, status=status.HTTP\_200\_OK)  
  
  
class TokenObtainPairView(TokenViewBase):  
 """  
 Takes a set of user credentials and returns an access and refresh JSON web  
 token pair to prove the authentication of those credentials.  
 """  
  
 \_serializer\_class = api\_settings.TOKEN\_OBTAIN\_SERIALIZER  
  
  
token\_obtain\_pair = TokenObtainPairView.as\_view()  
  
  
class TokenRefreshView(TokenViewBase):  
 """  
 Takes a refresh type JSON web token and returns an access type JSON web  
 token if the refresh token is valid.  
 """  
  
 \_serializer\_class = api\_settings.TOKEN\_REFRESH\_SERIALIZER  
  
  
token\_refresh = TokenRefreshView.as\_view()  
  
  
class TokenObtainSlidingView(TokenViewBase):  
 """  
 Takes a set of user credentials and returns a sliding JSON web token to  
 prove the authentication of those credentials.  
 """  
  
 \_serializer\_class = api\_settings.SLIDING\_TOKEN\_OBTAIN\_SERIALIZER  
  
  
token\_obtain\_sliding = TokenObtainSlidingView.as\_view()  
  
  
class TokenRefreshSlidingView(TokenViewBase):  
 """  
 Takes a sliding JSON web token and returns a new, refreshed version if the  
 token's refresh period has not expired.  
 """  
  
 \_serializer\_class = api\_settings.SLIDING\_TOKEN\_REFRESH\_SERIALIZER  
  
  
token\_refresh\_sliding = TokenRefreshSlidingView.as\_view()  
  
  
class TokenVerifyView(TokenViewBase):  
 """  
 Takes a token and indicates if it is valid. This view provides no  
 information about a token's fitness for a particular use.  
 """  
  
 \_serializer\_class = api\_settings.TOKEN\_VERIFY\_SERIALIZER  
  
  
token\_verify = TokenVerifyView.as\_view()  
  
  
class TokenBlacklistView(TokenViewBase):  
 """  
 Takes a token and blacklists it. Must be used with the  
 `rest\_framework\_simplejwt.token\_blacklist` app installed.  
 """  
  
 \_serializer\_class = api\_settings.TOKEN\_BLACKLIST\_SERIALIZER  
  
  
token\_blacklist = TokenBlacklistView.as\_view()

# Models Documentation

## File: jobs/models.py

from django.db import models  
from users.models import Profile  
from projects.models import Tag  
import uuid  
from datetime import datetime  
  
# Create your models here.  
  
  
class Job(models.Model):  
 owner = models.ForeignKey(Profile, null=True, blank=True, on\_delete=models.CASCADE)  
 title = models.CharField(max\_length=200, null=True, blank=True)  
 budget = models.CharField(max\_length=200, null=True, blank=True)  
 duration = models.CharField(max\_length=200, null=True, blank=True)  
 featured\_image = models.ImageField(null=True, blank=True, default='default.jpg')  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
 tags = models.ManyToManyField(Tag, blank=True)  
 click\_total = models.IntegerField(default=0, null=True, blank=True)  
 clicked = models.ManyToManyField(Profile, blank=True, related\_name="clicked")  
 assigned = models.OneToOneField(Profile, blank=True, related\_name="assigned", null=True, on\_delete=models.SET\_NULL)  
 assigned\_date = models.DateTimeField(default=datetime.now)  
 description = models.TextField(max\_length=2000, null=True, blank=True)  
 created = models.DateTimeField(auto\_now\_add=True)  
 is\_active = models.BooleanField(default=True, null=True, blank=True)  
 is\_done = models.BooleanField(default=False, null=True, blank=True)  
 is\_assigned = models.BooleanField(default=False, null=True, blank=True)  
  
 def \_\_str\_\_(self):  
 return self.title  
  
 class Meta:  
 ordering = ['-created', 'title']  
  
  
class Contract(models.Model):  
 job = models.ForeignKey(Job, on\_delete=models.CASCADE)  
 client = models.ForeignKey(Profile, null=True, blank=True, on\_delete=models.CASCADE, related\_name="Client")  
 freelancer = models.ForeignKey(Profile, null=True, blank=True, on\_delete=models.CASCADE, related\_name="Freelancer")  
 title = models.CharField(max\_length=200, null=True, blank=True)  
 budget = models.CharField(max\_length=200, null=True, blank=True)  
 duration = models.CharField(max\_length=200, null=True, blank=True)  
 featured\_image = models.ImageField(null=True, blank=True, default='default.jpg')  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
 description = models.TextField(max\_length=2000, null=True, blank=True)  
 created = models.DateTimeField(auto\_now\_add=True)  
 is\_signed = models.BooleanField(default=False, null=True, blank=True)  
  
 def \_\_str\_\_(self):  
 return self.title  
  
 class Meta:  
 ordering = ['-created', 'title']

## File: projects/models.py

from django.db import models  
import uuid  
from users.models import Profile  
  
  
  
  
class Project(models.Model):  
 owner = models.ForeignKey(Profile, null=True, blank=True, on\_delete=models.CASCADE)  
 title = models.CharField(max\_length=200)  
 featured\_image = models.ImageField(null=True, blank=True, default='default.jpg')  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
 tags = models.ManyToManyField('Tag', blank=True)  
 vote\_total = models.IntegerField(default=0, null=True, blank=True)  
 vote\_ratio = models.IntegerField(default=0, null=True, blank=True)  
 description = models.TextField(max\_length=2000, null=True, blank=True)  
 demo\_link = models.CharField(max\_length=200, null=True, blank=True)  
 source\_link = models.CharField(max\_length=200, null=True, blank=True)  
 created = models.DateTimeField(auto\_now\_add=True)  
  
 def \_\_str\_\_(self):  
 return self.title  
  
 class Meta:  
 ordering = ['-vote\_ratio', '-vote\_total', 'title']  
 #  
 # @property  
 # def imageURL(self):  
 # try:  
 # url = self.featured\_image.url  
 # except:  
 # url = "static 'default.jpg'"  
 # return url  
  
 @property  
 def reviewers(self):  
 queryset = self.review\_set.all().values\_list('owner\_\_id', flat=True)  
 return queryset  
  
  
  
 @property  
 def getVoteCount(self):  
 reviews = self.review\_set.all()  
   
   
 totalVotes = reviews.count()  
  
 ratio = (upVotes/totalVotes) \* 100  
 self.vote\_total =totalVotes  
 self.vote\_ratio = ratio  
 self.save()  
class Review(models.Model):  
 # owner  
 VOICE\_TYPE = (  
 ('up', 'Upvote'), ('down', 'Downvote')  
 )  
 owner = models.ForeignKey(Profile, on\_delete=models.CASCADE, null=True)  
 project = models.ForeignKey(Project, on\_delete=models.CASCADE)  
 body = models.TextField(max\_length=2000, null=True, blank=True)  
 created = models.DateTimeField(auto\_now\_add=True)  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
 value = models.CharField(max\_length=200, choices=VOICE\_TYPE)  
  
 class Meta:  
 unique\_together = [['owner', 'project']]  
  
 def \_\_str\_\_(self):  
 return self.value  
  
  
class Tag(models.Model):  
 name = models.CharField(max\_length=200)  
 created = models.DateTimeField(auto\_now\_add=True)  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
  
 def \_\_str\_\_(self):  
 return self.name

## File: users/models.py

from django.db import models  
from django.contrib.auth.models import User, Group  
import uuid  
from django.db.models.signals import post\_save, post\_delete  
from django.core.exceptions import ValidationError  
from django.utils.translation import gettext\_lazy as \_  
from django.core.validators import EmailValidator  
  
  
def custom\_email\_validator(value):  
 # Add your custom email validation logic here  
 forbidden\_strings = ['gmail', 'mail', 'yahoo', 'zoho', 'yandex']  
 if any(forbidden\_str in value for forbidden\_str in forbidden\_strings):  
 raise ValidationError(\_('Use your WIUT or company email!'))  
   
# Create your models here.  
  
class Profile(models.Model):  
 user\_type = models.CharField(max\_length=20, null=True, blank=True)  
 user = models.OneToOneField(User, on\_delete=models.CASCADE, null=True, blank=True, related\_name="profile")  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
 created = models.DateTimeField(auto\_now\_add=True)  
 username = models.CharField(max\_length=200, blank=True, null=True)  
 location = models.CharField(max\_length=200, blank=True, null=True)  
 email = models.EmailField(max\_length=200, blank=True, null=True, unique=False)  
 name = models.CharField(max\_length=200, blank=True, null=True)  
 short\_intro = models.CharField(max\_length=500, blank=True, null=True)  
 profile\_pic = models.ImageField(null=True, blank=True, upload\_to='profiles/', default='profiles/default.jpg')  
 bio = models.TextField(blank=True, null=True)  
 social\_linkedin = models.CharField(max\_length=200, blank=True, null=True)  
 social\_instagram = models.CharField(max\_length=200, blank=True, null=True)  
 social\_telegram = models.CharField(max\_length=200, blank=True, null=True)  
 social\_github = models.CharField(max\_length=200, blank=True, null=True)  
 social\_website = models.CharField(max\_length=200, blank=True, null=True)  
 is\_activated =models.BooleanField(default=False)  
   
  
 def validate\_unique(self, exclude=None):  
 super().validate\_unique(exclude=exclude)  
  
 # def clean\_email(self):  
 # super().clean()  
 # print(self.user\_type)  
 # # Check if the user\_type is a certain type where you want to bypass the custom\_email\_validator  
 # if self.user\_type == 'Client':  
 # print("Bypassing validation for Client")  
 # return self.cleaned\_data['email']  
 # else:  
 # # If not, perform the custom email validation  
 # forbidden\_strings = ['gmail', 'mail', 'yahoo', 'zoho', 'yandex']  
 # for forbidden\_str in forbidden\_strings:  
 # if forbidden\_str in self.email:  
 # raise ValidationError('Use your WIUT or company email!')  
   
 # return self.cleaned\_data['email']  
   
 def \_\_str\_\_(self):  
 return str(self.username)  
  
 class Meta:  
 ordering = ['-created']  
  
  
class Skill(models.Model):  
 owner = models.ForeignKey(Profile, on\_delete=models.CASCADE, null=True, blank=True)  
 name = models.CharField(max\_length=200, blank=True, null=True)  
 description = models.TextField(null=True, blank=True)  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
 created = models.DateTimeField(auto\_now\_add=True)  
  
 def \_\_str\_\_(self):  
 return str(self.name)  
  
  
class Message(models.Model):  
 sender = models.ForeignKey(Profile, on\_delete=models.SET\_NULL, null=True, blank=True)  
 recipient = models.ForeignKey(Profile, on\_delete=models.SET\_NULL, null=True, blank=True, related\_name="messages")  
 # name = models.CharField(max\_length=200, null=True, blank=True)  
 email = models.EmailField(max\_length=200, null=True, blank=True)  
 subject = models.CharField(max\_length=200, null=True, blank=True)  
 body = models.TextField()  
  
 created = models.DateTimeField(auto\_now\_add=True)  
 id = models.UUIDField(default=uuid.uuid4, unique=True, primary\_key=True, editable=False)  
 is\_read = models.BooleanField(default=False, null=True)  
 attached = models.FileField(upload\_to='attachments/', null=True, blank=True)  
 def \_\_str\_\_(self):  
 return self.subject  
  
 class Meta:  
 ordering = ['is\_read', '-created']

## File: venv/lib/python3.12/site-packages/captcha/models.py

import datetime  
import hashlib  
import logging  
import random  
import time  
  
from django.db import models  
from django.utils import timezone  
from django.utils.encoding import smart\_str  
  
from captcha.conf import settings as captcha\_settings  
  
  
# Heavily based on session key generation in Django  
# Use the system (hardware-based) random number generator if it exists.  
if hasattr(random, "SystemRandom"):  
 randrange = random.SystemRandom().randrange  
else:  
 randrange = random.randrange  
MAX\_RANDOM\_KEY = 18446744073709551616 # 2 << 63  
  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class CaptchaStore(models.Model):  
 id = models.AutoField(primary\_key=True)  
 challenge = models.CharField(blank=False, max\_length=32)  
 response = models.CharField(blank=False, max\_length=32)  
 hashkey = models.CharField(blank=False, max\_length=40, unique=True)  
 expiration = models.DateTimeField(blank=False)  
  
 def save(self, \*args, \*\*kwargs):  
 self.response = self.response.lower()  
 if not self.expiration:  
 self.expiration = timezone.now() + datetime.timedelta(  
 minutes=int(captcha\_settings.CAPTCHA\_TIMEOUT)  
 )  
 if not self.hashkey:  
 key\_ = (  
 smart\_str(randrange(0, MAX\_RANDOM\_KEY))  
 + smart\_str(time.time())  
 + smart\_str(self.challenge, errors="ignore")  
 + smart\_str(self.response, errors="ignore")  
 ).encode("utf8")  
 self.hashkey = hashlib.sha1(key\_).hexdigest()  
 del key\_  
 super().save(\*args, \*\*kwargs)  
  
 def \_\_str\_\_(self):  
 return self.challenge  
  
 def remove\_expired(cls):  
 cls.objects.filter(expiration\_\_lte=timezone.now()).delete()  
  
 remove\_expired = classmethod(remove\_expired)  
  
 @classmethod  
 def generate\_key(cls, generator=None):  
 challenge, response = captcha\_settings.get\_challenge(generator)()  
 store = cls.objects.create(challenge=challenge, response=response)  
  
 return store.hashkey  
  
 @classmethod  
 def pick(cls):  
 if not captcha\_settings.CAPTCHA\_GET\_FROM\_POOL:  
 return cls.generate\_key()  
  
 def fallback():  
 logger.error("Couldn't get a captcha from pool, generating")  
 return cls.generate\_key()  
  
 # Pick up a random item from pool  
 minimum\_expiration = timezone.now() + datetime.timedelta(  
 minutes=int(captcha\_settings.CAPTCHA\_GET\_FROM\_POOL\_TIMEOUT)  
 )  
 store = (  
 cls.objects.filter(expiration\_\_gt=minimum\_expiration).order\_by("?").first()  
 )  
  
 return (store and store.hashkey) or fallback()  
  
 @classmethod  
 def create\_pool(cls, count=1000):  
 assert count > 0  
 while count > 0:  
 cls.generate\_key()  
 count -= 1

## File: venv/lib/python3.12/site-packages/charset\_normalizer/models.py

from encodings.aliases import aliases  
from hashlib import sha256  
from json import dumps  
from typing import Any, Dict, Iterator, List, Optional, Tuple, Union  
  
from .constant import TOO\_BIG\_SEQUENCE  
from .utils import iana\_name, is\_multi\_byte\_encoding, unicode\_range  
  
  
class CharsetMatch:  
 def \_\_init\_\_(  
 self,  
 payload: bytes,  
 guessed\_encoding: str,  
 mean\_mess\_ratio: float,  
 has\_sig\_or\_bom: bool,  
 languages: "CoherenceMatches",  
 decoded\_payload: Optional[str] = None,  
 ):  
 self.\_payload: bytes = payload  
  
 self.\_encoding: str = guessed\_encoding  
 self.\_mean\_mess\_ratio: float = mean\_mess\_ratio  
 self.\_languages: CoherenceMatches = languages  
 self.\_has\_sig\_or\_bom: bool = has\_sig\_or\_bom  
 self.\_unicode\_ranges: Optional[List[str]] = None  
  
 self.\_leaves: List[CharsetMatch] = []  
 self.\_mean\_coherence\_ratio: float = 0.0  
  
 self.\_output\_payload: Optional[bytes] = None  
 self.\_output\_encoding: Optional[str] = None  
  
 self.\_string: Optional[str] = decoded\_payload  
  
 def \_\_eq\_\_(self, other: object) -> bool:  
 if not isinstance(other, CharsetMatch):  
 raise TypeError(  
 "\_\_eq\_\_ cannot be invoked on {} and {}.".format(  
 str(other.\_\_class\_\_), str(self.\_\_class\_\_)  
 )  
 )  
 return self.encoding == other.encoding and self.fingerprint == other.fingerprint  
  
 def \_\_lt\_\_(self, other: object) -> bool:  
 """  
 Implemented to make sorted available upon CharsetMatches items.  
 """  
 if not isinstance(other, CharsetMatch):  
 raise ValueError  
  
 chaos\_difference: float = abs(self.chaos - other.chaos)  
 coherence\_difference: float = abs(self.coherence - other.coherence)  
  
 # Below 1% difference --> Use Coherence  
 if chaos\_difference < 0.01 and coherence\_difference > 0.02:  
 return self.coherence > other.coherence  
 elif chaos\_difference < 0.01 and coherence\_difference <= 0.02:  
 # When having a difficult decision, use the result that decoded as many multi-byte as possible.  
 # preserve RAM usage!  
 if len(self.\_payload) >= TOO\_BIG\_SEQUENCE:  
 return self.chaos < other.chaos  
 return self.multi\_byte\_usage > other.multi\_byte\_usage  
  
 return self.chaos < other.chaos  
  
 @property  
 def multi\_byte\_usage(self) -> float:  
 return 1.0 - (len(str(self)) / len(self.raw))  
  
 def \_\_str\_\_(self) -> str:  
 # Lazy Str Loading  
 if self.\_string is None:  
 self.\_string = str(self.\_payload, self.\_encoding, "strict")  
 return self.\_string  
  
 def \_\_repr\_\_(self) -> str:  
 return "<CharsetMatch '{}' bytes({})>".format(self.encoding, self.fingerprint)  
  
 def add\_submatch(self, other: "CharsetMatch") -> None:  
 if not isinstance(other, CharsetMatch) or other == self:  
 raise ValueError(  
 "Unable to add instance <{}> as a submatch of a CharsetMatch".format(  
 other.\_\_class\_\_  
 )  
 )  
  
 other.\_string = None # Unload RAM usage; dirty trick.  
 self.\_leaves.append(other)  
  
 @property  
 def encoding(self) -> str:  
 return self.\_encoding  
  
 @property  
 def encoding\_aliases(self) -> List[str]:  
 """  
 Encoding name are known by many name, using this could help when searching for IBM855 when it's listed as CP855.  
 """  
 also\_known\_as: List[str] = []  
 for u, p in aliases.items():  
 if self.encoding == u:  
 also\_known\_as.append(p)  
 elif self.encoding == p:  
 also\_known\_as.append(u)  
 return also\_known\_as  
  
 @property  
 def bom(self) -> bool:  
 return self.\_has\_sig\_or\_bom  
  
 @property  
 def byte\_order\_mark(self) -> bool:  
 return self.\_has\_sig\_or\_bom  
  
 @property  
 def languages(self) -> List[str]:  
 """  
 Return the complete list of possible languages found in decoded sequence.  
 Usually not really useful. Returned list may be empty even if 'language' property return something != 'Unknown'.  
 """  
 return [e[0] for e in self.\_languages]  
  
 @property  
 def language(self) -> str:  
 """  
 Most probable language found in decoded sequence. If none were detected or inferred, the property will return  
 "Unknown".  
 """  
 if not self.\_languages:  
 # Trying to infer the language based on the given encoding  
 # Its either English or we should not pronounce ourselves in certain cases.  
 if "ascii" in self.could\_be\_from\_charset:  
 return "English"  
  
 # doing it there to avoid circular import  
 from charset\_normalizer.cd import encoding\_languages, mb\_encoding\_languages  
  
 languages = (  
 mb\_encoding\_languages(self.encoding)  
 if is\_multi\_byte\_encoding(self.encoding)  
 else encoding\_languages(self.encoding)  
 )  
  
 if len(languages) == 0 or "Latin Based" in languages:  
 return "Unknown"  
  
 return languages[0]  
  
 return self.\_languages[0][0]  
  
 @property  
 def chaos(self) -> float:  
 return self.\_mean\_mess\_ratio  
  
 @property  
 def coherence(self) -> float:  
 if not self.\_languages:  
 return 0.0  
 return self.\_languages[0][1]  
  
 @property  
 def percent\_chaos(self) -> float:  
 return round(self.chaos \* 100, ndigits=3)  
  
 @property  
 def percent\_coherence(self) -> float:  
 return round(self.coherence \* 100, ndigits=3)  
  
 @property  
 def raw(self) -> bytes:  
 """  
 Original untouched bytes.  
 """  
 return self.\_payload  
  
 @property  
 def submatch(self) -> List["CharsetMatch"]:  
 return self.\_leaves  
  
 @property  
 def has\_submatch(self) -> bool:  
 return len(self.\_leaves) > 0  
  
 @property  
 def alphabets(self) -> List[str]:  
 if self.\_unicode\_ranges is not None:  
 return self.\_unicode\_ranges  
 # list detected ranges  
 detected\_ranges: List[Optional[str]] = [  
 unicode\_range(char) for char in str(self)  
 ]  
 # filter and sort  
 self.\_unicode\_ranges = sorted(list({r for r in detected\_ranges if r}))  
 return self.\_unicode\_ranges  
  
 @property  
 def could\_be\_from\_charset(self) -> List[str]:  
 """  
 The complete list of encoding that output the exact SAME str result and therefore could be the originating  
 encoding.  
 This list does include the encoding available in property 'encoding'.  
 """  
 return [self.\_encoding] + [m.encoding for m in self.\_leaves]  
  
 def output(self, encoding: str = "utf\_8") -> bytes:  
 """  
 Method to get re-encoded bytes payload using given target encoding. Default to UTF-8.  
 Any errors will be simply ignored by the encoder NOT replaced.  
 """  
 if self.\_output\_encoding is None or self.\_output\_encoding != encoding:  
 self.\_output\_encoding = encoding  
 self.\_output\_payload = str(self).encode(encoding, "replace")  
  
 return self.\_output\_payload # type: ignore  
  
 @property  
 def fingerprint(self) -> str:  
 """  
 Retrieve the unique SHA256 computed using the transformed (re-encoded) payload. Not the original one.  
 """  
 return sha256(self.output()).hexdigest()  
  
  
class CharsetMatches:  
 """  
 Container with every CharsetMatch items ordered by default from most probable to the less one.  
 Act like a list(iterable) but does not implements all related methods.  
 """  
  
 def \_\_init\_\_(self, results: Optional[List[CharsetMatch]] = None):  
 self.\_results: List[CharsetMatch] = sorted(results) if results else []  
  
 def \_\_iter\_\_(self) -> Iterator[CharsetMatch]:  
 yield from self.\_results  
  
 def \_\_getitem\_\_(self, item: Union[int, str]) -> CharsetMatch:  
 """  
 Retrieve a single item either by its position or encoding name (alias may be used here).  
 Raise KeyError upon invalid index or encoding not present in results.  
 """  
 if isinstance(item, int):  
 return self.\_results[item]  
 if isinstance(item, str):  
 item = iana\_name(item, False)  
 for result in self.\_results:  
 if item in result.could\_be\_from\_charset:  
 return result  
 raise KeyError  
  
 def \_\_len\_\_(self) -> int:  
 return len(self.\_results)  
  
 def \_\_bool\_\_(self) -> bool:  
 return len(self.\_results) > 0  
  
 def append(self, item: CharsetMatch) -> None:  
 """  
 Insert a single match. Will be inserted accordingly to preserve sort.  
 Can be inserted as a submatch.  
 """  
 if not isinstance(item, CharsetMatch):  
 raise ValueError(  
 "Cannot append instance '{}' to CharsetMatches".format(  
 str(item.\_\_class\_\_)  
 )  
 )  
 # We should disable the submatch factoring when the input file is too heavy (conserve RAM usage)  
 if len(item.raw) <= TOO\_BIG\_SEQUENCE:  
 for match in self.\_results:  
 if match.fingerprint == item.fingerprint and match.chaos == item.chaos:  
 match.add\_submatch(item)  
 return  
 self.\_results.append(item)  
 self.\_results = sorted(self.\_results)  
  
 def best(self) -> Optional["CharsetMatch"]:  
 """  
 Simply return the first match. Strict equivalent to matches[0].  
 """  
 if not self.\_results:  
 return None  
 return self.\_results[0]  
  
 def first(self) -> Optional["CharsetMatch"]:  
 """  
 Redundant method, call the method best(). Kept for BC reasons.  
 """  
 return self.best()  
  
  
CoherenceMatch = Tuple[str, float]  
CoherenceMatches = List[CoherenceMatch]  
  
  
class CliDetectionResult:  
 def \_\_init\_\_(  
 self,  
 path: str,  
 encoding: Optional[str],  
 encoding\_aliases: List[str],  
 alternative\_encodings: List[str],  
 language: str,  
 alphabets: List[str],  
 has\_sig\_or\_bom: bool,  
 chaos: float,  
 coherence: float,  
 unicode\_path: Optional[str],  
 is\_preferred: bool,  
 ):  
 self.path: str = path  
 self.unicode\_path: Optional[str] = unicode\_path  
 self.encoding: Optional[str] = encoding  
 self.encoding\_aliases: List[str] = encoding\_aliases  
 self.alternative\_encodings: List[str] = alternative\_encodings  
 self.language: str = language  
 self.alphabets: List[str] = alphabets  
 self.has\_sig\_or\_bom: bool = has\_sig\_or\_bom  
 self.chaos: float = chaos  
 self.coherence: float = coherence  
 self.is\_preferred: bool = is\_preferred  
  
 @property  
 def \_\_dict\_\_(self) -> Dict[str, Any]: # type: ignore  
 return {  
 "path": self.path,  
 "encoding": self.encoding,  
 "encoding\_aliases": self.encoding\_aliases,  
 "alternative\_encodings": self.alternative\_encodings,  
 "language": self.language,  
 "alphabets": self.alphabets,  
 "has\_sig\_or\_bom": self.has\_sig\_or\_bom,  
 "chaos": self.chaos,  
 "coherence": self.coherence,  
 "unicode\_path": self.unicode\_path,  
 "is\_preferred": self.is\_preferred,  
 }  
  
 def to\_json(self) -> str:  
 return dumps(self.\_\_dict\_\_, ensure\_ascii=True, indent=4)

## File: venv/lib/python3.12/site-packages/django/contrib/admin/models.py

import json  
  
from django.conf import settings  
from django.contrib.admin.utils import quote  
from django.contrib.contenttypes.models import ContentType  
from django.db import models  
from django.urls import NoReverseMatch, reverse  
from django.utils import timezone  
from django.utils.text import get\_text\_list  
from django.utils.translation import gettext  
from django.utils.translation import gettext\_lazy as \_  
  
ADDITION = 1  
CHANGE = 2  
DELETION = 3  
  
ACTION\_FLAG\_CHOICES = [  
 (ADDITION, \_("Addition")),  
 (CHANGE, \_("Change")),  
 (DELETION, \_("Deletion")),  
]  
  
  
class LogEntryManager(models.Manager):  
 use\_in\_migrations = True  
  
 def log\_action(  
 self,  
 user\_id,  
 content\_type\_id,  
 object\_id,  
 object\_repr,  
 action\_flag,  
 change\_message="",  
 ):  
 if isinstance(change\_message, list):  
 change\_message = json.dumps(change\_message)  
 return self.model.objects.create(  
 user\_id=user\_id,  
 content\_type\_id=content\_type\_id,  
 object\_id=str(object\_id),  
 object\_repr=object\_repr[:200],  
 action\_flag=action\_flag,  
 change\_message=change\_message,  
 )  
  
  
class LogEntry(models.Model):  
 action\_time = models.DateTimeField(  
 \_("action time"),  
 default=timezone.now,  
 editable=False,  
 )  
 user = models.ForeignKey(  
 settings.AUTH\_USER\_MODEL,  
 models.CASCADE,  
 verbose\_name=\_("user"),  
 )  
 content\_type = models.ForeignKey(  
 ContentType,  
 models.SET\_NULL,  
 verbose\_name=\_("content type"),  
 blank=True,  
 null=True,  
 )  
 object\_id = models.TextField(\_("object id"), blank=True, null=True)  
 # Translators: 'repr' means representation  
 # (https://docs.python.org/library/functions.html#repr)  
 object\_repr = models.CharField(\_("object repr"), max\_length=200)  
 action\_flag = models.PositiveSmallIntegerField(  
 \_("action flag"), choices=ACTION\_FLAG\_CHOICES  
 )  
 # change\_message is either a string or a JSON structure  
 change\_message = models.TextField(\_("change message"), blank=True)  
  
 objects = LogEntryManager()  
  
 class Meta:  
 verbose\_name = \_("log entry")  
 verbose\_name\_plural = \_("log entries")  
 db\_table = "django\_admin\_log"  
 ordering = ["-action\_time"]  
  
 def \_\_repr\_\_(self):  
 return str(self.action\_time)  
  
 def \_\_str\_\_(self):  
 if self.is\_addition():  
 return gettext("Added “%(object)s”.") % {"object": self.object\_repr}  
 elif self.is\_change():  
 return gettext("Changed “%(object)s” — %(changes)s") % {  
 "object": self.object\_repr,  
 "changes": self.get\_change\_message(),  
 }  
 elif self.is\_deletion():  
 return gettext("Deleted “%(object)s.”") % {"object": self.object\_repr}  
  
 return gettext("LogEntry Object")  
  
 def is\_addition(self):  
 return self.action\_flag == ADDITION  
  
 def is\_change(self):  
 return self.action\_flag == CHANGE  
  
 def is\_deletion(self):  
 return self.action\_flag == DELETION  
  
 def get\_change\_message(self):  
 """  
 If self.change\_message is a JSON structure, interpret it as a change  
 string, properly translated.  
 """  
 if self.change\_message and self.change\_message[0] == "[":  
 try:  
 change\_message = json.loads(self.change\_message)  
 except json.JSONDecodeError:  
 return self.change\_message  
 messages = []  
 for sub\_message in change\_message:  
 if "added" in sub\_message:  
 if sub\_message["added"]:  
 sub\_message["added"]["name"] = gettext(  
 sub\_message["added"]["name"]  
 )  
 messages.append(  
 gettext("Added {name} “{object}”.").format(  
 \*\*sub\_message["added"]  
 )  
 )  
 else:  
 messages.append(gettext("Added."))  
  
 elif "changed" in sub\_message:  
 sub\_message["changed"]["fields"] = get\_text\_list(  
 [  
 gettext(field\_name)  
 for field\_name in sub\_message["changed"]["fields"]  
 ],  
 gettext("and"),  
 )  
 if "name" in sub\_message["changed"]:  
 sub\_message["changed"]["name"] = gettext(  
 sub\_message["changed"]["name"]  
 )  
 messages.append(  
 gettext("Changed {fields} for {name} “{object}”.").format(  
 \*\*sub\_message["changed"]  
 )  
 )  
 else:  
 messages.append(  
 gettext("Changed {fields}.").format(  
 \*\*sub\_message["changed"]  
 )  
 )  
  
 elif "deleted" in sub\_message:  
 sub\_message["deleted"]["name"] = gettext(  
 sub\_message["deleted"]["name"]  
 )  
 messages.append(  
 gettext("Deleted {name} “{object}”.").format(  
 \*\*sub\_message["deleted"]  
 )  
 )  
  
 change\_message = " ".join(msg[0].upper() + msg[1:] for msg in messages)  
 return change\_message or gettext("No fields changed.")  
 else:  
 return self.change\_message  
  
 def get\_edited\_object(self):  
 """Return the edited object represented by this log entry."""  
 return self.content\_type.get\_object\_for\_this\_type(pk=self.object\_id)  
  
 def get\_admin\_url(self):  
 """  
 Return the admin URL to edit the object represented by this log entry.  
 """  
 if self.content\_type and self.object\_id:  
 url\_name = "admin:%s\_%s\_change" % (  
 self.content\_type.app\_label,  
 self.content\_type.model,  
 )  
 try:  
 return reverse(url\_name, args=(quote(self.object\_id),))  
 except NoReverseMatch:  
 pass  
 return None

## File: venv/lib/python3.12/site-packages/django/contrib/auth/models.py

from django.apps import apps  
from django.contrib import auth  
from django.contrib.auth.base\_user import AbstractBaseUser, BaseUserManager  
from django.contrib.auth.hashers import make\_password  
from django.contrib.contenttypes.models import ContentType  
from django.core.exceptions import PermissionDenied  
from django.core.mail import send\_mail  
from django.db import models  
from django.db.models.manager import EmptyManager  
from django.utils import timezone  
from django.utils.itercompat import is\_iterable  
from django.utils.translation import gettext\_lazy as \_  
  
from .validators import UnicodeUsernameValidator  
  
  
def update\_last\_login(sender, user, \*\*kwargs):  
 """  
 A signal receiver which updates the last\_login date for  
 the user logging in.  
 """  
 user.last\_login = timezone.now()  
 user.save(update\_fields=["last\_login"])  
  
  
class PermissionManager(models.Manager):  
 use\_in\_migrations = True  
  
 def get\_by\_natural\_key(self, codename, app\_label, model):  
 return self.get(  
 codename=codename,  
 content\_type=ContentType.objects.db\_manager(self.db).get\_by\_natural\_key(  
 app\_label, model  
 ),  
 )  
  
  
class Permission(models.Model):  
 """  
 The permissions system provides a way to assign permissions to specific  
 users and groups of users.  
  
 The permission system is used by the Django admin site, but may also be  
 useful in your own code. The Django admin site uses permissions as follows:  
  
 - The "add" permission limits the user's ability to view the "add" form  
 and add an object.  
 - The "change" permission limits a user's ability to view the change  
 list, view the "change" form and change an object.  
 - The "delete" permission limits the ability to delete an object.  
 - The "view" permission limits the ability to view an object.  
  
 Permissions are set globally per type of object, not per specific object  
 instance. It is possible to say "Mary may change news stories," but it's  
 not currently possible to say "Mary may change news stories, but only the  
 ones she created herself" or "Mary may only change news stories that have a  
 certain status or publication date."  
  
 The permissions listed above are automatically created for each model.  
 """  
  
 name = models.CharField(\_("name"), max\_length=255)  
 content\_type = models.ForeignKey(  
 ContentType,  
 models.CASCADE,  
 verbose\_name=\_("content type"),  
 )  
 codename = models.CharField(\_("codename"), max\_length=100)  
  
 objects = PermissionManager()  
  
 class Meta:  
 verbose\_name = \_("permission")  
 verbose\_name\_plural = \_("permissions")  
 unique\_together = [["content\_type", "codename"]]  
 ordering = ["content\_type\_\_app\_label", "content\_type\_\_model", "codename"]  
  
 def \_\_str\_\_(self):  
 return "%s | %s" % (self.content\_type, self.name)  
  
 def natural\_key(self):  
 return (self.codename,) + self.content\_type.natural\_key()  
  
 natural\_key.dependencies = ["contenttypes.contenttype"]  
  
  
class GroupManager(models.Manager):  
 """  
 The manager for the auth's Group model.  
 """  
  
 use\_in\_migrations = True  
  
 def get\_by\_natural\_key(self, name):  
 return self.get(name=name)  
  
  
class Group(models.Model):  
 """  
 Groups are a generic way of categorizing users to apply permissions, or  
 some other label, to those users. A user can belong to any number of  
 groups.  
  
 A user in a group automatically has all the permissions granted to that  
 group. For example, if the group 'Site editors' has the permission  
 can\_edit\_home\_page, any user in that group will have that permission.  
  
 Beyond permissions, groups are a convenient way to categorize users to  
 apply some label, or extended functionality, to them. For example, you  
 could create a group 'Special users', and you could write code that would  
 do special things to those users -- such as giving them access to a  
 members-only portion of your site, or sending them members-only email  
 messages.  
 """  
  
 name = models.CharField(\_("name"), max\_length=150, unique=True)  
 permissions = models.ManyToManyField(  
 Permission,  
 verbose\_name=\_("permissions"),  
 blank=True,  
 )  
  
 objects = GroupManager()  
  
 class Meta:  
 verbose\_name = \_("group")  
 verbose\_name\_plural = \_("groups")  
  
 def \_\_str\_\_(self):  
 return self.name  
  
 def natural\_key(self):  
 return (self.name,)  
  
  
class UserManager(BaseUserManager):  
 use\_in\_migrations = True  
  
 def \_create\_user(self, username, email, password, \*\*extra\_fields):  
 """  
 Create and save a user with the given username, email, and password.  
 """  
 if not username:  
 raise ValueError("The given username must be set")  
 email = self.normalize\_email(email)  
 # Lookup the real model class from the global app registry so this  
 # manager method can be used in migrations. This is fine because  
 # managers are by definition working on the real model.  
 GlobalUserModel = apps.get\_model(  
 self.model.\_meta.app\_label, self.model.\_meta.object\_name  
 )  
 username = GlobalUserModel.normalize\_username(username)  
 user = self.model(username=username, email=email, \*\*extra\_fields)  
 user.password = make\_password(password)  
 user.save(using=self.\_db)  
 return user  
  
 def create\_user(self, username, email=None, password=None, \*\*extra\_fields):  
 extra\_fields.setdefault("is\_staff", False)  
 extra\_fields.setdefault("is\_superuser", False)  
 return self.\_create\_user(username, email, password, \*\*extra\_fields)  
  
 def create\_superuser(self, username, email=None, password=None, \*\*extra\_fields):  
 extra\_fields.setdefault("is\_staff", True)  
 extra\_fields.setdefault("is\_superuser", True)  
  
 if extra\_fields.get("is\_staff") is not True:  
 raise ValueError("Superuser must have is\_staff=True.")  
 if extra\_fields.get("is\_superuser") is not True:  
 raise ValueError("Superuser must have is\_superuser=True.")  
  
 return self.\_create\_user(username, email, password, \*\*extra\_fields)  
  
 def with\_perm(  
 self, perm, is\_active=True, include\_superusers=True, backend=None, obj=None  
 ):  
 if backend is None:  
 backends = auth.\_get\_backends(return\_tuples=True)  
 if len(backends) == 1:  
 backend, \_ = backends[0]  
 else:  
 raise ValueError(  
 "You have multiple authentication backends configured and "  
 "therefore must provide the `backend` argument."  
 )  
 elif not isinstance(backend, str):  
 raise TypeError(  
 "backend must be a dotted import path string (got %r)." % backend  
 )  
 else:  
 backend = auth.load\_backend(backend)  
 if hasattr(backend, "with\_perm"):  
 return backend.with\_perm(  
 perm,  
 is\_active=is\_active,  
 include\_superusers=include\_superusers,  
 obj=obj,  
 )  
 return self.none()  
  
  
# A few helper functions for common logic between User and AnonymousUser.  
def \_user\_get\_permissions(user, obj, from\_name):  
 permissions = set()  
 name = "get\_%s\_permissions" % from\_name  
 for backend in auth.get\_backends():  
 if hasattr(backend, name):  
 permissions.update(getattr(backend, name)(user, obj))  
 return permissions  
  
  
def \_user\_has\_perm(user, perm, obj):  
 """  
 A backend can raise `PermissionDenied` to short-circuit permission checking.  
 """  
 for backend in auth.get\_backends():  
 if not hasattr(backend, "has\_perm"):  
 continue  
 try:  
 if backend.has\_perm(user, perm, obj):  
 return True  
 except PermissionDenied:  
 return False  
 return False  
  
  
def \_user\_has\_module\_perms(user, app\_label):  
 """  
 A backend can raise `PermissionDenied` to short-circuit permission checking.  
 """  
 for backend in auth.get\_backends():  
 if not hasattr(backend, "has\_module\_perms"):  
 continue  
 try:  
 if backend.has\_module\_perms(user, app\_label):  
 return True  
 except PermissionDenied:  
 return False  
 return False  
  
  
class PermissionsMixin(models.Model):  
 """  
 Add the fields and methods necessary to support the Group and Permission  
 models using the ModelBackend.  
 """  
  
 is\_superuser = models.BooleanField(  
 \_("superuser status"),  
 default=False,  
 help\_text=\_(  
 "Designates that this user has all permissions without "  
 "explicitly assigning them."  
 ),  
 )  
 groups = models.ManyToManyField(  
 Group,  
 verbose\_name=\_("groups"),  
 blank=True,  
 help\_text=\_(  
 "The groups this user belongs to. A user will get all permissions "  
 "granted to each of their groups."  
 ),  
 related\_name="user\_set",  
 related\_query\_name="user",  
 )  
 user\_permissions = models.ManyToManyField(  
 Permission,  
 verbose\_name=\_("user permissions"),  
 blank=True,  
 help\_text=\_("Specific permissions for this user."),  
 related\_name="user\_set",  
 related\_query\_name="user",  
 )  
  
 class Meta:  
 abstract = True  
  
 def get\_user\_permissions(self, obj=None):  
 """  
 Return a list of permission strings that this user has directly.  
 Query all available auth backends. If an object is passed in,  
 return only permissions matching this object.  
 """  
 return \_user\_get\_permissions(self, obj, "user")  
  
 def get\_group\_permissions(self, obj=None):  
 """  
 Return a list of permission strings that this user has through their  
 groups. Query all available auth backends. If an object is passed in,  
 return only permissions matching this object.  
 """  
 return \_user\_get\_permissions(self, obj, "group")  
  
 def get\_all\_permissions(self, obj=None):  
 return \_user\_get\_permissions(self, obj, "all")  
  
 def has\_perm(self, perm, obj=None):  
 """  
 Return True if the user has the specified permission. Query all  
 available auth backends, but return immediately if any backend returns  
 True. Thus, a user who has permission from a single auth backend is  
 assumed to have permission in general. If an object is provided, check  
 permissions for that object.  
 """  
 # Active superusers have all permissions.  
 if self.is\_active and self.is\_superuser:  
 return True  
  
 # Otherwise we need to check the backends.  
 return \_user\_has\_perm(self, perm, obj)  
  
 def has\_perms(self, perm\_list, obj=None):  
 """  
 Return True if the user has each of the specified permissions. If  
 object is passed, check if the user has all required perms for it.  
 """  
 if not is\_iterable(perm\_list) or isinstance(perm\_list, str):  
 raise ValueError("perm\_list must be an iterable of permissions.")  
 return all(self.has\_perm(perm, obj) for perm in perm\_list)  
  
 def has\_module\_perms(self, app\_label):  
 """  
 Return True if the user has any permissions in the given app label.  
 Use similar logic as has\_perm(), above.  
 """  
 # Active superusers have all permissions.  
 if self.is\_active and self.is\_superuser:  
 return True  
  
 return \_user\_has\_module\_perms(self, app\_label)  
  
  
class AbstractUser(AbstractBaseUser, PermissionsMixin):  
 """  
 An abstract base class implementing a fully featured User model with  
 admin-compliant permissions.  
  
 Username and password are required. Other fields are optional.  
 """  
  
 username\_validator = UnicodeUsernameValidator()  
  
 username = models.CharField(  
 \_("username"),  
 max\_length=150,  
 unique=True,  
 help\_text=\_(  
 "Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only."  
 ),  
 validators=[username\_validator],  
 error\_messages={  
 "unique": \_("A user with that username already exists."),  
 },  
 )  
 first\_name = models.CharField(\_("first name"), max\_length=150, blank=True)  
 last\_name = models.CharField(\_("last name"), max\_length=150, blank=True)  
 email = models.EmailField(\_("email address"), blank=True)  
 is\_staff = models.BooleanField(  
 \_("staff status"),  
 default=False,  
 help\_text=\_("Designates whether the user can log into this admin site."),  
 )  
 is\_active = models.BooleanField(  
 \_("active"),  
 default=True,  
 help\_text=\_(  
 "Designates whether this user should be treated as active. "  
 "Unselect this instead of deleting accounts."  
 ),  
 )  
 date\_joined = models.DateTimeField(\_("date joined"), default=timezone.now)  
  
 objects = UserManager()  
  
 EMAIL\_FIELD = "email"  
 USERNAME\_FIELD = "username"  
 REQUIRED\_FIELDS = ["email"]  
  
 class Meta:  
 verbose\_name = \_("user")  
 verbose\_name\_plural = \_("users")  
 abstract = True  
  
 def clean(self):  
 super().clean()  
 self.email = self.\_\_class\_\_.objects.normalize\_email(self.email)  
  
 def get\_full\_name(self):  
 """  
 Return the first\_name plus the last\_name, with a space in between.  
 """  
 full\_name = "%s %s" % (self.first\_name, self.last\_name)  
 return full\_name.strip()  
  
 def get\_short\_name(self):  
 """Return the short name for the user."""  
 return self.first\_name  
  
 def email\_user(self, subject, message, from\_email=None, \*\*kwargs):  
 """Send an email to this user."""  
 send\_mail(subject, message, from\_email, [self.email], \*\*kwargs)  
  
  
class User(AbstractUser):  
 """  
 Users within the Django authentication system are represented by this  
 model.  
  
 Username and password are required. Other fields are optional.  
 """  
  
 class Meta(AbstractUser.Meta):  
 swappable = "AUTH\_USER\_MODEL"  
  
  
class AnonymousUser:  
 id = None  
 pk = None  
 username = ""  
 is\_staff = False  
 is\_active = False  
 is\_superuser = False  
 \_groups = EmptyManager(Group)  
 \_user\_permissions = EmptyManager(Permission)  
  
 def \_\_str\_\_(self):  
 return "AnonymousUser"  
  
 def \_\_eq\_\_(self, other):  
 return isinstance(other, self.\_\_class\_\_)  
  
 def \_\_hash\_\_(self):  
 return 1 # instances always return the same hash value  
  
 def \_\_int\_\_(self):  
 raise TypeError(  
 "Cannot cast AnonymousUser to int. Are you trying to use it in place of "  
 "User?"  
 )  
  
 def save(self):  
 raise NotImplementedError(  
 "Django doesn't provide a DB representation for AnonymousUser."  
 )  
  
 def delete(self):  
 raise NotImplementedError(  
 "Django doesn't provide a DB representation for AnonymousUser."  
 )  
  
 def set\_password(self, raw\_password):  
 raise NotImplementedError(  
 "Django doesn't provide a DB representation for AnonymousUser."  
 )  
  
 def check\_password(self, raw\_password):  
 raise NotImplementedError(  
 "Django doesn't provide a DB representation for AnonymousUser."  
 )  
  
 @property  
 def groups(self):  
 return self.\_groups  
  
 @property  
 def user\_permissions(self):  
 return self.\_user\_permissions  
  
 def get\_user\_permissions(self, obj=None):  
 return \_user\_get\_permissions(self, obj, "user")  
  
 def get\_group\_permissions(self, obj=None):  
 return set()  
  
 def get\_all\_permissions(self, obj=None):  
 return \_user\_get\_permissions(self, obj, "all")  
  
 def has\_perm(self, perm, obj=None):  
 return \_user\_has\_perm(self, perm, obj=obj)  
  
 def has\_perms(self, perm\_list, obj=None):  
 if not is\_iterable(perm\_list) or isinstance(perm\_list, str):  
 raise ValueError("perm\_list must be an iterable of permissions.")  
 return all(self.has\_perm(perm, obj) for perm in perm\_list)  
  
 def has\_module\_perms(self, module):  
 return \_user\_has\_module\_perms(self, module)  
  
 @property  
 def is\_anonymous(self):  
 return True  
  
 @property  
 def is\_authenticated(self):  
 return False  
  
 def get\_username(self):  
 return self.username

## File: venv/lib/python3.12/site-packages/django/contrib/contenttypes/models.py

from collections import defaultdict  
  
from django.apps import apps  
from django.db import models  
from django.db.models import Q  
from django.utils.translation import gettext\_lazy as \_  
  
  
class ContentTypeManager(models.Manager):  
 use\_in\_migrations = True  
  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
 # Cache shared by all the get\_for\_\* methods to speed up  
 # ContentType retrieval.  
 self.\_cache = {}  
  
 def get\_by\_natural\_key(self, app\_label, model):  
 try:  
 ct = self.\_cache[self.db][(app\_label, model)]  
 except KeyError:  
 ct = self.get(app\_label=app\_label, model=model)  
 self.\_add\_to\_cache(self.db, ct)  
 return ct  
  
 def \_get\_opts(self, model, for\_concrete\_model):  
 if for\_concrete\_model:  
 model = model.\_meta.concrete\_model  
 return model.\_meta  
  
 def \_get\_from\_cache(self, opts):  
 key = (opts.app\_label, opts.model\_name)  
 return self.\_cache[self.db][key]  
  
 def get\_for\_model(self, model, for\_concrete\_model=True):  
 """  
 Return the ContentType object for a given model, creating the  
 ContentType if necessary. Lookups are cached so that subsequent lookups  
 for the same model don't hit the database.  
 """  
 opts = self.\_get\_opts(model, for\_concrete\_model)  
 try:  
 return self.\_get\_from\_cache(opts)  
 except KeyError:  
 pass  
  
 # The ContentType entry was not found in the cache, therefore we  
 # proceed to load or create it.  
 try:  
 # Start with get() and not get\_or\_create() in order to use  
 # the db\_for\_read (see #20401).  
 ct = self.get(app\_label=opts.app\_label, model=opts.model\_name)  
 except self.model.DoesNotExist:  
 # Not found in the database; we proceed to create it. This time  
 # use get\_or\_create to take care of any race conditions.  
 ct, created = self.get\_or\_create(  
 app\_label=opts.app\_label,  
 model=opts.model\_name,  
 )  
 self.\_add\_to\_cache(self.db, ct)  
 return ct  
  
 def get\_for\_models(self, \*models, for\_concrete\_models=True):  
 """  
 Given \*models, return a dictionary mapping {model: content\_type}.  
 """  
 results = {}  
 # Models that aren't already in the cache grouped by app labels.  
 needed\_models = defaultdict(set)  
 # Mapping of opts to the list of models requiring it.  
 needed\_opts = defaultdict(list)  
 for model in models:  
 opts = self.\_get\_opts(model, for\_concrete\_models)  
 try:  
 ct = self.\_get\_from\_cache(opts)  
 except KeyError:  
 needed\_models[opts.app\_label].add(opts.model\_name)  
 needed\_opts[opts].append(model)  
 else:  
 results[model] = ct  
 if needed\_opts:  
 # Lookup required content types from the DB.  
 condition = Q(  
 \*(  
 Q(("app\_label", app\_label), ("model\_\_in", models))  
 for app\_label, models in needed\_models.items()  
 ),  
 \_connector=Q.OR,  
 )  
 cts = self.filter(condition)  
 for ct in cts:  
 opts\_models = needed\_opts.pop(  
 ct.\_meta.apps.get\_model(ct.app\_label, ct.model).\_meta, []  
 )  
 for model in opts\_models:  
 results[model] = ct  
 self.\_add\_to\_cache(self.db, ct)  
 # Create content types that weren't in the cache or DB.  
 for opts, opts\_models in needed\_opts.items():  
 ct = self.create(  
 app\_label=opts.app\_label,  
 model=opts.model\_name,  
 )  
 self.\_add\_to\_cache(self.db, ct)  
 for model in opts\_models:  
 results[model] = ct  
 return results  
  
 def get\_for\_id(self, id):  
 """  
 Lookup a ContentType by ID. Use the same shared cache as get\_for\_model  
 (though ContentTypes are not created on-the-fly by get\_by\_id).  
 """  
 try:  
 ct = self.\_cache[self.db][id]  
 except KeyError:  
 # This could raise a DoesNotExist; that's correct behavior and will  
 # make sure that only correct ctypes get stored in the cache dict.  
 ct = self.get(pk=id)  
 self.\_add\_to\_cache(self.db, ct)  
 return ct  
  
 def clear\_cache(self):  
 """  
 Clear out the content-type cache.  
 """  
 self.\_cache.clear()  
  
 def \_add\_to\_cache(self, using, ct):  
 """Insert a ContentType into the cache."""  
 # Note it's possible for ContentType objects to be stale; model\_class()  
 # will return None. Hence, there is no reliance on  
 # model.\_meta.app\_label here, just using the model fields instead.  
 key = (ct.app\_label, ct.model)  
 self.\_cache.setdefault(using, {})[key] = ct  
 self.\_cache.setdefault(using, {})[ct.id] = ct  
  
  
class ContentType(models.Model):  
 app\_label = models.CharField(max\_length=100)  
 model = models.CharField(\_("python model class name"), max\_length=100)  
 objects = ContentTypeManager()  
  
 class Meta:  
 verbose\_name = \_("content type")  
 verbose\_name\_plural = \_("content types")  
 db\_table = "django\_content\_type"  
 unique\_together = [["app\_label", "model"]]  
  
 def \_\_str\_\_(self):  
 return self.app\_labeled\_name  
  
 @property  
 def name(self):  
 model = self.model\_class()  
 if not model:  
 return self.model  
 return str(model.\_meta.verbose\_name)  
  
 @property  
 def app\_labeled\_name(self):  
 model = self.model\_class()  
 if not model:  
 return self.model  
 return "%s | %s" % (model.\_meta.app\_label, model.\_meta.verbose\_name)  
  
 def model\_class(self):  
 """Return the model class for this type of content."""  
 try:  
 return apps.get\_model(self.app\_label, self.model)  
 except LookupError:  
 return None  
  
 def get\_object\_for\_this\_type(self, \*\*kwargs):  
 """  
 Return an object of this type for the keyword arguments given.  
 Basically, this is a proxy around this object\_type's get\_object() model  
 method. The ObjectNotExist exception, if thrown, will not be caught,  
 so code that calls this method should catch it.  
 """  
 return self.model\_class().\_base\_manager.using(self.\_state.db).get(\*\*kwargs)  
  
 def get\_all\_objects\_for\_this\_type(self, \*\*kwargs):  
 """  
 Return all objects of this type for the keyword arguments given.  
 """  
 return self.model\_class().\_base\_manager.using(self.\_state.db).filter(\*\*kwargs)  
  
 def natural\_key(self):  
 return (self.app\_label, self.model)

## File: venv/lib/python3.12/site-packages/django/contrib/flatpages/models.py

from django.contrib.sites.models import Site  
from django.db import models  
from django.urls import NoReverseMatch, get\_script\_prefix, reverse  
from django.utils.encoding import iri\_to\_uri  
from django.utils.translation import gettext\_lazy as \_  
  
  
class FlatPage(models.Model):  
 url = models.CharField(\_("URL"), max\_length=100, db\_index=True)  
 title = models.CharField(\_("title"), max\_length=200)  
 content = models.TextField(\_("content"), blank=True)  
 enable\_comments = models.BooleanField(\_("enable comments"), default=False)  
 template\_name = models.CharField(  
 \_("template name"),  
 max\_length=70,  
 blank=True,  
 help\_text=\_(  
 "Example: “flatpages/contact\_page.html”. If this isn’t provided, "  
 "the system will use “flatpages/default.html”."  
 ),  
 )  
 registration\_required = models.BooleanField(  
 \_("registration required"),  
 help\_text=\_(  
 "If this is checked, only logged-in users will be able to view the page."  
 ),  
 default=False,  
 )  
 sites = models.ManyToManyField(Site, verbose\_name=\_("sites"))  
  
 class Meta:  
 db\_table = "django\_flatpage"  
 verbose\_name = \_("flat page")  
 verbose\_name\_plural = \_("flat pages")  
 ordering = ["url"]  
  
 def \_\_str\_\_(self):  
 return "%s -- %s" % (self.url, self.title)  
  
 def get\_absolute\_url(self):  
 from .views import flatpage  
  
 for url in (self.url.lstrip("/"), self.url):  
 try:  
 return reverse(flatpage, kwargs={"url": url})  
 except NoReverseMatch:  
 pass  
 # Handle script prefix manually because we bypass reverse()  
 return iri\_to\_uri(get\_script\_prefix().rstrip("/") + self.url)

## File: venv/lib/python3.12/site-packages/django/contrib/gis/db/backends/base/models.py

from django.contrib.gis import gdal  
  
  
class SpatialRefSysMixin:  
 """  
 The SpatialRefSysMixin is a class used by the database-dependent  
 SpatialRefSys objects to reduce redundant code.  
 """  
  
 @property  
 def srs(self):  
 """  
 Return a GDAL SpatialReference object.  
 """  
 # TODO: Is caching really necessary here? Is complexity worth it?  
 if hasattr(self, "\_srs"):  
 # Returning a clone of the cached SpatialReference object.  
 return self.\_srs.clone()  
 else:  
 # Attempting to cache a SpatialReference object.  
  
 # Trying to get from WKT first.  
 try:  
 self.\_srs = gdal.SpatialReference(self.wkt)  
 return self.srs  
 except Exception as e:  
 msg = e  
  
 try:  
 self.\_srs = gdal.SpatialReference(self.proj4text)  
 return self.srs  
 except Exception as e:  
 msg = e  
  
 raise Exception(  
 "Could not get OSR SpatialReference from WKT: %s\nError:\n%s"  
 % (self.wkt, msg)  
 )  
  
 @property  
 def ellipsoid(self):  
 """  
 Return a tuple of the ellipsoid parameters:  
 (semimajor axis, semiminor axis, and inverse flattening).  
 """  
 return self.srs.ellipsoid  
  
 @property  
 def name(self):  
 "Return the projection name."  
 return self.srs.name  
  
 @property  
 def spheroid(self):  
 "Return the spheroid name for this spatial reference."  
 return self.srs["spheroid"]  
  
 @property  
 def datum(self):  
 "Return the datum for this spatial reference."  
 return self.srs["datum"]  
  
 @property  
 def projected(self):  
 "Is this Spatial Reference projected?"  
 return self.srs.projected  
  
 @property  
 def local(self):  
 "Is this Spatial Reference local?"  
 return self.srs.local  
  
 @property  
 def geographic(self):  
 "Is this Spatial Reference geographic?"  
 return self.srs.geographic  
  
 @property  
 def linear\_name(self):  
 "Return the linear units name."  
 return self.srs.linear\_name  
  
 @property  
 def linear\_units(self):  
 "Return the linear units."  
 return self.srs.linear\_units  
  
 @property  
 def angular\_name(self):  
 "Return the name of the angular units."  
 return self.srs.angular\_name  
  
 @property  
 def angular\_units(self):  
 "Return the angular units."  
 return self.srs.angular\_units  
  
 @property  
 def units(self):  
 "Return a tuple of the units and the name."  
 if self.projected or self.local:  
 return (self.linear\_units, self.linear\_name)  
 elif self.geographic:  
 return (self.angular\_units, self.angular\_name)  
 else:  
 return (None, None)  
  
 @classmethod  
 def get\_units(cls, wkt):  
 """  
 Return a tuple of (unit\_value, unit\_name) for the given WKT without  
 using any of the database fields.  
 """  
 return gdal.SpatialReference(wkt).units  
  
 @classmethod  
 def get\_spheroid(cls, wkt, string=True):  
 """  
 Class method used by GeometryField on initialization to  
 retrieve the `SPHEROID[..]` parameters from the given WKT.  
 """  
 srs = gdal.SpatialReference(wkt)  
 sphere\_params = srs.ellipsoid  
 sphere\_name = srs["spheroid"]  
  
 if not string:  
 return sphere\_name, sphere\_params  
 else:  
 # `string` parameter used to place in format acceptable by PostGIS  
 if len(sphere\_params) == 3:  
 radius, flattening = sphere\_params[0], sphere\_params[2]  
 else:  
 radius, flattening = sphere\_params  
 return 'SPHEROID["%s",%s,%s]' % (sphere\_name, radius, flattening)  
  
 def \_\_str\_\_(self):  
 """  
 Return the string representation, a 'pretty' OGC WKT.  
 """  
 return str(self.srs)

## File: venv/lib/python3.12/site-packages/django/contrib/gis/db/backends/oracle/models.py

"""  
 The GeometryColumns and SpatialRefSys models for the Oracle spatial  
 backend.  
  
 It should be noted that Oracle Spatial does not have database tables  
 named according to the OGC standard, so the closest analogs are used.  
 For example, the `USER\_SDO\_GEOM\_METADATA` is used for the GeometryColumns  
 model and the `SDO\_COORD\_REF\_SYS` is used for the SpatialRefSys model.  
"""  
from django.contrib.gis.db import models  
from django.contrib.gis.db.backends.base.models import SpatialRefSysMixin  
  
  
class OracleGeometryColumns(models.Model):  
 "Maps to the Oracle USER\_SDO\_GEOM\_METADATA table."  
 table\_name = models.CharField(max\_length=32)  
 column\_name = models.CharField(max\_length=1024)  
 srid = models.IntegerField(primary\_key=True)  
 # TODO: Add support for `diminfo` column (type MDSYS.SDO\_DIM\_ARRAY).  
  
 class Meta:  
 app\_label = "gis"  
 db\_table = "USER\_SDO\_GEOM\_METADATA"  
 managed = False  
  
 def \_\_str\_\_(self):  
 return "%s - %s (SRID: %s)" % (self.table\_name, self.column\_name, self.srid)  
  
 @classmethod  
 def table\_name\_col(cls):  
 """  
 Return the name of the metadata column used to store the feature table  
 name.  
 """  
 return "table\_name"  
  
 @classmethod  
 def geom\_col\_name(cls):  
 """  
 Return the name of the metadata column used to store the feature  
 geometry column.  
 """  
 return "column\_name"  
  
  
class OracleSpatialRefSys(models.Model, SpatialRefSysMixin):  
 "Maps to the Oracle MDSYS.CS\_SRS table."  
 cs\_name = models.CharField(max\_length=68)  
 srid = models.IntegerField(primary\_key=True)  
 auth\_srid = models.IntegerField()  
 auth\_name = models.CharField(max\_length=256)  
 wktext = models.CharField(max\_length=2046)  
 # Optional geometry representing the bounds of this coordinate  
 # system. By default, all are NULL in the table.  
 cs\_bounds = models.PolygonField(null=True)  
  
 class Meta:  
 app\_label = "gis"  
 db\_table = "CS\_SRS"  
 managed = False  
  
 @property  
 def wkt(self):  
 return self.wktext

## File: venv/lib/python3.12/site-packages/django/contrib/gis/db/backends/postgis/models.py

"""  
 The GeometryColumns and SpatialRefSys models for the PostGIS backend.  
"""  
from django.contrib.gis.db.backends.base.models import SpatialRefSysMixin  
from django.db import models  
  
  
class PostGISGeometryColumns(models.Model):  
 """  
 The 'geometry\_columns' view from PostGIS. See the PostGIS  
 documentation at Ch. 4.3.2.  
 """  
  
 f\_table\_catalog = models.CharField(max\_length=256)  
 f\_table\_schema = models.CharField(max\_length=256)  
 f\_table\_name = models.CharField(max\_length=256)  
 f\_geometry\_column = models.CharField(max\_length=256)  
 coord\_dimension = models.IntegerField()  
 srid = models.IntegerField(primary\_key=True)  
 type = models.CharField(max\_length=30)  
  
 class Meta:  
 app\_label = "gis"  
 db\_table = "geometry\_columns"  
 managed = False  
  
 def \_\_str\_\_(self):  
 return "%s.%s - %dD %s field (SRID: %d)" % (  
 self.f\_table\_name,  
 self.f\_geometry\_column,  
 self.coord\_dimension,  
 self.type,  
 self.srid,  
 )  
  
 @classmethod  
 def table\_name\_col(cls):  
 """  
 Return the name of the metadata column used to store the feature table  
 name.  
 """  
 return "f\_table\_name"  
  
 @classmethod  
 def geom\_col\_name(cls):  
 """  
 Return the name of the metadata column used to store the feature  
 geometry column.  
 """  
 return "f\_geometry\_column"  
  
  
class PostGISSpatialRefSys(models.Model, SpatialRefSysMixin):  
 """  
 The 'spatial\_ref\_sys' table from PostGIS. See the PostGIS  
 documentation at Ch. 4.2.1.  
 """  
  
 srid = models.IntegerField(primary\_key=True)  
 auth\_name = models.CharField(max\_length=256)  
 auth\_srid = models.IntegerField()  
 srtext = models.CharField(max\_length=2048)  
 proj4text = models.CharField(max\_length=2048)  
  
 class Meta:  
 app\_label = "gis"  
 db\_table = "spatial\_ref\_sys"  
 managed = False  
  
 @property  
 def wkt(self):  
 return self.srtext

## File: venv/lib/python3.12/site-packages/django/contrib/gis/db/backends/spatialite/models.py

"""  
 The GeometryColumns and SpatialRefSys models for the SpatiaLite backend.  
"""  
from django.contrib.gis.db.backends.base.models import SpatialRefSysMixin  
from django.db import models  
  
  
class SpatialiteGeometryColumns(models.Model):  
 """  
 The 'geometry\_columns' table from SpatiaLite.  
 """  
  
 f\_table\_name = models.CharField(max\_length=256)  
 f\_geometry\_column = models.CharField(max\_length=256)  
 coord\_dimension = models.IntegerField()  
 srid = models.IntegerField(primary\_key=True)  
 spatial\_index\_enabled = models.IntegerField()  
 type = models.IntegerField(db\_column="geometry\_type")  
  
 class Meta:  
 app\_label = "gis"  
 db\_table = "geometry\_columns"  
 managed = False  
  
 def \_\_str\_\_(self):  
 return "%s.%s - %dD %s field (SRID: %d)" % (  
 self.f\_table\_name,  
 self.f\_geometry\_column,  
 self.coord\_dimension,  
 self.type,  
 self.srid,  
 )  
  
 @classmethod  
 def table\_name\_col(cls):  
 """  
 Return the name of the metadata column used to store the feature table  
 name.  
 """  
 return "f\_table\_name"  
  
 @classmethod  
 def geom\_col\_name(cls):  
 """  
 Return the name of the metadata column used to store the feature  
 geometry column.  
 """  
 return "f\_geometry\_column"  
  
  
class SpatialiteSpatialRefSys(models.Model, SpatialRefSysMixin):  
 """  
 The 'spatial\_ref\_sys' table from SpatiaLite.  
 """  
  
 srid = models.IntegerField(primary\_key=True)  
 auth\_name = models.CharField(max\_length=256)  
 auth\_srid = models.IntegerField()  
 ref\_sys\_name = models.CharField(max\_length=256)  
 proj4text = models.CharField(max\_length=2048)  
 srtext = models.CharField(max\_length=2048)  
  
 class Meta:  
 app\_label = "gis"  
 db\_table = "spatial\_ref\_sys"  
 managed = False  
  
 @property  
 def wkt(self):  
 return self.srtext

## File: venv/lib/python3.12/site-packages/django/contrib/redirects/models.py

from django.contrib.sites.models import Site  
from django.db import models  
from django.utils.translation import gettext\_lazy as \_  
  
  
class Redirect(models.Model):  
 site = models.ForeignKey(Site, models.CASCADE, verbose\_name=\_("site"))  
 old\_path = models.CharField(  
 \_("redirect from"),  
 max\_length=200,  
 db\_index=True,  
 help\_text=\_(  
 "This should be an absolute path, excluding the domain name. Example: "  
 "“/events/search/”."  
 ),  
 )  
 new\_path = models.CharField(  
 \_("redirect to"),  
 max\_length=200,  
 blank=True,  
 help\_text=\_(  
 "This can be either an absolute path (as above) or a full URL "  
 "starting with a scheme such as “https://”."  
 ),  
 )  
  
 class Meta:  
 verbose\_name = \_("redirect")  
 verbose\_name\_plural = \_("redirects")  
 db\_table = "django\_redirect"  
 unique\_together = [["site", "old\_path"]]  
 ordering = ["old\_path"]  
  
 def \_\_str\_\_(self):  
 return "%s ---> %s" % (self.old\_path, self.new\_path)

## File: venv/lib/python3.12/site-packages/django/contrib/sessions/models.py

from django.contrib.sessions.base\_session import AbstractBaseSession, BaseSessionManager  
  
  
class SessionManager(BaseSessionManager):  
 use\_in\_migrations = True  
  
  
class Session(AbstractBaseSession):  
 """  
 Django provides full support for anonymous sessions. The session  
 framework lets you store and retrieve arbitrary data on a  
 per-site-visitor basis. It stores data on the server side and  
 abstracts the sending and receiving of cookies. Cookies contain a  
 session ID -- not the data itself.  
  
 The Django sessions framework is entirely cookie-based. It does  
 not fall back to putting session IDs in URLs. This is an intentional  
 design decision. Not only does that behavior make URLs ugly, it makes  
 your site vulnerable to session-ID theft via the "Referer" header.  
  
 For complete documentation on using Sessions in your code, consult  
 the sessions documentation that is shipped with Django (also available  
 on the Django web site).  
 """  
  
 objects = SessionManager()  
  
 @classmethod  
 def get\_session\_store\_class(cls):  
 from django.contrib.sessions.backends.db import SessionStore  
  
 return SessionStore  
  
 class Meta(AbstractBaseSession.Meta):  
 db\_table = "django\_session"

## File: venv/lib/python3.12/site-packages/django/contrib/sites/models.py

import string  
  
from django.core.exceptions import ImproperlyConfigured, ValidationError  
from django.db import models  
from django.db.models.signals import pre\_delete, pre\_save  
from django.http.request import split\_domain\_port  
from django.utils.translation import gettext\_lazy as \_  
  
SITE\_CACHE = {}  
  
  
def \_simple\_domain\_name\_validator(value):  
 """  
 Validate that the given value contains no whitespaces to prevent common  
 typos.  
 """  
 checks = ((s in value) for s in string.whitespace)  
 if any(checks):  
 raise ValidationError(  
 \_("The domain name cannot contain any spaces or tabs."),  
 code="invalid",  
 )  
  
  
class SiteManager(models.Manager):  
 use\_in\_migrations = True  
  
 def \_get\_site\_by\_id(self, site\_id):  
 if site\_id not in SITE\_CACHE:  
 site = self.get(pk=site\_id)  
 SITE\_CACHE[site\_id] = site  
 return SITE\_CACHE[site\_id]  
  
 def \_get\_site\_by\_request(self, request):  
 host = request.get\_host()  
 try:  
 # First attempt to look up the site by host with or without port.  
 if host not in SITE\_CACHE:  
 SITE\_CACHE[host] = self.get(domain\_\_iexact=host)  
 return SITE\_CACHE[host]  
 except Site.DoesNotExist:  
 # Fallback to looking up site after stripping port from the host.  
 domain, port = split\_domain\_port(host)  
 if domain not in SITE\_CACHE:  
 SITE\_CACHE[domain] = self.get(domain\_\_iexact=domain)  
 return SITE\_CACHE[domain]  
  
 def get\_current(self, request=None):  
 """  
 Return the current Site based on the SITE\_ID in the project's settings.  
 If SITE\_ID isn't defined, return the site with domain matching  
 request.get\_host(). The ``Site`` object is cached the first time it's  
 retrieved from the database.  
 """  
 from django.conf import settings  
  
 if getattr(settings, "SITE\_ID", ""):  
 site\_id = settings.SITE\_ID  
 return self.\_get\_site\_by\_id(site\_id)  
 elif request:  
 return self.\_get\_site\_by\_request(request)  
  
 raise ImproperlyConfigured(  
 'You\'re using the Django "sites framework" without having '  
 "set the SITE\_ID setting. Create a site in your database and "  
 "set the SITE\_ID setting or pass a request to "  
 "Site.objects.get\_current() to fix this error."  
 )  
  
 def clear\_cache(self):  
 """Clear the ``Site`` object cache."""  
 global SITE\_CACHE  
 SITE\_CACHE = {}  
  
 def get\_by\_natural\_key(self, domain):  
 return self.get(domain=domain)  
  
  
class Site(models.Model):  
 domain = models.CharField(  
 \_("domain name"),  
 max\_length=100,  
 validators=[\_simple\_domain\_name\_validator],  
 unique=True,  
 )  
 name = models.CharField(\_("display name"), max\_length=50)  
  
 objects = SiteManager()  
  
 class Meta:  
 db\_table = "django\_site"  
 verbose\_name = \_("site")  
 verbose\_name\_plural = \_("sites")  
 ordering = ["domain"]  
  
 def \_\_str\_\_(self):  
 return self.domain  
  
 def natural\_key(self):  
 return (self.domain,)  
  
  
def clear\_site\_cache(sender, \*\*kwargs):  
 """  
 Clear the cache (if primed) each time a site is saved or deleted.  
 """  
 instance = kwargs["instance"]  
 using = kwargs["using"]  
 try:  
 del SITE\_CACHE[instance.pk]  
 except KeyError:  
 pass  
 try:  
 del SITE\_CACHE[Site.objects.using(using).get(pk=instance.pk).domain]  
 except (KeyError, Site.DoesNotExist):  
 pass  
  
  
pre\_save.connect(clear\_site\_cache, sender=Site)  
pre\_delete.connect(clear\_site\_cache, sender=Site)

## File: venv/lib/python3.12/site-packages/django/db/migrations/operations/models.py

from django.db import models  
from django.db.migrations.operations.base import Operation  
from django.db.migrations.state import ModelState  
from django.db.migrations.utils import field\_references, resolve\_relation  
from django.db.models.options import normalize\_together  
from django.utils.functional import cached\_property  
  
from .fields import AddField, AlterField, FieldOperation, RemoveField, RenameField  
  
  
def \_check\_for\_duplicates(arg\_name, objs):  
 used\_vals = set()  
 for val in objs:  
 if val in used\_vals:  
 raise ValueError(  
 "Found duplicate value %s in CreateModel %s argument." % (val, arg\_name)  
 )  
 used\_vals.add(val)  
  
  
class ModelOperation(Operation):  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 @cached\_property  
 def name\_lower(self):  
 return self.name.lower()  
  
 def references\_model(self, name, app\_label):  
 return name.lower() == self.name\_lower  
  
 def reduce(self, operation, app\_label):  
 return super().reduce(operation, app\_label) or self.can\_reduce\_through(  
 operation, app\_label  
 )  
  
 def can\_reduce\_through(self, operation, app\_label):  
 return not operation.references\_model(self.name, app\_label)  
  
  
class CreateModel(ModelOperation):  
 """Create a model's table."""  
  
 serialization\_expand\_args = ["fields", "options", "managers"]  
  
 def \_\_init\_\_(self, name, fields, options=None, bases=None, managers=None):  
 self.fields = fields  
 self.options = options or {}  
 self.bases = bases or (models.Model,)  
 self.managers = managers or []  
 super().\_\_init\_\_(name)  
 # Sanity-check that there are no duplicated field names, bases, or  
 # manager names  
 \_check\_for\_duplicates("fields", (name for name, \_ in self.fields))  
 \_check\_for\_duplicates(  
 "bases",  
 (  
 base.\_meta.label\_lower  
 if hasattr(base, "\_meta")  
 else base.lower()  
 if isinstance(base, str)  
 else base  
 for base in self.bases  
 ),  
 )  
 \_check\_for\_duplicates("managers", (name for name, \_ in self.managers))  
  
 def deconstruct(self):  
 kwargs = {  
 "name": self.name,  
 "fields": self.fields,  
 }  
 if self.options:  
 kwargs["options"] = self.options  
 if self.bases and self.bases != (models.Model,):  
 kwargs["bases"] = self.bases  
 if self.managers and self.managers != [("objects", models.Manager())]:  
 kwargs["managers"] = self.managers  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.add\_model(  
 ModelState(  
 app\_label,  
 self.name,  
 list(self.fields),  
 dict(self.options),  
 tuple(self.bases),  
 list(self.managers),  
 )  
 )  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.create\_model(model)  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = from\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.delete\_model(model)  
  
 def describe(self):  
 return "Create %smodel %s" % (  
 "proxy " if self.options.get("proxy", False) else "",  
 self.name,  
 )  
  
 @property  
 def migration\_name\_fragment(self):  
 return self.name\_lower  
  
 def references\_model(self, name, app\_label):  
 name\_lower = name.lower()  
 if name\_lower == self.name\_lower:  
 return True  
  
 # Check we didn't inherit from the model  
 reference\_model\_tuple = (app\_label, name\_lower)  
 for base in self.bases:  
 if (  
 base is not models.Model  
 and isinstance(base, (models.base.ModelBase, str))  
 and resolve\_relation(base, app\_label) == reference\_model\_tuple  
 ):  
 return True  
  
 # Check we have no FKs/M2Ms with it  
 for \_name, field in self.fields:  
 if field\_references(  
 (app\_label, self.name\_lower), field, reference\_model\_tuple  
 ):  
 return True  
 return False  
  
 def reduce(self, operation, app\_label):  
 if (  
 isinstance(operation, DeleteModel)  
 and self.name\_lower == operation.name\_lower  
 and not self.options.get("proxy", False)  
 ):  
 return []  
 elif (  
 isinstance(operation, RenameModel)  
 and self.name\_lower == operation.old\_name\_lower  
 ):  
 return [  
 CreateModel(  
 operation.new\_name,  
 fields=self.fields,  
 options=self.options,  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif (  
 isinstance(operation, AlterModelOptions)  
 and self.name\_lower == operation.name\_lower  
 ):  
 options = {\*\*self.options, \*\*operation.options}  
 for key in operation.ALTER\_OPTION\_KEYS:  
 if key not in operation.options:  
 options.pop(key, None)  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields,  
 options=options,  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif (  
 isinstance(operation, AlterModelManagers)  
 and self.name\_lower == operation.name\_lower  
 ):  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields,  
 options=self.options,  
 bases=self.bases,  
 managers=operation.managers,  
 ),  
 ]  
 elif (  
 isinstance(operation, AlterTogetherOptionOperation)  
 and self.name\_lower == operation.name\_lower  
 ):  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields,  
 options={  
 \*\*self.options,  
 \*\*{operation.option\_name: operation.option\_value},  
 },  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif (  
 isinstance(operation, AlterOrderWithRespectTo)  
 and self.name\_lower == operation.name\_lower  
 ):  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields,  
 options={  
 \*\*self.options,  
 "order\_with\_respect\_to": operation.order\_with\_respect\_to,  
 },  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif (  
 isinstance(operation, FieldOperation)  
 and self.name\_lower == operation.model\_name\_lower  
 ):  
 if isinstance(operation, AddField):  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields + [(operation.name, operation.field)],  
 options=self.options,  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif isinstance(operation, AlterField):  
 return [  
 CreateModel(  
 self.name,  
 fields=[  
 (n, operation.field if n == operation.name else v)  
 for n, v in self.fields  
 ],  
 options=self.options,  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif isinstance(operation, RemoveField):  
 options = self.options.copy()  
 for option\_name in ("unique\_together", "index\_together"):  
 option = options.pop(option\_name, None)  
 if option:  
 option = set(  
 filter(  
 bool,  
 (  
 tuple(  
 f for f in fields if f != operation.name\_lower  
 )  
 for fields in option  
 ),  
 )  
 )  
 if option:  
 options[option\_name] = option  
 order\_with\_respect\_to = options.get("order\_with\_respect\_to")  
 if order\_with\_respect\_to == operation.name\_lower:  
 del options["order\_with\_respect\_to"]  
 return [  
 CreateModel(  
 self.name,  
 fields=[  
 (n, v)  
 for n, v in self.fields  
 if n.lower() != operation.name\_lower  
 ],  
 options=options,  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif isinstance(operation, RenameField):  
 options = self.options.copy()  
 for option\_name in ("unique\_together", "index\_together"):  
 option = options.get(option\_name)  
 if option:  
 options[option\_name] = {  
 tuple(  
 operation.new\_name if f == operation.old\_name else f  
 for f in fields  
 )  
 for fields in option  
 }  
 order\_with\_respect\_to = options.get("order\_with\_respect\_to")  
 if order\_with\_respect\_to == operation.old\_name:  
 options["order\_with\_respect\_to"] = operation.new\_name  
 return [  
 CreateModel(  
 self.name,  
 fields=[  
 (operation.new\_name if n == operation.old\_name else n, v)  
 for n, v in self.fields  
 ],  
 options=options,  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif (  
 isinstance(operation, IndexOperation)  
 and self.name\_lower == operation.model\_name\_lower  
 ):  
 if isinstance(operation, AddIndex):  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields,  
 options={  
 \*\*self.options,  
 "indexes": [  
 \*self.options.get("indexes", []),  
 operation.index,  
 ],  
 },  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif isinstance(operation, RemoveIndex):  
 options\_indexes = [  
 index  
 for index in self.options.get("indexes", [])  
 if index.name != operation.name  
 ]  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields,  
 options={  
 \*\*self.options,  
 "indexes": options\_indexes,  
 },  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 elif isinstance(operation, RenameIndex) and operation.old\_fields:  
 options\_index\_together = {  
 fields  
 for fields in self.options.get("index\_together", [])  
 if fields != operation.old\_fields  
 }  
 if options\_index\_together:  
 self.options["index\_together"] = options\_index\_together  
 else:  
 self.options.pop("index\_together", None)  
 return [  
 CreateModel(  
 self.name,  
 fields=self.fields,  
 options={  
 \*\*self.options,  
 "indexes": [  
 \*self.options.get("indexes", []),  
 models.Index(  
 fields=operation.old\_fields, name=operation.new\_name  
 ),  
 ],  
 },  
 bases=self.bases,  
 managers=self.managers,  
 ),  
 ]  
 return super().reduce(operation, app\_label)  
  
  
class DeleteModel(ModelOperation):  
 """Drop a model's table."""  
  
 def deconstruct(self):  
 kwargs = {  
 "name": self.name,  
 }  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.remove\_model(app\_label, self.name\_lower)  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = from\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.delete\_model(model)  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.create\_model(model)  
  
 def references\_model(self, name, app\_label):  
 # The deleted model could be referencing the specified model through  
 # related fields.  
 return True  
  
 def describe(self):  
 return "Delete model %s" % self.name  
  
 @property  
 def migration\_name\_fragment(self):  
 return "delete\_%s" % self.name\_lower  
  
  
class RenameModel(ModelOperation):  
 """Rename a model."""  
  
 def \_\_init\_\_(self, old\_name, new\_name):  
 self.old\_name = old\_name  
 self.new\_name = new\_name  
 super().\_\_init\_\_(old\_name)  
  
 @cached\_property  
 def old\_name\_lower(self):  
 return self.old\_name.lower()  
  
 @cached\_property  
 def new\_name\_lower(self):  
 return self.new\_name.lower()  
  
 def deconstruct(self):  
 kwargs = {  
 "old\_name": self.old\_name,  
 "new\_name": self.new\_name,  
 }  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.rename\_model(app\_label, self.old\_name, self.new\_name)  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 new\_model = to\_state.apps.get\_model(app\_label, self.new\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, new\_model):  
 old\_model = from\_state.apps.get\_model(app\_label, self.old\_name)  
 # Move the main table  
 schema\_editor.alter\_db\_table(  
 new\_model,  
 old\_model.\_meta.db\_table,  
 new\_model.\_meta.db\_table,  
 )  
 # Alter the fields pointing to us  
 for related\_object in old\_model.\_meta.related\_objects:  
 if related\_object.related\_model == old\_model:  
 model = new\_model  
 related\_key = (app\_label, self.new\_name\_lower)  
 else:  
 model = related\_object.related\_model  
 related\_key = (  
 related\_object.related\_model.\_meta.app\_label,  
 related\_object.related\_model.\_meta.model\_name,  
 )  
 to\_field = to\_state.apps.get\_model(\*related\_key).\_meta.get\_field(  
 related\_object.field.name  
 )  
 schema\_editor.alter\_field(  
 model,  
 related\_object.field,  
 to\_field,  
 )  
 # Rename M2M fields whose name is based on this model's name.  
 fields = zip(  
 old\_model.\_meta.local\_many\_to\_many, new\_model.\_meta.local\_many\_to\_many  
 )  
 for old\_field, new\_field in fields:  
 # Skip self-referential fields as these are renamed above.  
 if (  
 new\_field.model == new\_field.related\_model  
 or not new\_field.remote\_field.through.\_meta.auto\_created  
 ):  
 continue  
 # Rename columns and the M2M table.  
 schema\_editor.\_alter\_many\_to\_many(  
 new\_model,  
 old\_field,  
 new\_field,  
 strict=False,  
 )  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 self.new\_name\_lower, self.old\_name\_lower = (  
 self.old\_name\_lower,  
 self.new\_name\_lower,  
 )  
 self.new\_name, self.old\_name = self.old\_name, self.new\_name  
  
 self.database\_forwards(app\_label, schema\_editor, from\_state, to\_state)  
  
 self.new\_name\_lower, self.old\_name\_lower = (  
 self.old\_name\_lower,  
 self.new\_name\_lower,  
 )  
 self.new\_name, self.old\_name = self.old\_name, self.new\_name  
  
 def references\_model(self, name, app\_label):  
 return (  
 name.lower() == self.old\_name\_lower or name.lower() == self.new\_name\_lower  
 )  
  
 def describe(self):  
 return "Rename model %s to %s" % (self.old\_name, self.new\_name)  
  
 @property  
 def migration\_name\_fragment(self):  
 return "rename\_%s\_%s" % (self.old\_name\_lower, self.new\_name\_lower)  
  
 def reduce(self, operation, app\_label):  
 if (  
 isinstance(operation, RenameModel)  
 and self.new\_name\_lower == operation.old\_name\_lower  
 ):  
 return [  
 RenameModel(  
 self.old\_name,  
 operation.new\_name,  
 ),  
 ]  
 # Skip `ModelOperation.reduce` as we want to run `references\_model`  
 # against self.new\_name.  
 return super(ModelOperation, self).reduce(  
 operation, app\_label  
 ) or not operation.references\_model(self.new\_name, app\_label)  
  
  
class ModelOptionOperation(ModelOperation):  
 def reduce(self, operation, app\_label):  
 if (  
 isinstance(operation, (self.\_\_class\_\_, DeleteModel))  
 and self.name\_lower == operation.name\_lower  
 ):  
 return [operation]  
 return super().reduce(operation, app\_label)  
  
  
class AlterModelTable(ModelOptionOperation):  
 """Rename a model's table."""  
  
 def \_\_init\_\_(self, name, table):  
 self.table = table  
 super().\_\_init\_\_(name)  
  
 def deconstruct(self):  
 kwargs = {  
 "name": self.name,  
 "table": self.table,  
 }  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.alter\_model\_options(app\_label, self.name\_lower, {"db\_table": self.table})  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 new\_model = to\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, new\_model):  
 old\_model = from\_state.apps.get\_model(app\_label, self.name)  
 schema\_editor.alter\_db\_table(  
 new\_model,  
 old\_model.\_meta.db\_table,  
 new\_model.\_meta.db\_table,  
 )  
 # Rename M2M fields whose name is based on this model's db\_table  
 for old\_field, new\_field in zip(  
 old\_model.\_meta.local\_many\_to\_many, new\_model.\_meta.local\_many\_to\_many  
 ):  
 if new\_field.remote\_field.through.\_meta.auto\_created:  
 schema\_editor.alter\_db\_table(  
 new\_field.remote\_field.through,  
 old\_field.remote\_field.through.\_meta.db\_table,  
 new\_field.remote\_field.through.\_meta.db\_table,  
 )  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 return self.database\_forwards(app\_label, schema\_editor, from\_state, to\_state)  
  
 def describe(self):  
 return "Rename table for %s to %s" % (  
 self.name,  
 self.table if self.table is not None else "(default)",  
 )  
  
 @property  
 def migration\_name\_fragment(self):  
 return "alter\_%s\_table" % self.name\_lower  
  
  
class AlterModelTableComment(ModelOptionOperation):  
 def \_\_init\_\_(self, name, table\_comment):  
 self.table\_comment = table\_comment  
 super().\_\_init\_\_(name)  
  
 def deconstruct(self):  
 kwargs = {  
 "name": self.name,  
 "table\_comment": self.table\_comment,  
 }  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.alter\_model\_options(  
 app\_label, self.name\_lower, {"db\_table\_comment": self.table\_comment}  
 )  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 new\_model = to\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, new\_model):  
 old\_model = from\_state.apps.get\_model(app\_label, self.name)  
 schema\_editor.alter\_db\_table\_comment(  
 new\_model,  
 old\_model.\_meta.db\_table\_comment,  
 new\_model.\_meta.db\_table\_comment,  
 )  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 return self.database\_forwards(app\_label, schema\_editor, from\_state, to\_state)  
  
 def describe(self):  
 return f"Alter {self.name} table comment"  
  
 @property  
 def migration\_name\_fragment(self):  
 return f"alter\_{self.name\_lower}\_table\_comment"  
  
  
class AlterTogetherOptionOperation(ModelOptionOperation):  
 option\_name = None  
  
 def \_\_init\_\_(self, name, option\_value):  
 if option\_value:  
 option\_value = set(normalize\_together(option\_value))  
 setattr(self, self.option\_name, option\_value)  
 super().\_\_init\_\_(name)  
  
 @cached\_property  
 def option\_value(self):  
 return getattr(self, self.option\_name)  
  
 def deconstruct(self):  
 kwargs = {  
 "name": self.name,  
 self.option\_name: self.option\_value,  
 }  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.alter\_model\_options(  
 app\_label,  
 self.name\_lower,  
 {self.option\_name: self.option\_value},  
 )  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 new\_model = to\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, new\_model):  
 old\_model = from\_state.apps.get\_model(app\_label, self.name)  
 alter\_together = getattr(schema\_editor, "alter\_%s" % self.option\_name)  
 alter\_together(  
 new\_model,  
 getattr(old\_model.\_meta, self.option\_name, set()),  
 getattr(new\_model.\_meta, self.option\_name, set()),  
 )  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 return self.database\_forwards(app\_label, schema\_editor, from\_state, to\_state)  
  
 def references\_field(self, model\_name, name, app\_label):  
 return self.references\_model(model\_name, app\_label) and (  
 not self.option\_value  
 or any((name in fields) for fields in self.option\_value)  
 )  
  
 def describe(self):  
 return "Alter %s for %s (%s constraint(s))" % (  
 self.option\_name,  
 self.name,  
 len(self.option\_value or ""),  
 )  
  
 @property  
 def migration\_name\_fragment(self):  
 return "alter\_%s\_%s" % (self.name\_lower, self.option\_name)  
  
 def can\_reduce\_through(self, operation, app\_label):  
 return super().can\_reduce\_through(operation, app\_label) or (  
 isinstance(operation, AlterTogetherOptionOperation)  
 and type(operation) is not type(self)  
 )  
  
  
class AlterUniqueTogether(AlterTogetherOptionOperation):  
 """  
 Change the value of unique\_together to the target one.  
 Input value of unique\_together must be a set of tuples.  
 """  
  
 option\_name = "unique\_together"  
  
 def \_\_init\_\_(self, name, unique\_together):  
 super().\_\_init\_\_(name, unique\_together)  
  
  
class AlterIndexTogether(AlterTogetherOptionOperation):  
 """  
 Change the value of index\_together to the target one.  
 Input value of index\_together must be a set of tuples.  
 """  
  
 option\_name = "index\_together"  
  
 def \_\_init\_\_(self, name, index\_together):  
 super().\_\_init\_\_(name, index\_together)  
  
  
class AlterOrderWithRespectTo(ModelOptionOperation):  
 """Represent a change with the order\_with\_respect\_to option."""  
  
 option\_name = "order\_with\_respect\_to"  
  
 def \_\_init\_\_(self, name, order\_with\_respect\_to):  
 self.order\_with\_respect\_to = order\_with\_respect\_to  
 super().\_\_init\_\_(name)  
  
 def deconstruct(self):  
 kwargs = {  
 "name": self.name,  
 "order\_with\_respect\_to": self.order\_with\_respect\_to,  
 }  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.alter\_model\_options(  
 app\_label,  
 self.name\_lower,  
 {self.option\_name: self.order\_with\_respect\_to},  
 )  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 to\_model = to\_state.apps.get\_model(app\_label, self.name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, to\_model):  
 from\_model = from\_state.apps.get\_model(app\_label, self.name)  
 # Remove a field if we need to  
 if (  
 from\_model.\_meta.order\_with\_respect\_to  
 and not to\_model.\_meta.order\_with\_respect\_to  
 ):  
 schema\_editor.remove\_field(  
 from\_model, from\_model.\_meta.get\_field("\_order")  
 )  
 # Add a field if we need to (altering the column is untouched as  
 # it's likely a rename)  
 elif (  
 to\_model.\_meta.order\_with\_respect\_to  
 and not from\_model.\_meta.order\_with\_respect\_to  
 ):  
 field = to\_model.\_meta.get\_field("\_order")  
 if not field.has\_default():  
 field.default = 0  
 schema\_editor.add\_field(  
 from\_model,  
 field,  
 )  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 self.database\_forwards(app\_label, schema\_editor, from\_state, to\_state)  
  
 def references\_field(self, model\_name, name, app\_label):  
 return self.references\_model(model\_name, app\_label) and (  
 self.order\_with\_respect\_to is None or name == self.order\_with\_respect\_to  
 )  
  
 def describe(self):  
 return "Set order\_with\_respect\_to on %s to %s" % (  
 self.name,  
 self.order\_with\_respect\_to,  
 )  
  
 @property  
 def migration\_name\_fragment(self):  
 return "alter\_%s\_order\_with\_respect\_to" % self.name\_lower  
  
  
class AlterModelOptions(ModelOptionOperation):  
 """  
 Set new model options that don't directly affect the database schema  
 (like verbose\_name, permissions, ordering). Python code in migrations  
 may still need them.  
 """  
  
 # Model options we want to compare and preserve in an AlterModelOptions op  
 ALTER\_OPTION\_KEYS = [  
 "base\_manager\_name",  
 "default\_manager\_name",  
 "default\_related\_name",  
 "get\_latest\_by",  
 "managed",  
 "ordering",  
 "permissions",  
 "default\_permissions",  
 "select\_on\_save",  
 "verbose\_name",  
 "verbose\_name\_plural",  
 ]  
  
 def \_\_init\_\_(self, name, options):  
 self.options = options  
 super().\_\_init\_\_(name)  
  
 def deconstruct(self):  
 kwargs = {  
 "name": self.name,  
 "options": self.options,  
 }  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 state.alter\_model\_options(  
 app\_label,  
 self.name\_lower,  
 self.options,  
 self.ALTER\_OPTION\_KEYS,  
 )  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 pass  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 pass  
  
 def describe(self):  
 return "Change Meta options on %s" % self.name  
  
 @property  
 def migration\_name\_fragment(self):  
 return "alter\_%s\_options" % self.name\_lower  
  
  
class AlterModelManagers(ModelOptionOperation):  
 """Alter the model's managers."""  
  
 serialization\_expand\_args = ["managers"]  
  
 def \_\_init\_\_(self, name, managers):  
 self.managers = managers  
 super().\_\_init\_\_(name)  
  
 def deconstruct(self):  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [self.name, self.managers], {})  
  
 def state\_forwards(self, app\_label, state):  
 state.alter\_model\_managers(app\_label, self.name\_lower, self.managers)  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 pass  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 pass  
  
 def describe(self):  
 return "Change managers on %s" % self.name  
  
 @property  
 def migration\_name\_fragment(self):  
 return "alter\_%s\_managers" % self.name\_lower  
  
  
class IndexOperation(Operation):  
 option\_name = "indexes"  
  
 @cached\_property  
 def model\_name\_lower(self):  
 return self.model\_name.lower()  
  
  
class AddIndex(IndexOperation):  
 """Add an index on a model."""  
  
 def \_\_init\_\_(self, model\_name, index):  
 self.model\_name = model\_name  
 if not index.name:  
 raise ValueError(  
 "Indexes passed to AddIndex operations require a name "  
 "argument. %r doesn't have one." % index  
 )  
 self.index = index  
  
 def state\_forwards(self, app\_label, state):  
 state.add\_index(app\_label, self.model\_name\_lower, self.index)  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.add\_index(model, self.index)  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = from\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.remove\_index(model, self.index)  
  
 def deconstruct(self):  
 kwargs = {  
 "model\_name": self.model\_name,  
 "index": self.index,  
 }  
 return (  
 self.\_\_class\_\_.\_\_qualname\_\_,  
 [],  
 kwargs,  
 )  
  
 def describe(self):  
 if self.index.expressions:  
 return "Create index %s on %s on model %s" % (  
 self.index.name,  
 ", ".join([str(expression) for expression in self.index.expressions]),  
 self.model\_name,  
 )  
 return "Create index %s on field(s) %s of model %s" % (  
 self.index.name,  
 ", ".join(self.index.fields),  
 self.model\_name,  
 )  
  
 @property  
 def migration\_name\_fragment(self):  
 return "%s\_%s" % (self.model\_name\_lower, self.index.name.lower())  
  
  
class RemoveIndex(IndexOperation):  
 """Remove an index from a model."""  
  
 def \_\_init\_\_(self, model\_name, name):  
 self.model\_name = model\_name  
 self.name = name  
  
 def state\_forwards(self, app\_label, state):  
 state.remove\_index(app\_label, self.model\_name\_lower, self.name)  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = from\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 from\_model\_state = from\_state.models[app\_label, self.model\_name\_lower]  
 index = from\_model\_state.get\_index\_by\_name(self.name)  
 schema\_editor.remove\_index(model, index)  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 to\_model\_state = to\_state.models[app\_label, self.model\_name\_lower]  
 index = to\_model\_state.get\_index\_by\_name(self.name)  
 schema\_editor.add\_index(model, index)  
  
 def deconstruct(self):  
 kwargs = {  
 "model\_name": self.model\_name,  
 "name": self.name,  
 }  
 return (  
 self.\_\_class\_\_.\_\_qualname\_\_,  
 [],  
 kwargs,  
 )  
  
 def describe(self):  
 return "Remove index %s from %s" % (self.name, self.model\_name)  
  
 @property  
 def migration\_name\_fragment(self):  
 return "remove\_%s\_%s" % (self.model\_name\_lower, self.name.lower())  
  
  
class RenameIndex(IndexOperation):  
 """Rename an index."""  
  
 def \_\_init\_\_(self, model\_name, new\_name, old\_name=None, old\_fields=None):  
 if not old\_name and not old\_fields:  
 raise ValueError(  
 "RenameIndex requires one of old\_name and old\_fields arguments to be "  
 "set."  
 )  
 if old\_name and old\_fields:  
 raise ValueError(  
 "RenameIndex.old\_name and old\_fields are mutually exclusive."  
 )  
 self.model\_name = model\_name  
 self.new\_name = new\_name  
 self.old\_name = old\_name  
 self.old\_fields = old\_fields  
  
 @cached\_property  
 def old\_name\_lower(self):  
 return self.old\_name.lower()  
  
 @cached\_property  
 def new\_name\_lower(self):  
 return self.new\_name.lower()  
  
 def deconstruct(self):  
 kwargs = {  
 "model\_name": self.model\_name,  
 "new\_name": self.new\_name,  
 }  
 if self.old\_name:  
 kwargs["old\_name"] = self.old\_name  
 if self.old\_fields:  
 kwargs["old\_fields"] = self.old\_fields  
 return (self.\_\_class\_\_.\_\_qualname\_\_, [], kwargs)  
  
 def state\_forwards(self, app\_label, state):  
 if self.old\_fields:  
 state.add\_index(  
 app\_label,  
 self.model\_name\_lower,  
 models.Index(fields=self.old\_fields, name=self.new\_name),  
 )  
 state.remove\_model\_options(  
 app\_label,  
 self.model\_name\_lower,  
 AlterIndexTogether.option\_name,  
 self.old\_fields,  
 )  
 else:  
 state.rename\_index(  
 app\_label, self.model\_name\_lower, self.old\_name, self.new\_name  
 )  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.model\_name)  
 if not self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 return  
  
 if self.old\_fields:  
 from\_model = from\_state.apps.get\_model(app\_label, self.model\_name)  
 columns = [  
 from\_model.\_meta.get\_field(field).column for field in self.old\_fields  
 ]  
 matching\_index\_name = schema\_editor.\_constraint\_names(  
 from\_model, column\_names=columns, index=True  
 )  
 if len(matching\_index\_name) != 1:  
 raise ValueError(  
 "Found wrong number (%s) of indexes for %s(%s)."  
 % (  
 len(matching\_index\_name),  
 from\_model.\_meta.db\_table,  
 ", ".join(columns),  
 )  
 )  
 old\_index = models.Index(  
 fields=self.old\_fields,  
 name=matching\_index\_name[0],  
 )  
 else:  
 from\_model\_state = from\_state.models[app\_label, self.model\_name\_lower]  
 old\_index = from\_model\_state.get\_index\_by\_name(self.old\_name)  
 # Don't alter when the index name is not changed.  
 if old\_index.name == self.new\_name:  
 return  
  
 to\_model\_state = to\_state.models[app\_label, self.model\_name\_lower]  
 new\_index = to\_model\_state.get\_index\_by\_name(self.new\_name)  
 schema\_editor.rename\_index(model, old\_index, new\_index)  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 if self.old\_fields:  
 # Backward operation with unnamed index is a no-op.  
 return  
  
 self.new\_name\_lower, self.old\_name\_lower = (  
 self.old\_name\_lower,  
 self.new\_name\_lower,  
 )  
 self.new\_name, self.old\_name = self.old\_name, self.new\_name  
  
 self.database\_forwards(app\_label, schema\_editor, from\_state, to\_state)  
  
 self.new\_name\_lower, self.old\_name\_lower = (  
 self.old\_name\_lower,  
 self.new\_name\_lower,  
 )  
 self.new\_name, self.old\_name = self.old\_name, self.new\_name  
  
 def describe(self):  
 if self.old\_name:  
 return (  
 f"Rename index {self.old\_name} on {self.model\_name} to {self.new\_name}"  
 )  
 return (  
 f"Rename unnamed index for {self.old\_fields} on {self.model\_name} to "  
 f"{self.new\_name}"  
 )  
  
 @property  
 def migration\_name\_fragment(self):  
 if self.old\_name:  
 return "rename\_%s\_%s" % (self.old\_name\_lower, self.new\_name\_lower)  
 return "rename\_%s\_%s\_%s" % (  
 self.model\_name\_lower,  
 "\_".join(self.old\_fields),  
 self.new\_name\_lower,  
 )  
  
 def reduce(self, operation, app\_label):  
 if (  
 isinstance(operation, RenameIndex)  
 and self.model\_name\_lower == operation.model\_name\_lower  
 and operation.old\_name  
 and self.new\_name\_lower == operation.old\_name\_lower  
 ):  
 return [  
 RenameIndex(  
 self.model\_name,  
 new\_name=operation.new\_name,  
 old\_name=self.old\_name,  
 old\_fields=self.old\_fields,  
 )  
 ]  
 return super().reduce(operation, app\_label)  
  
  
class AddConstraint(IndexOperation):  
 option\_name = "constraints"  
  
 def \_\_init\_\_(self, model\_name, constraint):  
 self.model\_name = model\_name  
 self.constraint = constraint  
  
 def state\_forwards(self, app\_label, state):  
 state.add\_constraint(app\_label, self.model\_name\_lower, self.constraint)  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.add\_constraint(model, self.constraint)  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 schema\_editor.remove\_constraint(model, self.constraint)  
  
 def deconstruct(self):  
 return (  
 self.\_\_class\_\_.\_\_name\_\_,  
 [],  
 {  
 "model\_name": self.model\_name,  
 "constraint": self.constraint,  
 },  
 )  
  
 def describe(self):  
 return "Create constraint %s on model %s" % (  
 self.constraint.name,  
 self.model\_name,  
 )  
  
 @property  
 def migration\_name\_fragment(self):  
 return "%s\_%s" % (self.model\_name\_lower, self.constraint.name.lower())  
  
  
class RemoveConstraint(IndexOperation):  
 option\_name = "constraints"  
  
 def \_\_init\_\_(self, model\_name, name):  
 self.model\_name = model\_name  
 self.name = name  
  
 def state\_forwards(self, app\_label, state):  
 state.remove\_constraint(app\_label, self.model\_name\_lower, self.name)  
  
 def database\_forwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 from\_model\_state = from\_state.models[app\_label, self.model\_name\_lower]  
 constraint = from\_model\_state.get\_constraint\_by\_name(self.name)  
 schema\_editor.remove\_constraint(model, constraint)  
  
 def database\_backwards(self, app\_label, schema\_editor, from\_state, to\_state):  
 model = to\_state.apps.get\_model(app\_label, self.model\_name)  
 if self.allow\_migrate\_model(schema\_editor.connection.alias, model):  
 to\_model\_state = to\_state.models[app\_label, self.model\_name\_lower]  
 constraint = to\_model\_state.get\_constraint\_by\_name(self.name)  
 schema\_editor.add\_constraint(model, constraint)  
  
 def deconstruct(self):  
 return (  
 self.\_\_class\_\_.\_\_name\_\_,  
 [],  
 {  
 "model\_name": self.model\_name,  
 "name": self.name,  
 },  
 )  
  
 def describe(self):  
 return "Remove constraint %s from model %s" % (self.name, self.model\_name)  
  
 @property  
 def migration\_name\_fragment(self):  
 return "remove\_%s\_%s" % (self.model\_name\_lower, self.name.lower())

## File: venv/lib/python3.12/site-packages/django/forms/models.py

"""  
Helper functions for creating Form classes from Django models  
and database field objects.  
"""  
from itertools import chain  
  
from django.core.exceptions import (  
 NON\_FIELD\_ERRORS,  
 FieldError,  
 ImproperlyConfigured,  
 ValidationError,  
)  
from django.db.models.utils import AltersData  
from django.forms.fields import ChoiceField, Field  
from django.forms.forms import BaseForm, DeclarativeFieldsMetaclass  
from django.forms.formsets import BaseFormSet, formset\_factory  
from django.forms.utils import ErrorList  
from django.forms.widgets import (  
 HiddenInput,  
 MultipleHiddenInput,  
 RadioSelect,  
 SelectMultiple,  
)  
from django.utils.text import capfirst, get\_text\_list  
from django.utils.translation import gettext  
from django.utils.translation import gettext\_lazy as \_  
  
\_\_all\_\_ = (  
 "ModelForm",  
 "BaseModelForm",  
 "model\_to\_dict",  
 "fields\_for\_model",  
 "ModelChoiceField",  
 "ModelMultipleChoiceField",  
 "ALL\_FIELDS",  
 "BaseModelFormSet",  
 "modelformset\_factory",  
 "BaseInlineFormSet",  
 "inlineformset\_factory",  
 "modelform\_factory",  
)  
  
ALL\_FIELDS = "\_\_all\_\_"  
  
  
def construct\_instance(form, instance, fields=None, exclude=None):  
 """  
 Construct and return a model instance from the bound ``form``'s  
 ``cleaned\_data``, but do not save the returned instance to the database.  
 """  
 from django.db import models  
  
 opts = instance.\_meta  
  
 cleaned\_data = form.cleaned\_data  
 file\_field\_list = []  
 for f in opts.fields:  
 if (  
 not f.editable  
 or isinstance(f, models.AutoField)  
 or f.name not in cleaned\_data  
 ):  
 continue  
 if fields is not None and f.name not in fields:  
 continue  
 if exclude and f.name in exclude:  
 continue  
 # Leave defaults for fields that aren't in POST data, except for  
 # checkbox inputs because they don't appear in POST data if not checked.  
 if (  
 f.has\_default()  
 and form[f.name].field.widget.value\_omitted\_from\_data(  
 form.data, form.files, form.add\_prefix(f.name)  
 )  
 and cleaned\_data.get(f.name) in form[f.name].field.empty\_values  
 ):  
 continue  
 # Defer saving file-type fields until after the other fields, so a  
 # callable upload\_to can use the values from other fields.  
 if isinstance(f, models.FileField):  
 file\_field\_list.append(f)  
 else:  
 f.save\_form\_data(instance, cleaned\_data[f.name])  
  
 for f in file\_field\_list:  
 f.save\_form\_data(instance, cleaned\_data[f.name])  
  
 return instance  
  
  
# ModelForms #################################################################  
  
  
def model\_to\_dict(instance, fields=None, exclude=None):  
 """  
 Return a dict containing the data in ``instance`` suitable for passing as  
 a Form's ``initial`` keyword argument.  
  
 ``fields`` is an optional list of field names. If provided, return only the  
 named.  
  
 ``exclude`` is an optional list of field names. If provided, exclude the  
 named from the returned dict, even if they are listed in the ``fields``  
 argument.  
 """  
 opts = instance.\_meta  
 data = {}  
 for f in chain(opts.concrete\_fields, opts.private\_fields, opts.many\_to\_many):  
 if not getattr(f, "editable", False):  
 continue  
 if fields is not None and f.name not in fields:  
 continue  
 if exclude and f.name in exclude:  
 continue  
 data[f.name] = f.value\_from\_object(instance)  
 return data  
  
  
def apply\_limit\_choices\_to\_to\_formfield(formfield):  
 """Apply limit\_choices\_to to the formfield's queryset if needed."""  
 from django.db.models import Exists, OuterRef, Q  
  
 if hasattr(formfield, "queryset") and hasattr(formfield, "get\_limit\_choices\_to"):  
 limit\_choices\_to = formfield.get\_limit\_choices\_to()  
 if limit\_choices\_to:  
 complex\_filter = limit\_choices\_to  
 if not isinstance(complex\_filter, Q):  
 complex\_filter = Q(\*\*limit\_choices\_to)  
 complex\_filter &= Q(pk=OuterRef("pk"))  
 # Use Exists() to avoid potential duplicates.  
 formfield.queryset = formfield.queryset.filter(  
 Exists(formfield.queryset.model.\_base\_manager.filter(complex\_filter)),  
 )  
  
  
def fields\_for\_model(  
 model,  
 fields=None,  
 exclude=None,  
 widgets=None,  
 formfield\_callback=None,  
 localized\_fields=None,  
 labels=None,  
 help\_texts=None,  
 error\_messages=None,  
 field\_classes=None,  
 \*,  
 apply\_limit\_choices\_to=True,  
):  
 """  
 Return a dictionary containing form fields for the given model.  
  
 ``fields`` is an optional list of field names. If provided, return only the  
 named fields.  
  
 ``exclude`` is an optional list of field names. If provided, exclude the  
 named fields from the returned fields, even if they are listed in the  
 ``fields`` argument.  
  
 ``widgets`` is a dictionary of model field names mapped to a widget.  
  
 ``formfield\_callback`` is a callable that takes a model field and returns  
 a form field.  
  
 ``localized\_fields`` is a list of names of fields which should be localized.  
  
 ``labels`` is a dictionary of model field names mapped to a label.  
  
 ``help\_texts`` is a dictionary of model field names mapped to a help text.  
  
 ``error\_messages`` is a dictionary of model field names mapped to a  
 dictionary of error messages.  
  
 ``field\_classes`` is a dictionary of model field names mapped to a form  
 field class.  
  
 ``apply\_limit\_choices\_to`` is a boolean indicating if limit\_choices\_to  
 should be applied to a field's queryset.  
 """  
 field\_dict = {}  
 ignored = []  
 opts = model.\_meta  
 # Avoid circular import  
 from django.db.models import Field as ModelField  
  
 sortable\_private\_fields = [  
 f for f in opts.private\_fields if isinstance(f, ModelField)  
 ]  
 for f in sorted(  
 chain(opts.concrete\_fields, sortable\_private\_fields, opts.many\_to\_many)  
 ):  
 if not getattr(f, "editable", False):  
 if (  
 fields is not None  
 and f.name in fields  
 and (exclude is None or f.name not in exclude)  
 ):  
 raise FieldError(  
 "'%s' cannot be specified for %s model form as it is a "  
 "non-editable field" % (f.name, model.\_\_name\_\_)  
 )  
 continue  
 if fields is not None and f.name not in fields:  
 continue  
 if exclude and f.name in exclude:  
 continue  
  
 kwargs = {}  
 if widgets and f.name in widgets:  
 kwargs["widget"] = widgets[f.name]  
 if localized\_fields == ALL\_FIELDS or (  
 localized\_fields and f.name in localized\_fields  
 ):  
 kwargs["localize"] = True  
 if labels and f.name in labels:  
 kwargs["label"] = labels[f.name]  
 if help\_texts and f.name in help\_texts:  
 kwargs["help\_text"] = help\_texts[f.name]  
 if error\_messages and f.name in error\_messages:  
 kwargs["error\_messages"] = error\_messages[f.name]  
 if field\_classes and f.name in field\_classes:  
 kwargs["form\_class"] = field\_classes[f.name]  
  
 if formfield\_callback is None:  
 formfield = f.formfield(\*\*kwargs)  
 elif not callable(formfield\_callback):  
 raise TypeError("formfield\_callback must be a function or callable")  
 else:  
 formfield = formfield\_callback(f, \*\*kwargs)  
  
 if formfield:  
 if apply\_limit\_choices\_to:  
 apply\_limit\_choices\_to\_to\_formfield(formfield)  
 field\_dict[f.name] = formfield  
 else:  
 ignored.append(f.name)  
 if fields:  
 field\_dict = {  
 f: field\_dict.get(f)  
 for f in fields  
 if (not exclude or f not in exclude) and f not in ignored  
 }  
 return field\_dict  
  
  
class ModelFormOptions:  
 def \_\_init\_\_(self, options=None):  
 self.model = getattr(options, "model", None)  
 self.fields = getattr(options, "fields", None)  
 self.exclude = getattr(options, "exclude", None)  
 self.widgets = getattr(options, "widgets", None)  
 self.localized\_fields = getattr(options, "localized\_fields", None)  
 self.labels = getattr(options, "labels", None)  
 self.help\_texts = getattr(options, "help\_texts", None)  
 self.error\_messages = getattr(options, "error\_messages", None)  
 self.field\_classes = getattr(options, "field\_classes", None)  
 self.formfield\_callback = getattr(options, "formfield\_callback", None)  
  
  
class ModelFormMetaclass(DeclarativeFieldsMetaclass):  
 def \_\_new\_\_(mcs, name, bases, attrs):  
 new\_class = super().\_\_new\_\_(mcs, name, bases, attrs)  
  
 if bases == (BaseModelForm,):  
 return new\_class  
  
 opts = new\_class.\_meta = ModelFormOptions(getattr(new\_class, "Meta", None))  
  
 # We check if a string was passed to `fields` or `exclude`,  
 # which is likely to be a mistake where the user typed ('foo') instead  
 # of ('foo',)  
 for opt in ["fields", "exclude", "localized\_fields"]:  
 value = getattr(opts, opt)  
 if isinstance(value, str) and value != ALL\_FIELDS:  
 msg = (  
 "%(model)s.Meta.%(opt)s cannot be a string. "  
 "Did you mean to type: ('%(value)s',)?"  
 % {  
 "model": new\_class.\_\_name\_\_,  
 "opt": opt,  
 "value": value,  
 }  
 )  
 raise TypeError(msg)  
  
 if opts.model:  
 # If a model is defined, extract form fields from it.  
 if opts.fields is None and opts.exclude is None:  
 raise ImproperlyConfigured(  
 "Creating a ModelForm without either the 'fields' attribute "  
 "or the 'exclude' attribute is prohibited; form %s "  
 "needs updating." % name  
 )  
  
 if opts.fields == ALL\_FIELDS:  
 # Sentinel for fields\_for\_model to indicate "get the list of  
 # fields from the model"  
 opts.fields = None  
  
 fields = fields\_for\_model(  
 opts.model,  
 opts.fields,  
 opts.exclude,  
 opts.widgets,  
 opts.formfield\_callback,  
 opts.localized\_fields,  
 opts.labels,  
 opts.help\_texts,  
 opts.error\_messages,  
 opts.field\_classes,  
 # limit\_choices\_to will be applied during ModelForm.\_\_init\_\_().  
 apply\_limit\_choices\_to=False,  
 )  
  
 # make sure opts.fields doesn't specify an invalid field  
 none\_model\_fields = {k for k, v in fields.items() if not v}  
 missing\_fields = none\_model\_fields.difference(new\_class.declared\_fields)  
 if missing\_fields:  
 message = "Unknown field(s) (%s) specified for %s"  
 message %= (", ".join(missing\_fields), opts.model.\_\_name\_\_)  
 raise FieldError(message)  
 # Override default model fields with any custom declared ones  
 # (plus, include all the other declared fields).  
 fields.update(new\_class.declared\_fields)  
 else:  
 fields = new\_class.declared\_fields  
  
 new\_class.base\_fields = fields  
  
 return new\_class  
  
  
class BaseModelForm(BaseForm, AltersData):  
 def \_\_init\_\_(  
 self,  
 data=None,  
 files=None,  
 auto\_id="id\_%s",  
 prefix=None,  
 initial=None,  
 error\_class=ErrorList,  
 label\_suffix=None,  
 empty\_permitted=False,  
 instance=None,  
 use\_required\_attribute=None,  
 renderer=None,  
 ):  
 opts = self.\_meta  
 if opts.model is None:  
 raise ValueError("ModelForm has no model class specified.")  
 if instance is None:  
 # if we didn't get an instance, instantiate a new one  
 self.instance = opts.model()  
 object\_data = {}  
 else:  
 self.instance = instance  
 object\_data = model\_to\_dict(instance, opts.fields, opts.exclude)  
 # if initial was provided, it should override the values from instance  
 if initial is not None:  
 object\_data.update(initial)  
 # self.\_validate\_unique will be set to True by BaseModelForm.clean().  
 # It is False by default so overriding self.clean() and failing to call  
 # super will stop validate\_unique from being called.  
 self.\_validate\_unique = False  
 super().\_\_init\_\_(  
 data,  
 files,  
 auto\_id,  
 prefix,  
 object\_data,  
 error\_class,  
 label\_suffix,  
 empty\_permitted,  
 use\_required\_attribute=use\_required\_attribute,  
 renderer=renderer,  
 )  
 for formfield in self.fields.values():  
 apply\_limit\_choices\_to\_to\_formfield(formfield)  
  
 def \_get\_validation\_exclusions(self):  
 """  
 For backwards-compatibility, exclude several types of fields from model  
 validation. See tickets #12507, #12521, #12553.  
 """  
 exclude = set()  
 # Build up a list of fields that should be excluded from model field  
 # validation and unique checks.  
 for f in self.instance.\_meta.fields:  
 field = f.name  
 # Exclude fields that aren't on the form. The developer may be  
 # adding these values to the model after form validation.  
 if field not in self.fields:  
 exclude.add(f.name)  
  
 # Don't perform model validation on fields that were defined  
 # manually on the form and excluded via the ModelForm's Meta  
 # class. See #12901.  
 elif self.\_meta.fields and field not in self.\_meta.fields:  
 exclude.add(f.name)  
 elif self.\_meta.exclude and field in self.\_meta.exclude:  
 exclude.add(f.name)  
  
 # Exclude fields that failed form validation. There's no need for  
 # the model fields to validate them as well.  
 elif field in self.\_errors:  
 exclude.add(f.name)  
  
 # Exclude empty fields that are not required by the form, if the  
 # underlying model field is required. This keeps the model field  
 # from raising a required error. Note: don't exclude the field from  
 # validation if the model field allows blanks. If it does, the blank  
 # value may be included in a unique check, so cannot be excluded  
 # from validation.  
 else:  
 form\_field = self.fields[field]  
 field\_value = self.cleaned\_data.get(field)  
 if (  
 not f.blank  
 and not form\_field.required  
 and field\_value in form\_field.empty\_values  
 ):  
 exclude.add(f.name)  
 return exclude  
  
 def clean(self):  
 self.\_validate\_unique = True  
 return self.cleaned\_data  
  
 def \_update\_errors(self, errors):  
 # Override any validation error messages defined at the model level  
 # with those defined at the form level.  
 opts = self.\_meta  
  
 # Allow the model generated by construct\_instance() to raise  
 # ValidationError and have them handled in the same way as others.  
 if hasattr(errors, "error\_dict"):  
 error\_dict = errors.error\_dict  
 else:  
 error\_dict = {NON\_FIELD\_ERRORS: errors}  
  
 for field, messages in error\_dict.items():  
 if (  
 field == NON\_FIELD\_ERRORS  
 and opts.error\_messages  
 and NON\_FIELD\_ERRORS in opts.error\_messages  
 ):  
 error\_messages = opts.error\_messages[NON\_FIELD\_ERRORS]  
 elif field in self.fields:  
 error\_messages = self.fields[field].error\_messages  
 else:  
 continue  
  
 for message in messages:  
 if (  
 isinstance(message, ValidationError)  
 and message.code in error\_messages  
 ):  
 message.message = error\_messages[message.code]  
  
 self.add\_error(None, errors)  
  
 def \_post\_clean(self):  
 opts = self.\_meta  
  
 exclude = self.\_get\_validation\_exclusions()  
  
 # Foreign Keys being used to represent inline relationships  
 # are excluded from basic field value validation. This is for two  
 # reasons: firstly, the value may not be supplied (#12507; the  
 # case of providing new values to the admin); secondly the  
 # object being referred to may not yet fully exist (#12749).  
 # However, these fields \*must\* be included in uniqueness checks,  
 # so this can't be part of \_get\_validation\_exclusions().  
 for name, field in self.fields.items():  
 if isinstance(field, InlineForeignKeyField):  
 exclude.add(name)  
  
 try:  
 self.instance = construct\_instance(  
 self, self.instance, opts.fields, opts.exclude  
 )  
 except ValidationError as e:  
 self.\_update\_errors(e)  
  
 try:  
 self.instance.full\_clean(exclude=exclude, validate\_unique=False)  
 except ValidationError as e:  
 self.\_update\_errors(e)  
  
 # Validate uniqueness if needed.  
 if self.\_validate\_unique:  
 self.validate\_unique()  
  
 def validate\_unique(self):  
 """  
 Call the instance's validate\_unique() method and update the form's  
 validation errors if any were raised.  
 """  
 exclude = self.\_get\_validation\_exclusions()  
 try:  
 self.instance.validate\_unique(exclude=exclude)  
 except ValidationError as e:  
 self.\_update\_errors(e)  
  
 def \_save\_m2m(self):  
 """  
 Save the many-to-many fields and generic relations for this form.  
 """  
 cleaned\_data = self.cleaned\_data  
 exclude = self.\_meta.exclude  
 fields = self.\_meta.fields  
 opts = self.instance.\_meta  
 # Note that for historical reasons we want to include also  
 # private\_fields here. (GenericRelation was previously a fake  
 # m2m field).  
 for f in chain(opts.many\_to\_many, opts.private\_fields):  
 if not hasattr(f, "save\_form\_data"):  
 continue  
 if fields and f.name not in fields:  
 continue  
 if exclude and f.name in exclude:  
 continue  
 if f.name in cleaned\_data:  
 f.save\_form\_data(self.instance, cleaned\_data[f.name])  
  
 def save(self, commit=True):  
 """  
 Save this form's self.instance object if commit=True. Otherwise, add  
 a save\_m2m() method to the form which can be called after the instance  
 is saved manually at a later time. Return the model instance.  
 """  
 if self.errors:  
 raise ValueError(  
 "The %s could not be %s because the data didn't validate."  
 % (  
 self.instance.\_meta.object\_name,  
 "created" if self.instance.\_state.adding else "changed",  
 )  
 )  
 if commit:  
 # If committing, save the instance and the m2m data immediately.  
 self.instance.save()  
 self.\_save\_m2m()  
 else:  
 # If not committing, add a method to the form to allow deferred  
 # saving of m2m data.  
 self.save\_m2m = self.\_save\_m2m  
 return self.instance  
  
 save.alters\_data = True  
  
  
class ModelForm(BaseModelForm, metaclass=ModelFormMetaclass):  
 pass  
  
  
def modelform\_factory(  
 model,  
 form=ModelForm,  
 fields=None,  
 exclude=None,  
 formfield\_callback=None,  
 widgets=None,  
 localized\_fields=None,  
 labels=None,  
 help\_texts=None,  
 error\_messages=None,  
 field\_classes=None,  
):  
 """  
 Return a ModelForm containing form fields for the given model. You can  
 optionally pass a `form` argument to use as a starting point for  
 constructing the ModelForm.  
  
 ``fields`` is an optional list of field names. If provided, include only  
 the named fields in the returned fields. If omitted or '\_\_all\_\_', use all  
 fields.  
  
 ``exclude`` is an optional list of field names. If provided, exclude the  
 named fields from the returned fields, even if they are listed in the  
 ``fields`` argument.  
  
 ``widgets`` is a dictionary of model field names mapped to a widget.  
  
 ``localized\_fields`` is a list of names of fields which should be localized.  
  
 ``formfield\_callback`` is a callable that takes a model field and returns  
 a form field.  
  
 ``labels`` is a dictionary of model field names mapped to a label.  
  
 ``help\_texts`` is a dictionary of model field names mapped to a help text.  
  
 ``error\_messages`` is a dictionary of model field names mapped to a  
 dictionary of error messages.  
  
 ``field\_classes`` is a dictionary of model field names mapped to a form  
 field class.  
 """  
 # Create the inner Meta class. FIXME: ideally, we should be able to  
 # construct a ModelForm without creating and passing in a temporary  
 # inner class.  
  
 # Build up a list of attributes that the Meta object will have.  
 attrs = {"model": model}  
 if fields is not None:  
 attrs["fields"] = fields  
 if exclude is not None:  
 attrs["exclude"] = exclude  
 if widgets is not None:  
 attrs["widgets"] = widgets  
 if localized\_fields is not None:  
 attrs["localized\_fields"] = localized\_fields  
 if labels is not None:  
 attrs["labels"] = labels  
 if help\_texts is not None:  
 attrs["help\_texts"] = help\_texts  
 if error\_messages is not None:  
 attrs["error\_messages"] = error\_messages  
 if field\_classes is not None:  
 attrs["field\_classes"] = field\_classes  
  
 # If parent form class already has an inner Meta, the Meta we're  
 # creating needs to inherit from the parent's inner meta.  
 bases = (form.Meta,) if hasattr(form, "Meta") else ()  
 Meta = type("Meta", bases, attrs)  
 if formfield\_callback:  
 Meta.formfield\_callback = staticmethod(formfield\_callback)  
 # Give this new form class a reasonable name.  
 class\_name = model.\_\_name\_\_ + "Form"  
  
 # Class attributes for the new form class.  
 form\_class\_attrs = {"Meta": Meta}  
  
 if getattr(Meta, "fields", None) is None and getattr(Meta, "exclude", None) is None:  
 raise ImproperlyConfigured(  
 "Calling modelform\_factory without defining 'fields' or "  
 "'exclude' explicitly is prohibited."  
 )  
  
 # Instantiate type(form) in order to use the same metaclass as form.  
 return type(form)(class\_name, (form,), form\_class\_attrs)  
  
  
# ModelFormSets ##############################################################  
  
  
class BaseModelFormSet(BaseFormSet, AltersData):  
 """  
 A ``FormSet`` for editing a queryset and/or adding new objects to it.  
 """  
  
 model = None  
 edit\_only = False  
  
 # Set of fields that must be unique among forms of this set.  
 unique\_fields = set()  
  
 def \_\_init\_\_(  
 self,  
 data=None,  
 files=None,  
 auto\_id="id\_%s",  
 prefix=None,  
 queryset=None,  
 \*,  
 initial=None,  
 \*\*kwargs,  
 ):  
 self.queryset = queryset  
 self.initial\_extra = initial  
 super().\_\_init\_\_(  
 \*\*{  
 "data": data,  
 "files": files,  
 "auto\_id": auto\_id,  
 "prefix": prefix,  
 \*\*kwargs,  
 }  
 )  
  
 def initial\_form\_count(self):  
 """Return the number of forms that are required in this FormSet."""  
 if not self.is\_bound:  
 return len(self.get\_queryset())  
 return super().initial\_form\_count()  
  
 def \_existing\_object(self, pk):  
 if not hasattr(self, "\_object\_dict"):  
 self.\_object\_dict = {o.pk: o for o in self.get\_queryset()}  
 return self.\_object\_dict.get(pk)  
  
 def \_get\_to\_python(self, field):  
 """  
 If the field is a related field, fetch the concrete field's (that  
 is, the ultimate pointed-to field's) to\_python.  
 """  
 while field.remote\_field is not None:  
 field = field.remote\_field.get\_related\_field()  
 return field.to\_python  
  
 def \_construct\_form(self, i, \*\*kwargs):  
 pk\_required = i < self.initial\_form\_count()  
 if pk\_required:  
 if self.is\_bound:  
 pk\_key = "%s-%s" % (self.add\_prefix(i), self.model.\_meta.pk.name)  
 try:  
 pk = self.data[pk\_key]  
 except KeyError:  
 # The primary key is missing. The user may have tampered  
 # with POST data.  
 pass  
 else:  
 to\_python = self.\_get\_to\_python(self.model.\_meta.pk)  
 try:  
 pk = to\_python(pk)  
 except ValidationError:  
 # The primary key exists but is an invalid value. The  
 # user may have tampered with POST data.  
 pass  
 else:  
 kwargs["instance"] = self.\_existing\_object(pk)  
 else:  
 kwargs["instance"] = self.get\_queryset()[i]  
 elif self.initial\_extra:  
 # Set initial values for extra forms  
 try:  
 kwargs["initial"] = self.initial\_extra[i - self.initial\_form\_count()]  
 except IndexError:  
 pass  
 form = super().\_construct\_form(i, \*\*kwargs)  
 if pk\_required:  
 form.fields[self.model.\_meta.pk.name].required = True  
 return form  
  
 def get\_queryset(self):  
 if not hasattr(self, "\_queryset"):  
 if self.queryset is not None:  
 qs = self.queryset  
 else:  
 qs = self.model.\_default\_manager.get\_queryset()  
  
 # If the queryset isn't already ordered we need to add an  
 # artificial ordering here to make sure that all formsets  
 # constructed from this queryset have the same form order.  
 if not qs.ordered:  
 qs = qs.order\_by(self.model.\_meta.pk.name)  
  
 # Removed queryset limiting here. As per discussion re: #13023  
 # on django-dev, max\_num should not prevent existing  
 # related objects/inlines from being displayed.  
 self.\_queryset = qs  
 return self.\_queryset  
  
 def save\_new(self, form, commit=True):  
 """Save and return a new model instance for the given form."""  
 return form.save(commit=commit)  
  
 def save\_existing(self, form, instance, commit=True):  
 """Save and return an existing model instance for the given form."""  
 return form.save(commit=commit)  
  
 def delete\_existing(self, obj, commit=True):  
 """Deletes an existing model instance."""  
 if commit:  
 obj.delete()  
  
 def save(self, commit=True):  
 """  
 Save model instances for every form, adding and changing instances  
 as necessary, and return the list of instances.  
 """  
 if not commit:  
 self.saved\_forms = []  
  
 def save\_m2m():  
 for form in self.saved\_forms:  
 form.save\_m2m()  
  
 self.save\_m2m = save\_m2m  
 if self.edit\_only:  
 return self.save\_existing\_objects(commit)  
 else:  
 return self.save\_existing\_objects(commit) + self.save\_new\_objects(commit)  
  
 save.alters\_data = True  
  
 def clean(self):  
 self.validate\_unique()  
  
 def validate\_unique(self):  
 # Collect unique\_checks and date\_checks to run from all the forms.  
 all\_unique\_checks = set()  
 all\_date\_checks = set()  
 forms\_to\_delete = self.deleted\_forms  
 valid\_forms = [  
 form  
 for form in self.forms  
 if form.is\_valid() and form not in forms\_to\_delete  
 ]  
 for form in valid\_forms:  
 exclude = form.\_get\_validation\_exclusions()  
 unique\_checks, date\_checks = form.instance.\_get\_unique\_checks(  
 exclude=exclude,  
 include\_meta\_constraints=True,  
 )  
 all\_unique\_checks.update(unique\_checks)  
 all\_date\_checks.update(date\_checks)  
  
 errors = []  
 # Do each of the unique checks (unique and unique\_together)  
 for uclass, unique\_check in all\_unique\_checks:  
 seen\_data = set()  
 for form in valid\_forms:  
 # Get the data for the set of fields that must be unique among  
 # the forms.  
 row\_data = (  
 field if field in self.unique\_fields else form.cleaned\_data[field]  
 for field in unique\_check  
 if field in form.cleaned\_data  
 )  
 # Reduce Model instances to their primary key values  
 row\_data = tuple(  
 d.\_get\_pk\_val() if hasattr(d, "\_get\_pk\_val")  
 # Prevent "unhashable type: list" errors later on.  
 else tuple(d) if isinstance(d, list) else d  
 for d in row\_data  
 )  
 if row\_data and None not in row\_data:  
 # if we've already seen it then we have a uniqueness failure  
 if row\_data in seen\_data:  
 # poke error messages into the right places and mark  
 # the form as invalid  
 errors.append(self.get\_unique\_error\_message(unique\_check))  
 form.\_errors[NON\_FIELD\_ERRORS] = self.error\_class(  
 [self.get\_form\_error()],  
 renderer=self.renderer,  
 )  
 # Remove the data from the cleaned\_data dict since it  
 # was invalid.  
 for field in unique\_check:  
 if field in form.cleaned\_data:  
 del form.cleaned\_data[field]  
 # mark the data as seen  
 seen\_data.add(row\_data)  
 # iterate over each of the date checks now  
 for date\_check in all\_date\_checks:  
 seen\_data = set()  
 uclass, lookup, field, unique\_for = date\_check  
 for form in valid\_forms:  
 # see if we have data for both fields  
 if (  
 form.cleaned\_data  
 and form.cleaned\_data[field] is not None  
 and form.cleaned\_data[unique\_for] is not None  
 ):  
 # if it's a date lookup we need to get the data for all the fields  
 if lookup == "date":  
 date = form.cleaned\_data[unique\_for]  
 date\_data = (date.year, date.month, date.day)  
 # otherwise it's just the attribute on the date/datetime  
 # object  
 else:  
 date\_data = (getattr(form.cleaned\_data[unique\_for], lookup),)  
 data = (form.cleaned\_data[field],) + date\_data  
 # if we've already seen it then we have a uniqueness failure  
 if data in seen\_data:  
 # poke error messages into the right places and mark  
 # the form as invalid  
 errors.append(self.get\_date\_error\_message(date\_check))  
 form.\_errors[NON\_FIELD\_ERRORS] = self.error\_class(  
 [self.get\_form\_error()],  
 renderer=self.renderer,  
 )  
 # Remove the data from the cleaned\_data dict since it  
 # was invalid.  
 del form.cleaned\_data[field]  
 # mark the data as seen  
 seen\_data.add(data)  
  
 if errors:  
 raise ValidationError(errors)  
  
 def get\_unique\_error\_message(self, unique\_check):  
 if len(unique\_check) == 1:  
 return gettext("Please correct the duplicate data for %(field)s.") % {  
 "field": unique\_check[0],  
 }  
 else:  
 return gettext(  
 "Please correct the duplicate data for %(field)s, which must be unique."  
 ) % {  
 "field": get\_text\_list(unique\_check, \_("and")),  
 }  
  
 def get\_date\_error\_message(self, date\_check):  
 return gettext(  
 "Please correct the duplicate data for %(field\_name)s "  
 "which must be unique for the %(lookup)s in %(date\_field)s."  
 ) % {  
 "field\_name": date\_check[2],  
 "date\_field": date\_check[3],  
 "lookup": str(date\_check[1]),  
 }  
  
 def get\_form\_error(self):  
 return gettext("Please correct the duplicate values below.")  
  
 def save\_existing\_objects(self, commit=True):  
 self.changed\_objects = []  
 self.deleted\_objects = []  
 if not self.initial\_forms:  
 return []  
  
 saved\_instances = []  
 forms\_to\_delete = self.deleted\_forms  
 for form in self.initial\_forms:  
 obj = form.instance  
 # If the pk is None, it means either:  
 # 1. The object is an unexpected empty model, created by invalid  
 # POST data such as an object outside the formset's queryset.  
 # 2. The object was already deleted from the database.  
 if obj.pk is None:  
 continue  
 if form in forms\_to\_delete:  
 self.deleted\_objects.append(obj)  
 self.delete\_existing(obj, commit=commit)  
 elif form.has\_changed():  
 self.changed\_objects.append((obj, form.changed\_data))  
 saved\_instances.append(self.save\_existing(form, obj, commit=commit))  
 if not commit:  
 self.saved\_forms.append(form)  
 return saved\_instances  
  
 def save\_new\_objects(self, commit=True):  
 self.new\_objects = []  
 for form in self.extra\_forms:  
 if not form.has\_changed():  
 continue  
 # If someone has marked an add form for deletion, don't save the  
 # object.  
 if self.can\_delete and self.\_should\_delete\_form(form):  
 continue  
 self.new\_objects.append(self.save\_new(form, commit=commit))  
 if not commit:  
 self.saved\_forms.append(form)  
 return self.new\_objects  
  
 def add\_fields(self, form, index):  
 """Add a hidden field for the object's primary key."""  
 from django.db.models import AutoField, ForeignKey, OneToOneField  
  
 self.\_pk\_field = pk = self.model.\_meta.pk  
 # If a pk isn't editable, then it won't be on the form, so we need to  
 # add it here so we can tell which object is which when we get the  
 # data back. Generally, pk.editable should be false, but for some  
 # reason, auto\_created pk fields and AutoField's editable attribute is  
 # True, so check for that as well.  
  
 def pk\_is\_not\_editable(pk):  
 return (  
 (not pk.editable)  
 or (pk.auto\_created or isinstance(pk, AutoField))  
 or (  
 pk.remote\_field  
 and pk.remote\_field.parent\_link  
 and pk\_is\_not\_editable(pk.remote\_field.model.\_meta.pk)  
 )  
 )  
  
 if pk\_is\_not\_editable(pk) or pk.name not in form.fields:  
 if form.is\_bound:  
 # If we're adding the related instance, ignore its primary key  
 # as it could be an auto-generated default which isn't actually  
 # in the database.  
 pk\_value = None if form.instance.\_state.adding else form.instance.pk  
 else:  
 try:  
 if index is not None:  
 pk\_value = self.get\_queryset()[index].pk  
 else:  
 pk\_value = None  
 except IndexError:  
 pk\_value = None  
 if isinstance(pk, (ForeignKey, OneToOneField)):  
 qs = pk.remote\_field.model.\_default\_manager.get\_queryset()  
 else:  
 qs = self.model.\_default\_manager.get\_queryset()  
 qs = qs.using(form.instance.\_state.db)  
 if form.\_meta.widgets:  
 widget = form.\_meta.widgets.get(self.\_pk\_field.name, HiddenInput)  
 else:  
 widget = HiddenInput  
 form.fields[self.\_pk\_field.name] = ModelChoiceField(  
 qs, initial=pk\_value, required=False, widget=widget  
 )  
 super().add\_fields(form, index)  
  
  
def modelformset\_factory(  
 model,  
 form=ModelForm,  
 formfield\_callback=None,  
 formset=BaseModelFormSet,  
 extra=1,  
 can\_delete=False,  
 can\_order=False,  
 max\_num=None,  
 fields=None,  
 exclude=None,  
 widgets=None,  
 validate\_max=False,  
 localized\_fields=None,  
 labels=None,  
 help\_texts=None,  
 error\_messages=None,  
 min\_num=None,  
 validate\_min=False,  
 field\_classes=None,  
 absolute\_max=None,  
 can\_delete\_extra=True,  
 renderer=None,  
 edit\_only=False,  
):  
 """Return a FormSet class for the given Django model class."""  
 meta = getattr(form, "Meta", None)  
 if (  
 getattr(meta, "fields", fields) is None  
 and getattr(meta, "exclude", exclude) is None  
 ):  
 raise ImproperlyConfigured(  
 "Calling modelformset\_factory without defining 'fields' or "  
 "'exclude' explicitly is prohibited."  
 )  
  
 form = modelform\_factory(  
 model,  
 form=form,  
 fields=fields,  
 exclude=exclude,  
 formfield\_callback=formfield\_callback,  
 widgets=widgets,  
 localized\_fields=localized\_fields,  
 labels=labels,  
 help\_texts=help\_texts,  
 error\_messages=error\_messages,  
 field\_classes=field\_classes,  
 )  
 FormSet = formset\_factory(  
 form,  
 formset,  
 extra=extra,  
 min\_num=min\_num,  
 max\_num=max\_num,  
 can\_order=can\_order,  
 can\_delete=can\_delete,  
 validate\_min=validate\_min,  
 validate\_max=validate\_max,  
 absolute\_max=absolute\_max,  
 can\_delete\_extra=can\_delete\_extra,  
 renderer=renderer,  
 )  
 FormSet.model = model  
 FormSet.edit\_only = edit\_only  
 return FormSet  
  
  
# InlineFormSets #############################################################  
  
  
class BaseInlineFormSet(BaseModelFormSet):  
 """A formset for child objects related to a parent."""  
  
 def \_\_init\_\_(  
 self,  
 data=None,  
 files=None,  
 instance=None,  
 save\_as\_new=False,  
 prefix=None,  
 queryset=None,  
 \*\*kwargs,  
 ):  
 if instance is None:  
 self.instance = self.fk.remote\_field.model()  
 else:  
 self.instance = instance  
 self.save\_as\_new = save\_as\_new  
 if queryset is None:  
 queryset = self.model.\_default\_manager  
 if self.instance.pk is not None:  
 qs = queryset.filter(\*\*{self.fk.name: self.instance})  
 else:  
 qs = queryset.none()  
 self.unique\_fields = {self.fk.name}  
 super().\_\_init\_\_(data, files, prefix=prefix, queryset=qs, \*\*kwargs)  
  
 # Add the generated field to form.\_meta.fields if it's defined to make  
 # sure validation isn't skipped on that field.  
 if self.form.\_meta.fields and self.fk.name not in self.form.\_meta.fields:  
 if isinstance(self.form.\_meta.fields, tuple):  
 self.form.\_meta.fields = list(self.form.\_meta.fields)  
 self.form.\_meta.fields.append(self.fk.name)  
  
 def initial\_form\_count(self):  
 if self.save\_as\_new:  
 return 0  
 return super().initial\_form\_count()  
  
 def \_construct\_form(self, i, \*\*kwargs):  
 form = super().\_construct\_form(i, \*\*kwargs)  
 if self.save\_as\_new:  
 mutable = getattr(form.data, "\_mutable", None)  
 # Allow modifying an immutable QueryDict.  
 if mutable is not None:  
 form.data.\_mutable = True  
 # Remove the primary key from the form's data, we are only  
 # creating new instances  
 form.data[form.add\_prefix(self.\_pk\_field.name)] = None  
 # Remove the foreign key from the form's data  
 form.data[form.add\_prefix(self.fk.name)] = None  
 if mutable is not None:  
 form.data.\_mutable = mutable  
  
 # Set the fk value here so that the form can do its validation.  
 fk\_value = self.instance.pk  
 if self.fk.remote\_field.field\_name != self.fk.remote\_field.model.\_meta.pk.name:  
 fk\_value = getattr(self.instance, self.fk.remote\_field.field\_name)  
 fk\_value = getattr(fk\_value, "pk", fk\_value)  
 setattr(form.instance, self.fk.get\_attname(), fk\_value)  
 return form  
  
 @classmethod  
 def get\_default\_prefix(cls):  
 return cls.fk.remote\_field.get\_accessor\_name(model=cls.model).replace("+", "")  
  
 def save\_new(self, form, commit=True):  
 # Ensure the latest copy of the related instance is present on each  
 # form (it may have been saved after the formset was originally  
 # instantiated).  
 setattr(form.instance, self.fk.name, self.instance)  
 return super().save\_new(form, commit=commit)  
  
 def add\_fields(self, form, index):  
 super().add\_fields(form, index)  
 if self.\_pk\_field == self.fk:  
 name = self.\_pk\_field.name  
 kwargs = {"pk\_field": True}  
 else:  
 # The foreign key field might not be on the form, so we poke at the  
 # Model field to get the label, since we need that for error messages.  
 name = self.fk.name  
 kwargs = {  
 "label": getattr(  
 form.fields.get(name), "label", capfirst(self.fk.verbose\_name)  
 )  
 }  
  
 # The InlineForeignKeyField assumes that the foreign key relation is  
 # based on the parent model's pk. If this isn't the case, set to\_field  
 # to correctly resolve the initial form value.  
 if self.fk.remote\_field.field\_name != self.fk.remote\_field.model.\_meta.pk.name:  
 kwargs["to\_field"] = self.fk.remote\_field.field\_name  
  
 # If we're adding a new object, ignore a parent's auto-generated key  
 # as it will be regenerated on the save request.  
 if self.instance.\_state.adding:  
 if kwargs.get("to\_field") is not None:  
 to\_field = self.instance.\_meta.get\_field(kwargs["to\_field"])  
 else:  
 to\_field = self.instance.\_meta.pk  
 if to\_field.has\_default():  
 setattr(self.instance, to\_field.attname, None)  
  
 form.fields[name] = InlineForeignKeyField(self.instance, \*\*kwargs)  
  
 def get\_unique\_error\_message(self, unique\_check):  
 unique\_check = [field for field in unique\_check if field != self.fk.name]  
 return super().get\_unique\_error\_message(unique\_check)  
  
  
def \_get\_foreign\_key(parent\_model, model, fk\_name=None, can\_fail=False):  
 """  
 Find and return the ForeignKey from model to parent if there is one  
 (return None if can\_fail is True and no such field exists). If fk\_name is  
 provided, assume it is the name of the ForeignKey field. Unless can\_fail is  
 True, raise an exception if there isn't a ForeignKey from model to  
 parent\_model.  
 """  
 # avoid circular import  
 from django.db.models import ForeignKey  
  
 opts = model.\_meta  
 if fk\_name:  
 fks\_to\_parent = [f for f in opts.fields if f.name == fk\_name]  
 if len(fks\_to\_parent) == 1:  
 fk = fks\_to\_parent[0]  
 parent\_list = parent\_model.\_meta.get\_parent\_list()  
 if (  
 not isinstance(fk, ForeignKey)  
 or (  
 # ForeignKey to proxy models.  
 fk.remote\_field.model.\_meta.proxy  
 and fk.remote\_field.model.\_meta.proxy\_for\_model not in parent\_list  
 )  
 or (  
 # ForeignKey to concrete models.  
 not fk.remote\_field.model.\_meta.proxy  
 and fk.remote\_field.model != parent\_model  
 and fk.remote\_field.model not in parent\_list  
 )  
 ):  
 raise ValueError(  
 "fk\_name '%s' is not a ForeignKey to '%s'."  
 % (fk\_name, parent\_model.\_meta.label)  
 )  
 elif not fks\_to\_parent:  
 raise ValueError(  
 "'%s' has no field named '%s'." % (model.\_meta.label, fk\_name)  
 )  
 else:  
 # Try to discover what the ForeignKey from model to parent\_model is  
 parent\_list = parent\_model.\_meta.get\_parent\_list()  
 fks\_to\_parent = [  
 f  
 for f in opts.fields  
 if isinstance(f, ForeignKey)  
 and (  
 f.remote\_field.model == parent\_model  
 or f.remote\_field.model in parent\_list  
 or (  
 f.remote\_field.model.\_meta.proxy  
 and f.remote\_field.model.\_meta.proxy\_for\_model in parent\_list  
 )  
 )  
 ]  
 if len(fks\_to\_parent) == 1:  
 fk = fks\_to\_parent[0]  
 elif not fks\_to\_parent:  
 if can\_fail:  
 return  
 raise ValueError(  
 "'%s' has no ForeignKey to '%s'."  
 % (  
 model.\_meta.label,  
 parent\_model.\_meta.label,  
 )  
 )  
 else:  
 raise ValueError(  
 "'%s' has more than one ForeignKey to '%s'. You must specify "  
 "a 'fk\_name' attribute."  
 % (  
 model.\_meta.label,  
 parent\_model.\_meta.label,  
 )  
 )  
 return fk  
  
  
def inlineformset\_factory(  
 parent\_model,  
 model,  
 form=ModelForm,  
 formset=BaseInlineFormSet,  
 fk\_name=None,  
 fields=None,  
 exclude=None,  
 extra=3,  
 can\_order=False,  
 can\_delete=True,  
 max\_num=None,  
 formfield\_callback=None,  
 widgets=None,  
 validate\_max=False,  
 localized\_fields=None,  
 labels=None,  
 help\_texts=None,  
 error\_messages=None,  
 min\_num=None,  
 validate\_min=False,  
 field\_classes=None,  
 absolute\_max=None,  
 can\_delete\_extra=True,  
 renderer=None,  
 edit\_only=False,  
):  
 """  
 Return an ``InlineFormSet`` for the given kwargs.  
  
 ``fk\_name`` must be provided if ``model`` has more than one ``ForeignKey``  
 to ``parent\_model``.  
 """  
 fk = \_get\_foreign\_key(parent\_model, model, fk\_name=fk\_name)  
 # enforce a max\_num=1 when the foreign key to the parent model is unique.  
 if fk.unique:  
 max\_num = 1  
 kwargs = {  
 "form": form,  
 "formfield\_callback": formfield\_callback,  
 "formset": formset,  
 "extra": extra,  
 "can\_delete": can\_delete,  
 "can\_order": can\_order,  
 "fields": fields,  
 "exclude": exclude,  
 "min\_num": min\_num,  
 "max\_num": max\_num,  
 "widgets": widgets,  
 "validate\_min": validate\_min,  
 "validate\_max": validate\_max,  
 "localized\_fields": localized\_fields,  
 "labels": labels,  
 "help\_texts": help\_texts,  
 "error\_messages": error\_messages,  
 "field\_classes": field\_classes,  
 "absolute\_max": absolute\_max,  
 "can\_delete\_extra": can\_delete\_extra,  
 "renderer": renderer,  
 "edit\_only": edit\_only,  
 }  
 FormSet = modelformset\_factory(model, \*\*kwargs)  
 FormSet.fk = fk  
 return FormSet  
  
  
# Fields #####################################################################  
  
  
class InlineForeignKeyField(Field):  
 """  
 A basic integer field that deals with validating the given value to a  
 given parent instance in an inline.  
 """  
  
 widget = HiddenInput  
 default\_error\_messages = {  
 "invalid\_choice": \_("The inline value did not match the parent instance."),  
 }  
  
 def \_\_init\_\_(self, parent\_instance, \*args, pk\_field=False, to\_field=None, \*\*kwargs):  
 self.parent\_instance = parent\_instance  
 self.pk\_field = pk\_field  
 self.to\_field = to\_field  
 if self.parent\_instance is not None:  
 if self.to\_field:  
 kwargs["initial"] = getattr(self.parent\_instance, self.to\_field)  
 else:  
 kwargs["initial"] = self.parent\_instance.pk  
 kwargs["required"] = False  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 def clean(self, value):  
 if value in self.empty\_values:  
 if self.pk\_field:  
 return None  
 # if there is no value act as we did before.  
 return self.parent\_instance  
 # ensure the we compare the values as equal types.  
 if self.to\_field:  
 orig = getattr(self.parent\_instance, self.to\_field)  
 else:  
 orig = self.parent\_instance.pk  
 if str(value) != str(orig):  
 raise ValidationError(  
 self.error\_messages["invalid\_choice"], code="invalid\_choice"  
 )  
 return self.parent\_instance  
  
 def has\_changed(self, initial, data):  
 return False  
  
  
class ModelChoiceIteratorValue:  
 def \_\_init\_\_(self, value, instance):  
 self.value = value  
 self.instance = instance  
  
 def \_\_str\_\_(self):  
 return str(self.value)  
  
 def \_\_hash\_\_(self):  
 return hash(self.value)  
  
 def \_\_eq\_\_(self, other):  
 if isinstance(other, ModelChoiceIteratorValue):  
 other = other.value  
 return self.value == other  
  
  
class ModelChoiceIterator:  
 def \_\_init\_\_(self, field):  
 self.field = field  
 self.queryset = field.queryset  
  
 def \_\_iter\_\_(self):  
 if self.field.empty\_label is not None:  
 yield ("", self.field.empty\_label)  
 queryset = self.queryset  
 # Can't use iterator() when queryset uses prefetch\_related()  
 if not queryset.\_prefetch\_related\_lookups:  
 queryset = queryset.iterator()  
 for obj in queryset:  
 yield self.choice(obj)  
  
 def \_\_len\_\_(self):  
 # count() adds a query but uses less memory since the QuerySet results  
 # won't be cached. In most cases, the choices will only be iterated on,  
 # and \_\_len\_\_() won't be called.  
 return self.queryset.count() + (1 if self.field.empty\_label is not None else 0)  
  
 def \_\_bool\_\_(self):  
 return self.field.empty\_label is not None or self.queryset.exists()  
  
 def choice(self, obj):  
 return (  
 ModelChoiceIteratorValue(self.field.prepare\_value(obj), obj),  
 self.field.label\_from\_instance(obj),  
 )  
  
  
class ModelChoiceField(ChoiceField):  
 """A ChoiceField whose choices are a model QuerySet."""  
  
 # This class is a subclass of ChoiceField for purity, but it doesn't  
 # actually use any of ChoiceField's implementation.  
 default\_error\_messages = {  
 "invalid\_choice": \_(  
 "Select a valid choice. That choice is not one of the available choices."  
 ),  
 }  
 iterator = ModelChoiceIterator  
  
 def \_\_init\_\_(  
 self,  
 queryset,  
 \*,  
 empty\_label="---------",  
 required=True,  
 widget=None,  
 label=None,  
 initial=None,  
 help\_text="",  
 to\_field\_name=None,  
 limit\_choices\_to=None,  
 blank=False,  
 \*\*kwargs,  
 ):  
 # Call Field instead of ChoiceField \_\_init\_\_() because we don't need  
 # ChoiceField.\_\_init\_\_().  
 Field.\_\_init\_\_(  
 self,  
 required=required,  
 widget=widget,  
 label=label,  
 initial=initial,  
 help\_text=help\_text,  
 \*\*kwargs,  
 )  
 if (required and initial is not None) or (  
 isinstance(self.widget, RadioSelect) and not blank  
 ):  
 self.empty\_label = None  
 else:  
 self.empty\_label = empty\_label  
 self.queryset = queryset  
 self.limit\_choices\_to = limit\_choices\_to # limit the queryset later.  
 self.to\_field\_name = to\_field\_name  
  
 def get\_limit\_choices\_to(self):  
 """  
 Return ``limit\_choices\_to`` for this form field.  
  
 If it is a callable, invoke it and return the result.  
 """  
 if callable(self.limit\_choices\_to):  
 return self.limit\_choices\_to()  
 return self.limit\_choices\_to  
  
 def \_\_deepcopy\_\_(self, memo):  
 result = super(ChoiceField, self).\_\_deepcopy\_\_(memo)  
 # Need to force a new ModelChoiceIterator to be created, bug #11183  
 if self.queryset is not None:  
 result.queryset = self.queryset.all()  
 return result  
  
 def \_get\_queryset(self):  
 return self.\_queryset  
  
 def \_set\_queryset(self, queryset):  
 self.\_queryset = None if queryset is None else queryset.all()  
 self.widget.choices = self.choices  
  
 queryset = property(\_get\_queryset, \_set\_queryset)  
  
 # this method will be used to create object labels by the QuerySetIterator.  
 # Override it to customize the label.  
 def label\_from\_instance(self, obj):  
 """  
 Convert objects into strings and generate the labels for the choices  
 presented by this object. Subclasses can override this method to  
 customize the display of the choices.  
 """  
 return str(obj)  
  
 def \_get\_choices(self):  
 # If self.\_choices is set, then somebody must have manually set  
 # the property self.choices. In this case, just return self.\_choices.  
 if hasattr(self, "\_choices"):  
 return self.\_choices  
  
 # Otherwise, execute the QuerySet in self.queryset to determine the  
 # choices dynamically. Return a fresh ModelChoiceIterator that has not been  
 # consumed. Note that we're instantiating a new ModelChoiceIterator \*each\*  
 # time \_get\_choices() is called (and, thus, each time self.choices is  
 # accessed) so that we can ensure the QuerySet has not been consumed. This  
 # construct might look complicated but it allows for lazy evaluation of  
 # the queryset.  
 return self.iterator(self)  
  
 choices = property(\_get\_choices, ChoiceField.\_set\_choices)  
  
 def prepare\_value(self, value):  
 if hasattr(value, "\_meta"):  
 if self.to\_field\_name:  
 return value.serializable\_value(self.to\_field\_name)  
 else:  
 return value.pk  
 return super().prepare\_value(value)  
  
 def to\_python(self, value):  
 if value in self.empty\_values:  
 return None  
 try:  
 key = self.to\_field\_name or "pk"  
 if isinstance(value, self.queryset.model):  
 value = getattr(value, key)  
 value = self.queryset.get(\*\*{key: value})  
 except (ValueError, TypeError, self.queryset.model.DoesNotExist):  
 raise ValidationError(  
 self.error\_messages["invalid\_choice"],  
 code="invalid\_choice",  
 params={"value": value},  
 )  
 return value  
  
 def validate(self, value):  
 return Field.validate(self, value)  
  
 def has\_changed(self, initial, data):  
 if self.disabled:  
 return False  
 initial\_value = initial if initial is not None else ""  
 data\_value = data if data is not None else ""  
 return str(self.prepare\_value(initial\_value)) != str(data\_value)  
  
  
class ModelMultipleChoiceField(ModelChoiceField):  
 """A MultipleChoiceField whose choices are a model QuerySet."""  
  
 widget = SelectMultiple  
 hidden\_widget = MultipleHiddenInput  
 default\_error\_messages = {  
 "invalid\_list": \_("Enter a list of values."),  
 "invalid\_choice": \_(  
 "Select a valid choice. %(value)s is not one of the available choices."  
 ),  
 "invalid\_pk\_value": \_("“%(pk)s” is not a valid value."),  
 }  
  
 def \_\_init\_\_(self, queryset, \*\*kwargs):  
 super().\_\_init\_\_(queryset, empty\_label=None, \*\*kwargs)  
  
 def to\_python(self, value):  
 if not value:  
 return []  
 return list(self.\_check\_values(value))  
  
 def clean(self, value):  
 value = self.prepare\_value(value)  
 if self.required and not value:  
 raise ValidationError(self.error\_messages["required"], code="required")  
 elif not self.required and not value:  
 return self.queryset.none()  
 if not isinstance(value, (list, tuple)):  
 raise ValidationError(  
 self.error\_messages["invalid\_list"],  
 code="invalid\_list",  
 )  
 qs = self.\_check\_values(value)  
 # Since this overrides the inherited ModelChoiceField.clean  
 # we run custom validators here  
 self.run\_validators(value)  
 return qs  
  
 def \_check\_values(self, value):  
 """  
 Given a list of possible PK values, return a QuerySet of the  
 corresponding objects. Raise a ValidationError if a given value is  
 invalid (not a valid PK, not in the queryset, etc.)  
 """  
 key = self.to\_field\_name or "pk"  
 # deduplicate given values to avoid creating many querysets or  
 # requiring the database backend deduplicate efficiently.  
 try:  
 value = frozenset(value)  
 except TypeError:  
 # list of lists isn't hashable, for example  
 raise ValidationError(  
 self.error\_messages["invalid\_list"],  
 code="invalid\_list",  
 )  
 for pk in value:  
 try:  
 self.queryset.filter(\*\*{key: pk})  
 except (ValueError, TypeError):  
 raise ValidationError(  
 self.error\_messages["invalid\_pk\_value"],  
 code="invalid\_pk\_value",  
 params={"pk": pk},  
 )  
 qs = self.queryset.filter(\*\*{"%s\_\_in" % key: value})  
 pks = {str(getattr(o, key)) for o in qs}  
 for val in value:  
 if str(val) not in pks:  
 raise ValidationError(  
 self.error\_messages["invalid\_choice"],  
 code="invalid\_choice",  
 params={"value": val},  
 )  
 return qs  
  
 def prepare\_value(self, value):  
 if (  
 hasattr(value, "\_\_iter\_\_")  
 and not isinstance(value, str)  
 and not hasattr(value, "\_meta")  
 ):  
 prepare\_value = super().prepare\_value  
 return [prepare\_value(v) for v in value]  
 return super().prepare\_value(value)  
  
 def has\_changed(self, initial, data):  
 if self.disabled:  
 return False  
 if initial is None:  
 initial = []  
 if data is None:  
 data = []  
 if len(initial) != len(data):  
 return True  
 initial\_set = {str(value) for value in self.prepare\_value(initial)}  
 data\_set = {str(value) for value in data}  
 return data\_set != initial\_set  
  
  
def modelform\_defines\_fields(form\_class):  
 return hasattr(form\_class, "\_meta") and (  
 form\_class.\_meta.fields is not None or form\_class.\_meta.exclude is not None  
 )

## File: venv/lib/python3.12/site-packages/pip/\_internal/utils/models.py

"""Utilities for defining models  
"""  
  
import operator  
from typing import Any, Callable, Type  
  
  
class KeyBasedCompareMixin:  
 """Provides comparison capabilities that is based on a key"""  
  
 \_\_slots\_\_ = ["\_compare\_key", "\_defining\_class"]  
  
 def \_\_init\_\_(self, key: Any, defining\_class: Type["KeyBasedCompareMixin"]) -> None:  
 self.\_compare\_key = key  
 self.\_defining\_class = defining\_class  
  
 def \_\_hash\_\_(self) -> int:  
 return hash(self.\_compare\_key)  
  
 def \_\_lt\_\_(self, other: Any) -> bool:  
 return self.\_compare(other, operator.\_\_lt\_\_)  
  
 def \_\_le\_\_(self, other: Any) -> bool:  
 return self.\_compare(other, operator.\_\_le\_\_)  
  
 def \_\_gt\_\_(self, other: Any) -> bool:  
 return self.\_compare(other, operator.\_\_gt\_\_)  
  
 def \_\_ge\_\_(self, other: Any) -> bool:  
 return self.\_compare(other, operator.\_\_ge\_\_)  
  
 def \_\_eq\_\_(self, other: Any) -> bool:  
 return self.\_compare(other, operator.\_\_eq\_\_)  
  
 def \_compare(self, other: Any, method: Callable[[Any, Any], bool]) -> bool:  
 if not isinstance(other, self.\_defining\_class):  
 return NotImplemented  
  
 return method(self.\_compare\_key, other.\_compare\_key)

## File: venv/lib/python3.12/site-packages/pip/\_vendor/requests/models.py

"""  
requests.models  
~~~~~~~~~~~~~~~  
  
This module contains the primary objects that power Requests.  
"""  
  
import datetime  
  
# Import encoding now, to avoid implicit import later.  
# Implicit import within threads may cause LookupError when standard library is in a ZIP,  
# such as in Embedded Python. See https://github.com/psf/requests/issues/3578.  
import encodings.idna # noqa: F401  
from io import UnsupportedOperation  
  
from pip.\_vendor.urllib3.exceptions import (  
 DecodeError,  
 LocationParseError,  
 ProtocolError,  
 ReadTimeoutError,  
 SSLError,  
)  
from pip.\_vendor.urllib3.fields import RequestField  
from pip.\_vendor.urllib3.filepost import encode\_multipart\_formdata  
from pip.\_vendor.urllib3.util import parse\_url  
  
from .\_internal\_utils import to\_native\_string, unicode\_is\_ascii  
from .auth import HTTPBasicAuth  
from .compat import (  
 Callable,  
 JSONDecodeError,  
 Mapping,  
 basestring,  
 builtin\_str,  
 chardet,  
 cookielib,  
)  
from .compat import json as complexjson  
from .compat import urlencode, urlsplit, urlunparse  
from .cookies import \_copy\_cookie\_jar, cookiejar\_from\_dict, get\_cookie\_header  
from .exceptions import (  
 ChunkedEncodingError,  
 ConnectionError,  
 ContentDecodingError,  
 HTTPError,  
 InvalidJSONError,  
 InvalidURL,  
)  
from .exceptions import JSONDecodeError as RequestsJSONDecodeError  
from .exceptions import MissingSchema  
from .exceptions import SSLError as RequestsSSLError  
from .exceptions import StreamConsumedError  
from .hooks import default\_hooks  
from .status\_codes import codes  
from .structures import CaseInsensitiveDict  
from .utils import (  
 check\_header\_validity,  
 get\_auth\_from\_url,  
 guess\_filename,  
 guess\_json\_utf,  
 iter\_slices,  
 parse\_header\_links,  
 requote\_uri,  
 stream\_decode\_response\_unicode,  
 super\_len,  
 to\_key\_val\_list,  
)  
  
#: The set of HTTP status codes that indicate an automatically  
#: processable redirect.  
REDIRECT\_STATI = (  
 codes.moved, # 301  
 codes.found, # 302  
 codes.other, # 303  
 codes.temporary\_redirect, # 307  
 codes.permanent\_redirect, # 308  
)  
  
DEFAULT\_REDIRECT\_LIMIT = 30  
CONTENT\_CHUNK\_SIZE = 10 \* 1024  
ITER\_CHUNK\_SIZE = 512  
  
  
class RequestEncodingMixin:  
 @property  
 def path\_url(self):  
 """Build the path URL to use."""  
  
 url = []  
  
 p = urlsplit(self.url)  
  
 path = p.path  
 if not path:  
 path = "/"  
  
 url.append(path)  
  
 query = p.query  
 if query:  
 url.append("?")  
 url.append(query)  
  
 return "".join(url)  
  
 @staticmethod  
 def \_encode\_params(data):  
 """Encode parameters in a piece of data.  
  
 Will successfully encode parameters when passed as a dict or a list of  
 2-tuples. Order is retained if data is a list of 2-tuples but arbitrary  
 if parameters are supplied as a dict.  
 """  
  
 if isinstance(data, (str, bytes)):  
 return data  
 elif hasattr(data, "read"):  
 return data  
 elif hasattr(data, "\_\_iter\_\_"):  
 result = []  
 for k, vs in to\_key\_val\_list(data):  
 if isinstance(vs, basestring) or not hasattr(vs, "\_\_iter\_\_"):  
 vs = [vs]  
 for v in vs:  
 if v is not None:  
 result.append(  
 (  
 k.encode("utf-8") if isinstance(k, str) else k,  
 v.encode("utf-8") if isinstance(v, str) else v,  
 )  
 )  
 return urlencode(result, doseq=True)  
 else:  
 return data  
  
 @staticmethod  
 def \_encode\_files(files, data):  
 """Build the body for a multipart/form-data request.  
  
 Will successfully encode files when passed as a dict or a list of  
 tuples. Order is retained if data is a list of tuples but arbitrary  
 if parameters are supplied as a dict.  
 The tuples may be 2-tuples (filename, fileobj), 3-tuples (filename, fileobj, contentype)  
 or 4-tuples (filename, fileobj, contentype, custom\_headers).  
 """  
 if not files:  
 raise ValueError("Files must be provided.")  
 elif isinstance(data, basestring):  
 raise ValueError("Data must not be a string.")  
  
 new\_fields = []  
 fields = to\_key\_val\_list(data or {})  
 files = to\_key\_val\_list(files or {})  
  
 for field, val in fields:  
 if isinstance(val, basestring) or not hasattr(val, "\_\_iter\_\_"):  
 val = [val]  
 for v in val:  
 if v is not None:  
 # Don't call str() on bytestrings: in Py3 it all goes wrong.  
 if not isinstance(v, bytes):  
 v = str(v)  
  
 new\_fields.append(  
 (  
 field.decode("utf-8")  
 if isinstance(field, bytes)  
 else field,  
 v.encode("utf-8") if isinstance(v, str) else v,  
 )  
 )  
  
 for (k, v) in files:  
 # support for explicit filename  
 ft = None  
 fh = None  
 if isinstance(v, (tuple, list)):  
 if len(v) == 2:  
 fn, fp = v  
 elif len(v) == 3:  
 fn, fp, ft = v  
 else:  
 fn, fp, ft, fh = v  
 else:  
 fn = guess\_filename(v) or k  
 fp = v  
  
 if isinstance(fp, (str, bytes, bytearray)):  
 fdata = fp  
 elif hasattr(fp, "read"):  
 fdata = fp.read()  
 elif fp is None:  
 continue  
 else:  
 fdata = fp  
  
 rf = RequestField(name=k, data=fdata, filename=fn, headers=fh)  
 rf.make\_multipart(content\_type=ft)  
 new\_fields.append(rf)  
  
 body, content\_type = encode\_multipart\_formdata(new\_fields)  
  
 return body, content\_type  
  
  
class RequestHooksMixin:  
 def register\_hook(self, event, hook):  
 """Properly register a hook."""  
  
 if event not in self.hooks:  
 raise ValueError(f'Unsupported event specified, with event name "{event}"')  
  
 if isinstance(hook, Callable):  
 self.hooks[event].append(hook)  
 elif hasattr(hook, "\_\_iter\_\_"):  
 self.hooks[event].extend(h for h in hook if isinstance(h, Callable))  
  
 def deregister\_hook(self, event, hook):  
 """Deregister a previously registered hook.  
 Returns True if the hook existed, False if not.  
 """  
  
 try:  
 self.hooks[event].remove(hook)  
 return True  
 except ValueError:  
 return False  
  
  
class Request(RequestHooksMixin):  
 """A user-created :class:`Request <Request>` object.  
  
 Used to prepare a :class:`PreparedRequest <PreparedRequest>`, which is sent to the server.  
  
 :param method: HTTP method to use.  
 :param url: URL to send.  
 :param headers: dictionary of headers to send.  
 :param files: dictionary of {filename: fileobject} files to multipart upload.  
 :param data: the body to attach to the request. If a dictionary or  
 list of tuples ``[(key, value)]`` is provided, form-encoding will  
 take place.  
 :param json: json for the body to attach to the request (if files or data is not specified).  
 :param params: URL parameters to append to the URL. If a dictionary or  
 list of tuples ``[(key, value)]`` is provided, form-encoding will  
 take place.  
 :param auth: Auth handler or (user, pass) tuple.  
 :param cookies: dictionary or CookieJar of cookies to attach to this request.  
 :param hooks: dictionary of callback hooks, for internal usage.  
  
 Usage::  
  
 >>> import requests  
 >>> req = requests.Request('GET', 'https://httpbin.org/get')  
 >>> req.prepare()  
 <PreparedRequest [GET]>  
 """  
  
 def \_\_init\_\_(  
 self,  
 method=None,  
 url=None,  
 headers=None,  
 files=None,  
 data=None,  
 params=None,  
 auth=None,  
 cookies=None,  
 hooks=None,  
 json=None,  
 ):  
  
 # Default empty dicts for dict params.  
 data = [] if data is None else data  
 files = [] if files is None else files  
 headers = {} if headers is None else headers  
 params = {} if params is None else params  
 hooks = {} if hooks is None else hooks  
  
 self.hooks = default\_hooks()  
 for (k, v) in list(hooks.items()):  
 self.register\_hook(event=k, hook=v)  
  
 self.method = method  
 self.url = url  
 self.headers = headers  
 self.files = files  
 self.data = data  
 self.json = json  
 self.params = params  
 self.auth = auth  
 self.cookies = cookies  
  
 def \_\_repr\_\_(self):  
 return f"<Request [{self.method}]>"  
  
 def prepare(self):  
 """Constructs a :class:`PreparedRequest <PreparedRequest>` for transmission and returns it."""  
 p = PreparedRequest()  
 p.prepare(  
 method=self.method,  
 url=self.url,  
 headers=self.headers,  
 files=self.files,  
 data=self.data,  
 json=self.json,  
 params=self.params,  
 auth=self.auth,  
 cookies=self.cookies,  
 hooks=self.hooks,  
 )  
 return p  
  
  
class PreparedRequest(RequestEncodingMixin, RequestHooksMixin):  
 """The fully mutable :class:`PreparedRequest <PreparedRequest>` object,  
 containing the exact bytes that will be sent to the server.  
  
 Instances are generated from a :class:`Request <Request>` object, and  
 should not be instantiated manually; doing so may produce undesirable  
 effects.  
  
 Usage::  
  
 >>> import requests  
 >>> req = requests.Request('GET', 'https://httpbin.org/get')  
 >>> r = req.prepare()  
 >>> r  
 <PreparedRequest [GET]>  
  
 >>> s = requests.Session()  
 >>> s.send(r)  
 <Response [200]>  
 """  
  
 def \_\_init\_\_(self):  
 #: HTTP verb to send to the server.  
 self.method = None  
 #: HTTP URL to send the request to.  
 self.url = None  
 #: dictionary of HTTP headers.  
 self.headers = None  
 # The `CookieJar` used to create the Cookie header will be stored here  
 # after prepare\_cookies is called  
 self.\_cookies = None  
 #: request body to send to the server.  
 self.body = None  
 #: dictionary of callback hooks, for internal usage.  
 self.hooks = default\_hooks()  
 #: integer denoting starting position of a readable file-like body.  
 self.\_body\_position = None  
  
 def prepare(  
 self,  
 method=None,  
 url=None,  
 headers=None,  
 files=None,  
 data=None,  
 params=None,  
 auth=None,  
 cookies=None,  
 hooks=None,  
 json=None,  
 ):  
 """Prepares the entire request with the given parameters."""  
  
 self.prepare\_method(method)  
 self.prepare\_url(url, params)  
 self.prepare\_headers(headers)  
 self.prepare\_cookies(cookies)  
 self.prepare\_body(data, files, json)  
 self.prepare\_auth(auth, url)  
  
 # Note that prepare\_auth must be last to enable authentication schemes  
 # such as OAuth to work on a fully prepared request.  
  
 # This MUST go after prepare\_auth. Authenticators could add a hook  
 self.prepare\_hooks(hooks)  
  
 def \_\_repr\_\_(self):  
 return f"<PreparedRequest [{self.method}]>"  
  
 def copy(self):  
 p = PreparedRequest()  
 p.method = self.method  
 p.url = self.url  
 p.headers = self.headers.copy() if self.headers is not None else None  
 p.\_cookies = \_copy\_cookie\_jar(self.\_cookies)  
 p.body = self.body  
 p.hooks = self.hooks  
 p.\_body\_position = self.\_body\_position  
 return p  
  
 def prepare\_method(self, method):  
 """Prepares the given HTTP method."""  
 self.method = method  
 if self.method is not None:  
 self.method = to\_native\_string(self.method.upper())  
  
 @staticmethod  
 def \_get\_idna\_encoded\_host(host):  
 from pip.\_vendor import idna  
  
 try:  
 host = idna.encode(host, uts46=True).decode("utf-8")  
 except idna.IDNAError:  
 raise UnicodeError  
 return host  
  
 def prepare\_url(self, url, params):  
 """Prepares the given HTTP URL."""  
 #: Accept objects that have string representations.  
 #: We're unable to blindly call unicode/str functions  
 #: as this will include the bytestring indicator (b'')  
 #: on python 3.x.  
 #: https://github.com/psf/requests/pull/2238  
 if isinstance(url, bytes):  
 url = url.decode("utf8")  
 else:  
 url = str(url)  
  
 # Remove leading whitespaces from url  
 url = url.lstrip()  
  
 # Don't do any URL preparation for non-HTTP schemes like `mailto`,  
 # `data` etc to work around exceptions from `url\_parse`, which  
 # handles RFC 3986 only.  
 if ":" in url and not url.lower().startswith("http"):  
 self.url = url  
 return  
  
 # Support for unicode domain names and paths.  
 try:  
 scheme, auth, host, port, path, query, fragment = parse\_url(url)  
 except LocationParseError as e:  
 raise InvalidURL(\*e.args)  
  
 if not scheme:  
 raise MissingSchema(  
 f"Invalid URL {url!r}: No scheme supplied. "  
 f"Perhaps you meant https://{url}?"  
 )  
  
 if not host:  
 raise InvalidURL(f"Invalid URL {url!r}: No host supplied")  
  
 # In general, we want to try IDNA encoding the hostname if the string contains  
 # non-ASCII characters. This allows users to automatically get the correct IDNA  
 # behaviour. For strings containing only ASCII characters, we need to also verify  
 # it doesn't start with a wildcard (\*), before allowing the unencoded hostname.  
 if not unicode\_is\_ascii(host):  
 try:  
 host = self.\_get\_idna\_encoded\_host(host)  
 except UnicodeError:  
 raise InvalidURL("URL has an invalid label.")  
 elif host.startswith(("\*", ".")):  
 raise InvalidURL("URL has an invalid label.")  
  
 # Carefully reconstruct the network location  
 netloc = auth or ""  
 if netloc:  
 netloc += "@"  
 netloc += host  
 if port:  
 netloc += f":{port}"  
  
 # Bare domains aren't valid URLs.  
 if not path:  
 path = "/"  
  
 if isinstance(params, (str, bytes)):  
 params = to\_native\_string(params)  
  
 enc\_params = self.\_encode\_params(params)  
 if enc\_params:  
 if query:  
 query = f"{query}&{enc\_params}"  
 else:  
 query = enc\_params  
  
 url = requote\_uri(urlunparse([scheme, netloc, path, None, query, fragment]))  
 self.url = url  
  
 def prepare\_headers(self, headers):  
 """Prepares the given HTTP headers."""  
  
 self.headers = CaseInsensitiveDict()  
 if headers:  
 for header in headers.items():  
 # Raise exception on invalid header value.  
 check\_header\_validity(header)  
 name, value = header  
 self.headers[to\_native\_string(name)] = value  
  
 def prepare\_body(self, data, files, json=None):  
 """Prepares the given HTTP body data."""  
  
 # Check if file, fo, generator, iterator.  
 # If not, run through normal process.  
  
 # Nottin' on you.  
 body = None  
 content\_type = None  
  
 if not data and json is not None:  
 # urllib3 requires a bytes-like body. Python 2's json.dumps  
 # provides this natively, but Python 3 gives a Unicode string.  
 content\_type = "application/json"  
  
 try:  
 body = complexjson.dumps(json, allow\_nan=False)  
 except ValueError as ve:  
 raise InvalidJSONError(ve, request=self)  
  
 if not isinstance(body, bytes):  
 body = body.encode("utf-8")  
  
 is\_stream = all(  
 [  
 hasattr(data, "\_\_iter\_\_"),  
 not isinstance(data, (basestring, list, tuple, Mapping)),  
 ]  
 )  
  
 if is\_stream:  
 try:  
 length = super\_len(data)  
 except (TypeError, AttributeError, UnsupportedOperation):  
 length = None  
  
 body = data  
  
 if getattr(body, "tell", None) is not None:  
 # Record the current file position before reading.  
 # This will allow us to rewind a file in the event  
 # of a redirect.  
 try:  
 self.\_body\_position = body.tell()  
 except OSError:  
 # This differentiates from None, allowing us to catch  
 # a failed `tell()` later when trying to rewind the body  
 self.\_body\_position = object()  
  
 if files:  
 raise NotImplementedError(  
 "Streamed bodies and files are mutually exclusive."  
 )  
  
 if length:  
 self.headers["Content-Length"] = builtin\_str(length)  
 else:  
 self.headers["Transfer-Encoding"] = "chunked"  
 else:  
 # Multi-part file uploads.  
 if files:  
 (body, content\_type) = self.\_encode\_files(files, data)  
 else:  
 if data:  
 body = self.\_encode\_params(data)  
 if isinstance(data, basestring) or hasattr(data, "read"):  
 content\_type = None  
 else:  
 content\_type = "application/x-www-form-urlencoded"  
  
 self.prepare\_content\_length(body)  
  
 # Add content-type if it wasn't explicitly provided.  
 if content\_type and ("content-type" not in self.headers):  
 self.headers["Content-Type"] = content\_type  
  
 self.body = body  
  
 def prepare\_content\_length(self, body):  
 """Prepare Content-Length header based on request method and body"""  
 if body is not None:  
 length = super\_len(body)  
 if length:  
 # If length exists, set it. Otherwise, we fallback  
 # to Transfer-Encoding: chunked.  
 self.headers["Content-Length"] = builtin\_str(length)  
 elif (  
 self.method not in ("GET", "HEAD")  
 and self.headers.get("Content-Length") is None  
 ):  
 # Set Content-Length to 0 for methods that can have a body  
 # but don't provide one. (i.e. not GET or HEAD)  
 self.headers["Content-Length"] = "0"  
  
 def prepare\_auth(self, auth, url=""):  
 """Prepares the given HTTP auth data."""  
  
 # If no Auth is explicitly provided, extract it from the URL first.  
 if auth is None:  
 url\_auth = get\_auth\_from\_url(self.url)  
 auth = url\_auth if any(url\_auth) else None  
  
 if auth:  
 if isinstance(auth, tuple) and len(auth) == 2:  
 # special-case basic HTTP auth  
 auth = HTTPBasicAuth(\*auth)  
  
 # Allow auth to make its changes.  
 r = auth(self)  
  
 # Update self to reflect the auth changes.  
 self.\_\_dict\_\_.update(r.\_\_dict\_\_)  
  
 # Recompute Content-Length  
 self.prepare\_content\_length(self.body)  
  
 def prepare\_cookies(self, cookies):  
 """Prepares the given HTTP cookie data.  
  
 This function eventually generates a ``Cookie`` header from the  
 given cookies using cookielib. Due to cookielib's design, the header  
 will not be regenerated if it already exists, meaning this function  
 can only be called once for the life of the  
 :class:`PreparedRequest <PreparedRequest>` object. Any subsequent calls  
 to ``prepare\_cookies`` will have no actual effect, unless the "Cookie"  
 header is removed beforehand.  
 """  
 if isinstance(cookies, cookielib.CookieJar):  
 self.\_cookies = cookies  
 else:  
 self.\_cookies = cookiejar\_from\_dict(cookies)  
  
 cookie\_header = get\_cookie\_header(self.\_cookies, self)  
 if cookie\_header is not None:  
 self.headers["Cookie"] = cookie\_header  
  
 def prepare\_hooks(self, hooks):  
 """Prepares the given hooks."""  
 # hooks can be passed as None to the prepare method and to this  
 # method. To prevent iterating over None, simply use an empty list  
 # if hooks is False-y  
 hooks = hooks or []  
 for event in hooks:  
 self.register\_hook(event, hooks[event])  
  
  
class Response:  
 """The :class:`Response <Response>` object, which contains a  
 server's response to an HTTP request.  
 """  
  
 \_\_attrs\_\_ = [  
 "\_content",  
 "status\_code",  
 "headers",  
 "url",  
 "history",  
 "encoding",  
 "reason",  
 "cookies",  
 "elapsed",  
 "request",  
 ]  
  
 def \_\_init\_\_(self):  
 self.\_content = False  
 self.\_content\_consumed = False  
 self.\_next = None  
  
 #: Integer Code of responded HTTP Status, e.g. 404 or 200.  
 self.status\_code = None  
  
 #: Case-insensitive Dictionary of Response Headers.  
 #: For example, ``headers['content-encoding']`` will return the  
 #: value of a ``'Content-Encoding'`` response header.  
 self.headers = CaseInsensitiveDict()  
  
 #: File-like object representation of response (for advanced usage).  
 #: Use of ``raw`` requires that ``stream=True`` be set on the request.  
 #: This requirement does not apply for use internally to Requests.  
 self.raw = None  
  
 #: Final URL location of Response.  
 self.url = None  
  
 #: Encoding to decode with when accessing r.text.  
 self.encoding = None  
  
 #: A list of :class:`Response <Response>` objects from  
 #: the history of the Request. Any redirect responses will end  
 #: up here. The list is sorted from the oldest to the most recent request.  
 self.history = []  
  
 #: Textual reason of responded HTTP Status, e.g. "Not Found" or "OK".  
 self.reason = None  
  
 #: A CookieJar of Cookies the server sent back.  
 self.cookies = cookiejar\_from\_dict({})  
  
 #: The amount of time elapsed between sending the request  
 #: and the arrival of the response (as a timedelta).  
 #: This property specifically measures the time taken between sending  
 #: the first byte of the request and finishing parsing the headers. It  
 #: is therefore unaffected by consuming the response content or the  
 #: value of the ``stream`` keyword argument.  
 self.elapsed = datetime.timedelta(0)  
  
 #: The :class:`PreparedRequest <PreparedRequest>` object to which this  
 #: is a response.  
 self.request = None  
  
 def \_\_enter\_\_(self):  
 return self  
  
 def \_\_exit\_\_(self, \*args):  
 self.close()  
  
 def \_\_getstate\_\_(self):  
 # Consume everything; accessing the content attribute makes  
 # sure the content has been fully read.  
 if not self.\_content\_consumed:  
 self.content  
  
 return {attr: getattr(self, attr, None) for attr in self.\_\_attrs\_\_}  
  
 def \_\_setstate\_\_(self, state):  
 for name, value in state.items():  
 setattr(self, name, value)  
  
 # pickled objects do not have .raw  
 setattr(self, "\_content\_consumed", True)  
 setattr(self, "raw", None)  
  
 def \_\_repr\_\_(self):  
 return f"<Response [{self.status\_code}]>"  
  
 def \_\_bool\_\_(self):  
 """Returns True if :attr:`status\_code` is less than 400.  
  
 This attribute checks if the status code of the response is between  
 400 and 600 to see if there was a client error or a server error. If  
 the status code, is between 200 and 400, this will return True. This  
 is \*\*not\*\* a check to see if the response code is ``200 OK``.  
 """  
 return self.ok  
  
 def \_\_nonzero\_\_(self):  
 """Returns True if :attr:`status\_code` is less than 400.  
  
 This attribute checks if the status code of the response is between  
 400 and 600 to see if there was a client error or a server error. If  
 the status code, is between 200 and 400, this will return True. This  
 is \*\*not\*\* a check to see if the response code is ``200 OK``.  
 """  
 return self.ok  
  
 def \_\_iter\_\_(self):  
 """Allows you to use a response as an iterator."""  
 return self.iter\_content(128)  
  
 @property  
 def ok(self):  
 """Returns True if :attr:`status\_code` is less than 400, False if not.  
  
 This attribute checks if the status code of the response is between  
 400 and 600 to see if there was a client error or a server error. If  
 the status code is between 200 and 400, this will return True. This  
 is \*\*not\*\* a check to see if the response code is ``200 OK``.  
 """  
 try:  
 self.raise\_for\_status()  
 except HTTPError:  
 return False  
 return True  
  
 @property  
 def is\_redirect(self):  
 """True if this Response is a well-formed HTTP redirect that could have  
 been processed automatically (by :meth:`Session.resolve\_redirects`).  
 """  
 return "location" in self.headers and self.status\_code in REDIRECT\_STATI  
  
 @property  
 def is\_permanent\_redirect(self):  
 """True if this Response one of the permanent versions of redirect."""  
 return "location" in self.headers and self.status\_code in (  
 codes.moved\_permanently,  
 codes.permanent\_redirect,  
 )  
  
 @property  
 def next(self):  
 """Returns a PreparedRequest for the next request in a redirect chain, if there is one."""  
 return self.\_next  
  
 @property  
 def apparent\_encoding(self):  
 """The apparent encoding, provided by the charset\_normalizer or chardet libraries."""  
 return chardet.detect(self.content)["encoding"]  
  
 def iter\_content(self, chunk\_size=1, decode\_unicode=False):  
 """Iterates over the response data. When stream=True is set on the  
 request, this avoids reading the content at once into memory for  
 large responses. The chunk size is the number of bytes it should  
 read into memory. This is not necessarily the length of each item  
 returned as decoding can take place.  
  
 chunk\_size must be of type int or None. A value of None will  
 function differently depending on the value of `stream`.  
 stream=True will read data as it arrives in whatever size the  
 chunks are received. If stream=False, data is returned as  
 a single chunk.  
  
 If decode\_unicode is True, content will be decoded using the best  
 available encoding based on the response.  
 """  
  
 def generate():  
 # Special case for urllib3.  
 if hasattr(self.raw, "stream"):  
 try:  
 yield from self.raw.stream(chunk\_size, decode\_content=True)  
 except ProtocolError as e:  
 raise ChunkedEncodingError(e)  
 except DecodeError as e:  
 raise ContentDecodingError(e)  
 except ReadTimeoutError as e:  
 raise ConnectionError(e)  
 except SSLError as e:  
 raise RequestsSSLError(e)  
 else:  
 # Standard file-like object.  
 while True:  
 chunk = self.raw.read(chunk\_size)  
 if not chunk:  
 break  
 yield chunk  
  
 self.\_content\_consumed = True  
  
 if self.\_content\_consumed and isinstance(self.\_content, bool):  
 raise StreamConsumedError()  
 elif chunk\_size is not None and not isinstance(chunk\_size, int):  
 raise TypeError(  
 f"chunk\_size must be an int, it is instead a {type(chunk\_size)}."  
 )  
 # simulate reading small chunks of the content  
 reused\_chunks = iter\_slices(self.\_content, chunk\_size)  
  
 stream\_chunks = generate()  
  
 chunks = reused\_chunks if self.\_content\_consumed else stream\_chunks  
  
 if decode\_unicode:  
 chunks = stream\_decode\_response\_unicode(chunks, self)  
  
 return chunks  
  
 def iter\_lines(  
 self, chunk\_size=ITER\_CHUNK\_SIZE, decode\_unicode=False, delimiter=None  
 ):  
 """Iterates over the response data, one line at a time. When  
 stream=True is set on the request, this avoids reading the  
 content at once into memory for large responses.  
  
 .. note:: This method is not reentrant safe.  
 """  
  
 pending = None  
  
 for chunk in self.iter\_content(  
 chunk\_size=chunk\_size, decode\_unicode=decode\_unicode  
 ):  
  
 if pending is not None:  
 chunk = pending + chunk  
  
 if delimiter:  
 lines = chunk.split(delimiter)  
 else:  
 lines = chunk.splitlines()  
  
 if lines and lines[-1] and chunk and lines[-1][-1] == chunk[-1]:  
 pending = lines.pop()  
 else:  
 pending = None  
  
 yield from lines  
  
 if pending is not None:  
 yield pending  
  
 @property  
 def content(self):  
 """Content of the response, in bytes."""  
  
 if self.\_content is False:  
 # Read the contents.  
 if self.\_content\_consumed:  
 raise RuntimeError("The content for this response was already consumed")  
  
 if self.status\_code == 0 or self.raw is None:  
 self.\_content = None  
 else:  
 self.\_content = b"".join(self.iter\_content(CONTENT\_CHUNK\_SIZE)) or b""  
  
 self.\_content\_consumed = True  
 # don't need to release the connection; that's been handled by urllib3  
 # since we exhausted the data.  
 return self.\_content  
  
 @property  
 def text(self):  
 """Content of the response, in unicode.  
  
 If Response.encoding is None, encoding will be guessed using  
 ``charset\_normalizer`` or ``chardet``.  
  
 The encoding of the response content is determined based solely on HTTP  
 headers, following RFC 2616 to the letter. If you can take advantage of  
 non-HTTP knowledge to make a better guess at the encoding, you should  
 set ``r.encoding`` appropriately before accessing this property.  
 """  
  
 # Try charset from content-type  
 content = None  
 encoding = self.encoding  
  
 if not self.content:  
 return ""  
  
 # Fallback to auto-detected encoding.  
 if self.encoding is None:  
 encoding = self.apparent\_encoding  
  
 # Decode unicode from given encoding.  
 try:  
 content = str(self.content, encoding, errors="replace")  
 except (LookupError, TypeError):  
 # A LookupError is raised if the encoding was not found which could  
 # indicate a misspelling or similar mistake.  
 #  
 # A TypeError can be raised if encoding is None  
 #  
 # So we try blindly encoding.  
 content = str(self.content, errors="replace")  
  
 return content  
  
 def json(self, \*\*kwargs):  
 r"""Returns the json-encoded content of a response, if any.  
  
 :param \\*\\*kwargs: Optional arguments that ``json.loads`` takes.  
 :raises requests.exceptions.JSONDecodeError: If the response body does not  
 contain valid json.  
 """  
  
 if not self.encoding and self.content and len(self.content) > 3:  
 # No encoding set. JSON RFC 4627 section 3 states we should expect  
 # UTF-8, -16 or -32. Detect which one to use; If the detection or  
 # decoding fails, fall back to `self.text` (using charset\_normalizer to make  
 # a best guess).  
 encoding = guess\_json\_utf(self.content)  
 if encoding is not None:  
 try:  
 return complexjson.loads(self.content.decode(encoding), \*\*kwargs)  
 except UnicodeDecodeError:  
 # Wrong UTF codec detected; usually because it's not UTF-8  
 # but some other 8-bit codec. This is an RFC violation,  
 # and the server didn't bother to tell us what codec \*was\*  
 # used.  
 pass  
 except JSONDecodeError as e:  
 raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)  
  
 try:  
 return complexjson.loads(self.text, \*\*kwargs)  
 except JSONDecodeError as e:  
 # Catch JSON-related errors and raise as requests.JSONDecodeError  
 # This aliases json.JSONDecodeError and simplejson.JSONDecodeError  
 raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)  
  
 @property  
 def links(self):  
 """Returns the parsed header links of the response, if any."""  
  
 header = self.headers.get("link")  
  
 resolved\_links = {}  
  
 if header:  
 links = parse\_header\_links(header)  
  
 for link in links:  
 key = link.get("rel") or link.get("url")  
 resolved\_links[key] = link  
  
 return resolved\_links  
  
 def raise\_for\_status(self):  
 """Raises :class:`HTTPError`, if one occurred."""  
  
 http\_error\_msg = ""  
 if isinstance(self.reason, bytes):  
 # We attempt to decode utf-8 first because some servers  
 # choose to localize their reason strings. If the string  
 # isn't utf-8, we fall back to iso-8859-1 for all other  
 # encodings. (See PR #3538)  
 try:  
 reason = self.reason.decode("utf-8")  
 except UnicodeDecodeError:  
 reason = self.reason.decode("iso-8859-1")  
 else:  
 reason = self.reason  
  
 if 400 <= self.status\_code < 500:  
 http\_error\_msg = (  
 f"{self.status\_code} Client Error: {reason} for url: {self.url}"  
 )  
  
 elif 500 <= self.status\_code < 600:  
 http\_error\_msg = (  
 f"{self.status\_code} Server Error: {reason} for url: {self.url}"  
 )  
  
 if http\_error\_msg:  
 raise HTTPError(http\_error\_msg, response=self)  
  
 def close(self):  
 """Releases the connection back to the pool. Once this method has been  
 called the underlying ``raw`` object must not be accessed again.  
  
 \*Note: Should not normally need to be called explicitly.\*  
 """  
 if not self.\_content\_consumed:  
 self.raw.close()  
  
 release\_conn = getattr(self.raw, "release\_conn", None)  
 if release\_conn is not None:  
 release\_conn()

## File: venv/lib/python3.12/site-packages/requests/models.py

"""  
requests.models  
~~~~~~~~~~~~~~~  
  
This module contains the primary objects that power Requests.  
"""  
  
import datetime  
  
# Import encoding now, to avoid implicit import later.  
# Implicit import within threads may cause LookupError when standard library is in a ZIP,  
# such as in Embedded Python. See https://github.com/psf/requests/issues/3578.  
import encodings.idna # noqa: F401  
from io import UnsupportedOperation  
  
from urllib3.exceptions import (  
 DecodeError,  
 LocationParseError,  
 ProtocolError,  
 ReadTimeoutError,  
 SSLError,  
)  
from urllib3.fields import RequestField  
from urllib3.filepost import encode\_multipart\_formdata  
from urllib3.util import parse\_url  
  
from .\_internal\_utils import to\_native\_string, unicode\_is\_ascii  
from .auth import HTTPBasicAuth  
from .compat import (  
 Callable,  
 JSONDecodeError,  
 Mapping,  
 basestring,  
 builtin\_str,  
 chardet,  
 cookielib,  
)  
from .compat import json as complexjson  
from .compat import urlencode, urlsplit, urlunparse  
from .cookies import \_copy\_cookie\_jar, cookiejar\_from\_dict, get\_cookie\_header  
from .exceptions import (  
 ChunkedEncodingError,  
 ConnectionError,  
 ContentDecodingError,  
 HTTPError,  
 InvalidJSONError,  
 InvalidURL,  
)  
from .exceptions import JSONDecodeError as RequestsJSONDecodeError  
from .exceptions import MissingSchema  
from .exceptions import SSLError as RequestsSSLError  
from .exceptions import StreamConsumedError  
from .hooks import default\_hooks  
from .status\_codes import codes  
from .structures import CaseInsensitiveDict  
from .utils import (  
 check\_header\_validity,  
 get\_auth\_from\_url,  
 guess\_filename,  
 guess\_json\_utf,  
 iter\_slices,  
 parse\_header\_links,  
 requote\_uri,  
 stream\_decode\_response\_unicode,  
 super\_len,  
 to\_key\_val\_list,  
)  
  
#: The set of HTTP status codes that indicate an automatically  
#: processable redirect.  
REDIRECT\_STATI = (  
 codes.moved, # 301  
 codes.found, # 302  
 codes.other, # 303  
 codes.temporary\_redirect, # 307  
 codes.permanent\_redirect, # 308  
)  
  
DEFAULT\_REDIRECT\_LIMIT = 30  
CONTENT\_CHUNK\_SIZE = 10 \* 1024  
ITER\_CHUNK\_SIZE = 512  
  
  
class RequestEncodingMixin:  
 @property  
 def path\_url(self):  
 """Build the path URL to use."""  
  
 url = []  
  
 p = urlsplit(self.url)  
  
 path = p.path  
 if not path:  
 path = "/"  
  
 url.append(path)  
  
 query = p.query  
 if query:  
 url.append("?")  
 url.append(query)  
  
 return "".join(url)  
  
 @staticmethod  
 def \_encode\_params(data):  
 """Encode parameters in a piece of data.  
  
 Will successfully encode parameters when passed as a dict or a list of  
 2-tuples. Order is retained if data is a list of 2-tuples but arbitrary  
 if parameters are supplied as a dict.  
 """  
  
 if isinstance(data, (str, bytes)):  
 return data  
 elif hasattr(data, "read"):  
 return data  
 elif hasattr(data, "\_\_iter\_\_"):  
 result = []  
 for k, vs in to\_key\_val\_list(data):  
 if isinstance(vs, basestring) or not hasattr(vs, "\_\_iter\_\_"):  
 vs = [vs]  
 for v in vs:  
 if v is not None:  
 result.append(  
 (  
 k.encode("utf-8") if isinstance(k, str) else k,  
 v.encode("utf-8") if isinstance(v, str) else v,  
 )  
 )  
 return urlencode(result, doseq=True)  
 else:  
 return data  
  
 @staticmethod  
 def \_encode\_files(files, data):  
 """Build the body for a multipart/form-data request.  
  
 Will successfully encode files when passed as a dict or a list of  
 tuples. Order is retained if data is a list of tuples but arbitrary  
 if parameters are supplied as a dict.  
 The tuples may be 2-tuples (filename, fileobj), 3-tuples (filename, fileobj, contentype)  
 or 4-tuples (filename, fileobj, contentype, custom\_headers).  
 """  
 if not files:  
 raise ValueError("Files must be provided.")  
 elif isinstance(data, basestring):  
 raise ValueError("Data must not be a string.")  
  
 new\_fields = []  
 fields = to\_key\_val\_list(data or {})  
 files = to\_key\_val\_list(files or {})  
  
 for field, val in fields:  
 if isinstance(val, basestring) or not hasattr(val, "\_\_iter\_\_"):  
 val = [val]  
 for v in val:  
 if v is not None:  
 # Don't call str() on bytestrings: in Py3 it all goes wrong.  
 if not isinstance(v, bytes):  
 v = str(v)  
  
 new\_fields.append(  
 (  
 field.decode("utf-8")  
 if isinstance(field, bytes)  
 else field,  
 v.encode("utf-8") if isinstance(v, str) else v,  
 )  
 )  
  
 for (k, v) in files:  
 # support for explicit filename  
 ft = None  
 fh = None  
 if isinstance(v, (tuple, list)):  
 if len(v) == 2:  
 fn, fp = v  
 elif len(v) == 3:  
 fn, fp, ft = v  
 else:  
 fn, fp, ft, fh = v  
 else:  
 fn = guess\_filename(v) or k  
 fp = v  
  
 if isinstance(fp, (str, bytes, bytearray)):  
 fdata = fp  
 elif hasattr(fp, "read"):  
 fdata = fp.read()  
 elif fp is None:  
 continue  
 else:  
 fdata = fp  
  
 rf = RequestField(name=k, data=fdata, filename=fn, headers=fh)  
 rf.make\_multipart(content\_type=ft)  
 new\_fields.append(rf)  
  
 body, content\_type = encode\_multipart\_formdata(new\_fields)  
  
 return body, content\_type  
  
  
class RequestHooksMixin:  
 def register\_hook(self, event, hook):  
 """Properly register a hook."""  
  
 if event not in self.hooks:  
 raise ValueError(f'Unsupported event specified, with event name "{event}"')  
  
 if isinstance(hook, Callable):  
 self.hooks[event].append(hook)  
 elif hasattr(hook, "\_\_iter\_\_"):  
 self.hooks[event].extend(h for h in hook if isinstance(h, Callable))  
  
 def deregister\_hook(self, event, hook):  
 """Deregister a previously registered hook.  
 Returns True if the hook existed, False if not.  
 """  
  
 try:  
 self.hooks[event].remove(hook)  
 return True  
 except ValueError:  
 return False  
  
  
class Request(RequestHooksMixin):  
 """A user-created :class:`Request <Request>` object.  
  
 Used to prepare a :class:`PreparedRequest <PreparedRequest>`, which is sent to the server.  
  
 :param method: HTTP method to use.  
 :param url: URL to send.  
 :param headers: dictionary of headers to send.  
 :param files: dictionary of {filename: fileobject} files to multipart upload.  
 :param data: the body to attach to the request. If a dictionary or  
 list of tuples ``[(key, value)]`` is provided, form-encoding will  
 take place.  
 :param json: json for the body to attach to the request (if files or data is not specified).  
 :param params: URL parameters to append to the URL. If a dictionary or  
 list of tuples ``[(key, value)]`` is provided, form-encoding will  
 take place.  
 :param auth: Auth handler or (user, pass) tuple.  
 :param cookies: dictionary or CookieJar of cookies to attach to this request.  
 :param hooks: dictionary of callback hooks, for internal usage.  
  
 Usage::  
  
 >>> import requests  
 >>> req = requests.Request('GET', 'https://httpbin.org/get')  
 >>> req.prepare()  
 <PreparedRequest [GET]>  
 """  
  
 def \_\_init\_\_(  
 self,  
 method=None,  
 url=None,  
 headers=None,  
 files=None,  
 data=None,  
 params=None,  
 auth=None,  
 cookies=None,  
 hooks=None,  
 json=None,  
 ):  
  
 # Default empty dicts for dict params.  
 data = [] if data is None else data  
 files = [] if files is None else files  
 headers = {} if headers is None else headers  
 params = {} if params is None else params  
 hooks = {} if hooks is None else hooks  
  
 self.hooks = default\_hooks()  
 for (k, v) in list(hooks.items()):  
 self.register\_hook(event=k, hook=v)  
  
 self.method = method  
 self.url = url  
 self.headers = headers  
 self.files = files  
 self.data = data  
 self.json = json  
 self.params = params  
 self.auth = auth  
 self.cookies = cookies  
  
 def \_\_repr\_\_(self):  
 return f"<Request [{self.method}]>"  
  
 def prepare(self):  
 """Constructs a :class:`PreparedRequest <PreparedRequest>` for transmission and returns it."""  
 p = PreparedRequest()  
 p.prepare(  
 method=self.method,  
 url=self.url,  
 headers=self.headers,  
 files=self.files,  
 data=self.data,  
 json=self.json,  
 params=self.params,  
 auth=self.auth,  
 cookies=self.cookies,  
 hooks=self.hooks,  
 )  
 return p  
  
  
class PreparedRequest(RequestEncodingMixin, RequestHooksMixin):  
 """The fully mutable :class:`PreparedRequest <PreparedRequest>` object,  
 containing the exact bytes that will be sent to the server.  
  
 Instances are generated from a :class:`Request <Request>` object, and  
 should not be instantiated manually; doing so may produce undesirable  
 effects.  
  
 Usage::  
  
 >>> import requests  
 >>> req = requests.Request('GET', 'https://httpbin.org/get')  
 >>> r = req.prepare()  
 >>> r  
 <PreparedRequest [GET]>  
  
 >>> s = requests.Session()  
 >>> s.send(r)  
 <Response [200]>  
 """  
  
 def \_\_init\_\_(self):  
 #: HTTP verb to send to the server.  
 self.method = None  
 #: HTTP URL to send the request to.  
 self.url = None  
 #: dictionary of HTTP headers.  
 self.headers = None  
 # The `CookieJar` used to create the Cookie header will be stored here  
 # after prepare\_cookies is called  
 self.\_cookies = None  
 #: request body to send to the server.  
 self.body = None  
 #: dictionary of callback hooks, for internal usage.  
 self.hooks = default\_hooks()  
 #: integer denoting starting position of a readable file-like body.  
 self.\_body\_position = None  
  
 def prepare(  
 self,  
 method=None,  
 url=None,  
 headers=None,  
 files=None,  
 data=None,  
 params=None,  
 auth=None,  
 cookies=None,  
 hooks=None,  
 json=None,  
 ):  
 """Prepares the entire request with the given parameters."""  
  
 self.prepare\_method(method)  
 self.prepare\_url(url, params)  
 self.prepare\_headers(headers)  
 self.prepare\_cookies(cookies)  
 self.prepare\_body(data, files, json)  
 self.prepare\_auth(auth, url)  
  
 # Note that prepare\_auth must be last to enable authentication schemes  
 # such as OAuth to work on a fully prepared request.  
  
 # This MUST go after prepare\_auth. Authenticators could add a hook  
 self.prepare\_hooks(hooks)  
  
 def \_\_repr\_\_(self):  
 return f"<PreparedRequest [{self.method}]>"  
  
 def copy(self):  
 p = PreparedRequest()  
 p.method = self.method  
 p.url = self.url  
 p.headers = self.headers.copy() if self.headers is not None else None  
 p.\_cookies = \_copy\_cookie\_jar(self.\_cookies)  
 p.body = self.body  
 p.hooks = self.hooks  
 p.\_body\_position = self.\_body\_position  
 return p  
  
 def prepare\_method(self, method):  
 """Prepares the given HTTP method."""  
 self.method = method  
 if self.method is not None:  
 self.method = to\_native\_string(self.method.upper())  
  
 @staticmethod  
 def \_get\_idna\_encoded\_host(host):  
 import idna  
  
 try:  
 host = idna.encode(host, uts46=True).decode("utf-8")  
 except idna.IDNAError:  
 raise UnicodeError  
 return host  
  
 def prepare\_url(self, url, params):  
 """Prepares the given HTTP URL."""  
 #: Accept objects that have string representations.  
 #: We're unable to blindly call unicode/str functions  
 #: as this will include the bytestring indicator (b'')  
 #: on python 3.x.  
 #: https://github.com/psf/requests/pull/2238  
 if isinstance(url, bytes):  
 url = url.decode("utf8")  
 else:  
 url = str(url)  
  
 # Remove leading whitespaces from url  
 url = url.lstrip()  
  
 # Don't do any URL preparation for non-HTTP schemes like `mailto`,  
 # `data` etc to work around exceptions from `url\_parse`, which  
 # handles RFC 3986 only.  
 if ":" in url and not url.lower().startswith("http"):  
 self.url = url  
 return  
  
 # Support for unicode domain names and paths.  
 try:  
 scheme, auth, host, port, path, query, fragment = parse\_url(url)  
 except LocationParseError as e:  
 raise InvalidURL(\*e.args)  
  
 if not scheme:  
 raise MissingSchema(  
 f"Invalid URL {url!r}: No scheme supplied. "  
 f"Perhaps you meant https://{url}?"  
 )  
  
 if not host:  
 raise InvalidURL(f"Invalid URL {url!r}: No host supplied")  
  
 # In general, we want to try IDNA encoding the hostname if the string contains  
 # non-ASCII characters. This allows users to automatically get the correct IDNA  
 # behaviour. For strings containing only ASCII characters, we need to also verify  
 # it doesn't start with a wildcard (\*), before allowing the unencoded hostname.  
 if not unicode\_is\_ascii(host):  
 try:  
 host = self.\_get\_idna\_encoded\_host(host)  
 except UnicodeError:  
 raise InvalidURL("URL has an invalid label.")  
 elif host.startswith(("\*", ".")):  
 raise InvalidURL("URL has an invalid label.")  
  
 # Carefully reconstruct the network location  
 netloc = auth or ""  
 if netloc:  
 netloc += "@"  
 netloc += host  
 if port:  
 netloc += f":{port}"  
  
 # Bare domains aren't valid URLs.  
 if not path:  
 path = "/"  
  
 if isinstance(params, (str, bytes)):  
 params = to\_native\_string(params)  
  
 enc\_params = self.\_encode\_params(params)  
 if enc\_params:  
 if query:  
 query = f"{query}&{enc\_params}"  
 else:  
 query = enc\_params  
  
 url = requote\_uri(urlunparse([scheme, netloc, path, None, query, fragment]))  
 self.url = url  
  
 def prepare\_headers(self, headers):  
 """Prepares the given HTTP headers."""  
  
 self.headers = CaseInsensitiveDict()  
 if headers:  
 for header in headers.items():  
 # Raise exception on invalid header value.  
 check\_header\_validity(header)  
 name, value = header  
 self.headers[to\_native\_string(name)] = value  
  
 def prepare\_body(self, data, files, json=None):  
 """Prepares the given HTTP body data."""  
  
 # Check if file, fo, generator, iterator.  
 # If not, run through normal process.  
  
 # Nottin' on you.  
 body = None  
 content\_type = None  
  
 if not data and json is not None:  
 # urllib3 requires a bytes-like body. Python 2's json.dumps  
 # provides this natively, but Python 3 gives a Unicode string.  
 content\_type = "application/json"  
  
 try:  
 body = complexjson.dumps(json, allow\_nan=False)  
 except ValueError as ve:  
 raise InvalidJSONError(ve, request=self)  
  
 if not isinstance(body, bytes):  
 body = body.encode("utf-8")  
  
 is\_stream = all(  
 [  
 hasattr(data, "\_\_iter\_\_"),  
 not isinstance(data, (basestring, list, tuple, Mapping)),  
 ]  
 )  
  
 if is\_stream:  
 try:  
 length = super\_len(data)  
 except (TypeError, AttributeError, UnsupportedOperation):  
 length = None  
  
 body = data  
  
 if getattr(body, "tell", None) is not None:  
 # Record the current file position before reading.  
 # This will allow us to rewind a file in the event  
 # of a redirect.  
 try:  
 self.\_body\_position = body.tell()  
 except OSError:  
 # This differentiates from None, allowing us to catch  
 # a failed `tell()` later when trying to rewind the body  
 self.\_body\_position = object()  
  
 if files:  
 raise NotImplementedError(  
 "Streamed bodies and files are mutually exclusive."  
 )  
  
 if length:  
 self.headers["Content-Length"] = builtin\_str(length)  
 else:  
 self.headers["Transfer-Encoding"] = "chunked"  
 else:  
 # Multi-part file uploads.  
 if files:  
 (body, content\_type) = self.\_encode\_files(files, data)  
 else:  
 if data:  
 body = self.\_encode\_params(data)  
 if isinstance(data, basestring) or hasattr(data, "read"):  
 content\_type = None  
 else:  
 content\_type = "application/x-www-form-urlencoded"  
  
 self.prepare\_content\_length(body)  
  
 # Add content-type if it wasn't explicitly provided.  
 if content\_type and ("content-type" not in self.headers):  
 self.headers["Content-Type"] = content\_type  
  
 self.body = body  
  
 def prepare\_content\_length(self, body):  
 """Prepare Content-Length header based on request method and body"""  
 if body is not None:  
 length = super\_len(body)  
 if length:  
 # If length exists, set it. Otherwise, we fallback  
 # to Transfer-Encoding: chunked.  
 self.headers["Content-Length"] = builtin\_str(length)  
 elif (  
 self.method not in ("GET", "HEAD")  
 and self.headers.get("Content-Length") is None  
 ):  
 # Set Content-Length to 0 for methods that can have a body  
 # but don't provide one. (i.e. not GET or HEAD)  
 self.headers["Content-Length"] = "0"  
  
 def prepare\_auth(self, auth, url=""):  
 """Prepares the given HTTP auth data."""  
  
 # If no Auth is explicitly provided, extract it from the URL first.  
 if auth is None:  
 url\_auth = get\_auth\_from\_url(self.url)  
 auth = url\_auth if any(url\_auth) else None  
  
 if auth:  
 if isinstance(auth, tuple) and len(auth) == 2:  
 # special-case basic HTTP auth  
 auth = HTTPBasicAuth(\*auth)  
  
 # Allow auth to make its changes.  
 r = auth(self)  
  
 # Update self to reflect the auth changes.  
 self.\_\_dict\_\_.update(r.\_\_dict\_\_)  
  
 # Recompute Content-Length  
 self.prepare\_content\_length(self.body)  
  
 def prepare\_cookies(self, cookies):  
 """Prepares the given HTTP cookie data.  
  
 This function eventually generates a ``Cookie`` header from the  
 given cookies using cookielib. Due to cookielib's design, the header  
 will not be regenerated if it already exists, meaning this function  
 can only be called once for the life of the  
 :class:`PreparedRequest <PreparedRequest>` object. Any subsequent calls  
 to ``prepare\_cookies`` will have no actual effect, unless the "Cookie"  
 header is removed beforehand.  
 """  
 if isinstance(cookies, cookielib.CookieJar):  
 self.\_cookies = cookies  
 else:  
 self.\_cookies = cookiejar\_from\_dict(cookies)  
  
 cookie\_header = get\_cookie\_header(self.\_cookies, self)  
 if cookie\_header is not None:  
 self.headers["Cookie"] = cookie\_header  
  
 def prepare\_hooks(self, hooks):  
 """Prepares the given hooks."""  
 # hooks can be passed as None to the prepare method and to this  
 # method. To prevent iterating over None, simply use an empty list  
 # if hooks is False-y  
 hooks = hooks or []  
 for event in hooks:  
 self.register\_hook(event, hooks[event])  
  
  
class Response:  
 """The :class:`Response <Response>` object, which contains a  
 server's response to an HTTP request.  
 """  
  
 \_\_attrs\_\_ = [  
 "\_content",  
 "status\_code",  
 "headers",  
 "url",  
 "history",  
 "encoding",  
 "reason",  
 "cookies",  
 "elapsed",  
 "request",  
 ]  
  
 def \_\_init\_\_(self):  
 self.\_content = False  
 self.\_content\_consumed = False  
 self.\_next = None  
  
 #: Integer Code of responded HTTP Status, e.g. 404 or 200.  
 self.status\_code = None  
  
 #: Case-insensitive Dictionary of Response Headers.  
 #: For example, ``headers['content-encoding']`` will return the  
 #: value of a ``'Content-Encoding'`` response header.  
 self.headers = CaseInsensitiveDict()  
  
 #: File-like object representation of response (for advanced usage).  
 #: Use of ``raw`` requires that ``stream=True`` be set on the request.  
 #: This requirement does not apply for use internally to Requests.  
 self.raw = None  
  
 #: Final URL location of Response.  
 self.url = None  
  
 #: Encoding to decode with when accessing r.text.  
 self.encoding = None  
  
 #: A list of :class:`Response <Response>` objects from  
 #: the history of the Request. Any redirect responses will end  
 #: up here. The list is sorted from the oldest to the most recent request.  
 self.history = []  
  
 #: Textual reason of responded HTTP Status, e.g. "Not Found" or "OK".  
 self.reason = None  
  
 #: A CookieJar of Cookies the server sent back.  
 self.cookies = cookiejar\_from\_dict({})  
  
 #: The amount of time elapsed between sending the request  
 #: and the arrival of the response (as a timedelta).  
 #: This property specifically measures the time taken between sending  
 #: the first byte of the request and finishing parsing the headers. It  
 #: is therefore unaffected by consuming the response content or the  
 #: value of the ``stream`` keyword argument.  
 self.elapsed = datetime.timedelta(0)  
  
 #: The :class:`PreparedRequest <PreparedRequest>` object to which this  
 #: is a response.  
 self.request = None  
  
 def \_\_enter\_\_(self):  
 return self  
  
 def \_\_exit\_\_(self, \*args):  
 self.close()  
  
 def \_\_getstate\_\_(self):  
 # Consume everything; accessing the content attribute makes  
 # sure the content has been fully read.  
 if not self.\_content\_consumed:  
 self.content  
  
 return {attr: getattr(self, attr, None) for attr in self.\_\_attrs\_\_}  
  
 def \_\_setstate\_\_(self, state):  
 for name, value in state.items():  
 setattr(self, name, value)  
  
 # pickled objects do not have .raw  
 setattr(self, "\_content\_consumed", True)  
 setattr(self, "raw", None)  
  
 def \_\_repr\_\_(self):  
 return f"<Response [{self.status\_code}]>"  
  
 def \_\_bool\_\_(self):  
 """Returns True if :attr:`status\_code` is less than 400.  
  
 This attribute checks if the status code of the response is between  
 400 and 600 to see if there was a client error or a server error. If  
 the status code, is between 200 and 400, this will return True. This  
 is \*\*not\*\* a check to see if the response code is ``200 OK``.  
 """  
 return self.ok  
  
 def \_\_nonzero\_\_(self):  
 """Returns True if :attr:`status\_code` is less than 400.  
  
 This attribute checks if the status code of the response is between  
 400 and 600 to see if there was a client error or a server error. If  
 the status code, is between 200 and 400, this will return True. This  
 is \*\*not\*\* a check to see if the response code is ``200 OK``.  
 """  
 return self.ok  
  
 def \_\_iter\_\_(self):  
 """Allows you to use a response as an iterator."""  
 return self.iter\_content(128)  
  
 @property  
 def ok(self):  
 """Returns True if :attr:`status\_code` is less than 400, False if not.  
  
 This attribute checks if the status code of the response is between  
 400 and 600 to see if there was a client error or a server error. If  
 the status code is between 200 and 400, this will return True. This  
 is \*\*not\*\* a check to see if the response code is ``200 OK``.  
 """  
 try:  
 self.raise\_for\_status()  
 except HTTPError:  
 return False  
 return True  
  
 @property  
 def is\_redirect(self):  
 """True if this Response is a well-formed HTTP redirect that could have  
 been processed automatically (by :meth:`Session.resolve\_redirects`).  
 """  
 return "location" in self.headers and self.status\_code in REDIRECT\_STATI  
  
 @property  
 def is\_permanent\_redirect(self):  
 """True if this Response one of the permanent versions of redirect."""  
 return "location" in self.headers and self.status\_code in (  
 codes.moved\_permanently,  
 codes.permanent\_redirect,  
 )  
  
 @property  
 def next(self):  
 """Returns a PreparedRequest for the next request in a redirect chain, if there is one."""  
 return self.\_next  
  
 @property  
 def apparent\_encoding(self):  
 """The apparent encoding, provided by the charset\_normalizer or chardet libraries."""  
 return chardet.detect(self.content)["encoding"]  
  
 def iter\_content(self, chunk\_size=1, decode\_unicode=False):  
 """Iterates over the response data. When stream=True is set on the  
 request, this avoids reading the content at once into memory for  
 large responses. The chunk size is the number of bytes it should  
 read into memory. This is not necessarily the length of each item  
 returned as decoding can take place.  
  
 chunk\_size must be of type int or None. A value of None will  
 function differently depending on the value of `stream`.  
 stream=True will read data as it arrives in whatever size the  
 chunks are received. If stream=False, data is returned as  
 a single chunk.  
  
 If decode\_unicode is True, content will be decoded using the best  
 available encoding based on the response.  
 """  
  
 def generate():  
 # Special case for urllib3.  
 if hasattr(self.raw, "stream"):  
 try:  
 yield from self.raw.stream(chunk\_size, decode\_content=True)  
 except ProtocolError as e:  
 raise ChunkedEncodingError(e)  
 except DecodeError as e:  
 raise ContentDecodingError(e)  
 except ReadTimeoutError as e:  
 raise ConnectionError(e)  
 except SSLError as e:  
 raise RequestsSSLError(e)  
 else:  
 # Standard file-like object.  
 while True:  
 chunk = self.raw.read(chunk\_size)  
 if not chunk:  
 break  
 yield chunk  
  
 self.\_content\_consumed = True  
  
 if self.\_content\_consumed and isinstance(self.\_content, bool):  
 raise StreamConsumedError()  
 elif chunk\_size is not None and not isinstance(chunk\_size, int):  
 raise TypeError(  
 f"chunk\_size must be an int, it is instead a {type(chunk\_size)}."  
 )  
 # simulate reading small chunks of the content  
 reused\_chunks = iter\_slices(self.\_content, chunk\_size)  
  
 stream\_chunks = generate()  
  
 chunks = reused\_chunks if self.\_content\_consumed else stream\_chunks  
  
 if decode\_unicode:  
 chunks = stream\_decode\_response\_unicode(chunks, self)  
  
 return chunks  
  
 def iter\_lines(  
 self, chunk\_size=ITER\_CHUNK\_SIZE, decode\_unicode=False, delimiter=None  
 ):  
 """Iterates over the response data, one line at a time. When  
 stream=True is set on the request, this avoids reading the  
 content at once into memory for large responses.  
  
 .. note:: This method is not reentrant safe.  
 """  
  
 pending = None  
  
 for chunk in self.iter\_content(  
 chunk\_size=chunk\_size, decode\_unicode=decode\_unicode  
 ):  
  
 if pending is not None:  
 chunk = pending + chunk  
  
 if delimiter:  
 lines = chunk.split(delimiter)  
 else:  
 lines = chunk.splitlines()  
  
 if lines and lines[-1] and chunk and lines[-1][-1] == chunk[-1]:  
 pending = lines.pop()  
 else:  
 pending = None  
  
 yield from lines  
  
 if pending is not None:  
 yield pending  
  
 @property  
 def content(self):  
 """Content of the response, in bytes."""  
  
 if self.\_content is False:  
 # Read the contents.  
 if self.\_content\_consumed:  
 raise RuntimeError("The content for this response was already consumed")  
  
 if self.status\_code == 0 or self.raw is None:  
 self.\_content = None  
 else:  
 self.\_content = b"".join(self.iter\_content(CONTENT\_CHUNK\_SIZE)) or b""  
  
 self.\_content\_consumed = True  
 # don't need to release the connection; that's been handled by urllib3  
 # since we exhausted the data.  
 return self.\_content  
  
 @property  
 def text(self):  
 """Content of the response, in unicode.  
  
 If Response.encoding is None, encoding will be guessed using  
 ``charset\_normalizer`` or ``chardet``.  
  
 The encoding of the response content is determined based solely on HTTP  
 headers, following RFC 2616 to the letter. If you can take advantage of  
 non-HTTP knowledge to make a better guess at the encoding, you should  
 set ``r.encoding`` appropriately before accessing this property.  
 """  
  
 # Try charset from content-type  
 content = None  
 encoding = self.encoding  
  
 if not self.content:  
 return ""  
  
 # Fallback to auto-detected encoding.  
 if self.encoding is None:  
 encoding = self.apparent\_encoding  
  
 # Decode unicode from given encoding.  
 try:  
 content = str(self.content, encoding, errors="replace")  
 except (LookupError, TypeError):  
 # A LookupError is raised if the encoding was not found which could  
 # indicate a misspelling or similar mistake.  
 #  
 # A TypeError can be raised if encoding is None  
 #  
 # So we try blindly encoding.  
 content = str(self.content, errors="replace")  
  
 return content  
  
 def json(self, \*\*kwargs):  
 r"""Returns the json-encoded content of a response, if any.  
  
 :param \\*\\*kwargs: Optional arguments that ``json.loads`` takes.  
 :raises requests.exceptions.JSONDecodeError: If the response body does not  
 contain valid json.  
 """  
  
 if not self.encoding and self.content and len(self.content) > 3:  
 # No encoding set. JSON RFC 4627 section 3 states we should expect  
 # UTF-8, -16 or -32. Detect which one to use; If the detection or  
 # decoding fails, fall back to `self.text` (using charset\_normalizer to make  
 # a best guess).  
 encoding = guess\_json\_utf(self.content)  
 if encoding is not None:  
 try:  
 return complexjson.loads(self.content.decode(encoding), \*\*kwargs)  
 except UnicodeDecodeError:  
 # Wrong UTF codec detected; usually because it's not UTF-8  
 # but some other 8-bit codec. This is an RFC violation,  
 # and the server didn't bother to tell us what codec \*was\*  
 # used.  
 pass  
 except JSONDecodeError as e:  
 raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)  
  
 try:  
 return complexjson.loads(self.text, \*\*kwargs)  
 except JSONDecodeError as e:  
 # Catch JSON-related errors and raise as requests.JSONDecodeError  
 # This aliases json.JSONDecodeError and simplejson.JSONDecodeError  
 raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)  
  
 @property  
 def links(self):  
 """Returns the parsed header links of the response, if any."""  
  
 header = self.headers.get("link")  
  
 resolved\_links = {}  
  
 if header:  
 links = parse\_header\_links(header)  
  
 for link in links:  
 key = link.get("rel") or link.get("url")  
 resolved\_links[key] = link  
  
 return resolved\_links  
  
 def raise\_for\_status(self):  
 """Raises :class:`HTTPError`, if one occurred."""  
  
 http\_error\_msg = ""  
 if isinstance(self.reason, bytes):  
 # We attempt to decode utf-8 first because some servers  
 # choose to localize their reason strings. If the string  
 # isn't utf-8, we fall back to iso-8859-1 for all other  
 # encodings. (See PR #3538)  
 try:  
 reason = self.reason.decode("utf-8")  
 except UnicodeDecodeError:  
 reason = self.reason.decode("iso-8859-1")  
 else:  
 reason = self.reason  
  
 if 400 <= self.status\_code < 500:  
 http\_error\_msg = (  
 f"{self.status\_code} Client Error: {reason} for url: {self.url}"  
 )  
  
 elif 500 <= self.status\_code < 600:  
 http\_error\_msg = (  
 f"{self.status\_code} Server Error: {reason} for url: {self.url}"  
 )  
  
 if http\_error\_msg:  
 raise HTTPError(http\_error\_msg, response=self)  
  
 def close(self):  
 """Releases the connection back to the pool. Once this method has been  
 called the underlying ``raw`` object must not be accessed again.  
  
 \*Note: Should not normally need to be called explicitly.\*  
 """  
 if not self.\_content\_consumed:  
 self.raw.close()  
  
 release\_conn = getattr(self.raw, "release\_conn", None)  
 if release\_conn is not None:  
 release\_conn()

## File: venv/lib/python3.12/site-packages/rest\_framework/authtoken/models.py

import binascii  
import os  
  
from django.conf import settings  
from django.db import models  
from django.utils.translation import gettext\_lazy as \_  
  
  
class Token(models.Model):  
 """  
 The default authorization token model.  
 """  
 key = models.CharField(\_("Key"), max\_length=40, primary\_key=True)  
 user = models.OneToOneField(  
 settings.AUTH\_USER\_MODEL, related\_name='auth\_token',  
 on\_delete=models.CASCADE, verbose\_name=\_("User")  
 )  
 created = models.DateTimeField(\_("Created"), auto\_now\_add=True)  
  
 class Meta:  
 # Work around for a bug in Django:  
 # https://code.djangoproject.com/ticket/19422  
 #  
 # Also see corresponding ticket:  
 # https://github.com/encode/django-rest-framework/issues/705  
 abstract = 'rest\_framework.authtoken' not in settings.INSTALLED\_APPS  
 verbose\_name = \_("Token")  
 verbose\_name\_plural = \_("Tokens")  
  
 def save(self, \*args, \*\*kwargs):  
 if not self.key:  
 self.key = self.generate\_key()  
 return super().save(\*args, \*\*kwargs)  
  
 @classmethod  
 def generate\_key(cls):  
 return binascii.hexlify(os.urandom(20)).decode()  
  
 def \_\_str\_\_(self):  
 return self.key  
  
  
class TokenProxy(Token):  
 """  
 Proxy mapping pk to user pk for use in admin.  
 """  
 @property  
 def pk(self):  
 return self.user\_id  
  
 class Meta:  
 proxy = 'rest\_framework.authtoken' in settings.INSTALLED\_APPS  
 abstract = 'rest\_framework.authtoken' not in settings.INSTALLED\_APPS  
 verbose\_name = "token"

## File: venv/lib/python3.12/site-packages/rest\_framework\_simplejwt/models.py

from typing import TYPE\_CHECKING, Any, List, Optional, Union  
  
from django.contrib.auth import models as auth\_models  
from django.db.models.manager import EmptyManager  
from django.utils.functional import cached\_property  
  
from .settings import api\_settings  
  
if TYPE\_CHECKING:  
 from .tokens import Token  
  
  
class TokenUser:  
 """  
 A dummy user class modeled after django.contrib.auth.models.AnonymousUser.  
 Used in conjunction with the `JWTStatelessUserAuthentication` backend to  
 implement single sign-on functionality across services which share the same  
 secret key. `JWTStatelessUserAuthentication` will return an instance of this  
 class instead of a `User` model instance. Instances of this class act as  
 stateless user objects which are backed by validated tokens.  
 """  
  
 # User is always active since Simple JWT will never issue a token for an  
 # inactive user  
 is\_active = True  
  
 \_groups = EmptyManager(auth\_models.Group)  
 \_user\_permissions = EmptyManager(auth\_models.Permission)  
  
 def \_\_init\_\_(self, token: "Token") -> None:  
 self.token = token  
  
 def \_\_str\_\_(self) -> str:  
 return f"TokenUser {self.id}"  
  
 @cached\_property  
 def id(self) -> Union[int, str]:  
 return self.token[api\_settings.USER\_ID\_CLAIM]  
  
 @cached\_property  
 def pk(self) -> Union[int, str]:  
 return self.id  
  
 @cached\_property  
 def username(self) -> str:  
 return self.token.get("username", "")  
  
 @cached\_property  
 def is\_staff(self) -> bool:  
 return self.token.get("is\_staff", False)  
  
 @cached\_property  
 def is\_superuser(self) -> bool:  
 return self.token.get("is\_superuser", False)  
  
 def \_\_eq\_\_(self, other: object) -> bool:  
 if not isinstance(other, TokenUser):  
 return NotImplemented  
 return self.id == other.id  
  
 def \_\_ne\_\_(self, other: object) -> bool:  
 return not self.\_\_eq\_\_(other)  
  
 def \_\_hash\_\_(self) -> int:  
 return hash(self.id)  
  
 def save(self) -> None:  
 raise NotImplementedError("Token users have no DB representation")  
  
 def delete(self) -> None:  
 raise NotImplementedError("Token users have no DB representation")  
  
 def set\_password(self, raw\_password: str) -> None:  
 raise NotImplementedError("Token users have no DB representation")  
  
 def check\_password(self, raw\_password: str) -> None:  
 raise NotImplementedError("Token users have no DB representation")  
  
 @property  
 def groups(self) -> auth\_models.Group:  
 return self.\_groups  
  
 @property  
 def user\_permissions(self) -> auth\_models.Permission:  
 return self.\_user\_permissions  
  
 def get\_group\_permissions(self, obj: Optional[object] = None) -> set:  
 return set()  
  
 def get\_all\_permissions(self, obj: Optional[object] = None) -> set:  
 return set()  
  
 def has\_perm(self, perm: str, obj: Optional[object] = None) -> bool:  
 return False  
  
 def has\_perms(self, perm\_list: List[str], obj: Optional[object] = None) -> bool:  
 return False  
  
 def has\_module\_perms(self, module: str) -> bool:  
 return False  
  
 @property  
 def is\_anonymous(self) -> bool:  
 return False  
  
 @property  
 def is\_authenticated(self) -> bool:  
 return True  
  
 def get\_username(self) -> str:  
 return self.username  
  
 def \_\_getattr\_\_(self, attr: str) -> Optional[Any]:  
 """This acts as a backup attribute getter for custom claims defined in Token serializers."""  
 return self.token.get(attr, None)

## File: venv/lib/python3.12/site-packages/rest\_framework\_simplejwt/token\_blacklist/models.py

from django.conf import settings  
from django.db import models  
  
  
class OutstandingToken(models.Model):  
 id = models.BigAutoField(primary\_key=True, serialize=False)  
 user = models.ForeignKey(  
 settings.AUTH\_USER\_MODEL, on\_delete=models.SET\_NULL, null=True, blank=True  
 )  
  
 jti = models.CharField(unique=True, max\_length=255)  
 token = models.TextField()  
  
 created\_at = models.DateTimeField(null=True, blank=True)  
 expires\_at = models.DateTimeField()  
  
 class Meta:  
 # Work around for a bug in Django:  
 # https://code.djangoproject.com/ticket/19422  
 #  
 # Also see corresponding ticket:  
 # https://github.com/encode/django-rest-framework/issues/705  
 abstract = (  
 "rest\_framework\_simplejwt.token\_blacklist" not in settings.INSTALLED\_APPS  
 )  
 ordering = ("user",)  
  
 def \_\_str\_\_(self) -> str:  
 return "Token for {} ({})".format(  
 self.user,  
 self.jti,  
 )  
  
  
class BlacklistedToken(models.Model):  
 id = models.BigAutoField(primary\_key=True, serialize=False)  
 token = models.OneToOneField(OutstandingToken, on\_delete=models.CASCADE)  
  
 blacklisted\_at = models.DateTimeField(auto\_now\_add=True)  
  
 class Meta:  
 # Work around for a bug in Django:  
 # https://code.djangoproject.com/ticket/19422  
 #  
 # Also see corresponding ticket:  
 # https://github.com/encode/django-rest-framework/issues/705  
 abstract = (  
 "rest\_framework\_simplejwt.token\_blacklist" not in settings.INSTALLED\_APPS  
 )  
  
 def \_\_str\_\_(self) -> str:  
 return f"Blacklisted token for {self.token.user}"