accounts.admin

from django.contrib import admin

from .models import Profile

@admin.register(Profile)

class ProfileAdmin(admin.ModelAdmin):

list_display = ("user", "is_seller", "stripe_onboarding_complete", "is_owner", "created_at")

list_filter = ("is_seller", "stripe_onboarding_complete", "is_owner")

search_fields = ("user__username", "email", "first_name", "last_name")

readonly_fields = ("created_at", "updated_at")

accounts.apps

from __future__ import annotations

from django.apps import AppConfig

class AccountsConfig(AppConfig):

default_auto_field = "django.db.models.BigAutoField"

name = "accounts"

def ready(self) -> None:

Signal registration

from . import signals # noqa: F401

accounts.forms

from __future__ import annotations

from django import forms

from django.contrib.auth import get_user_model

from django.contrib.auth.forms import AuthenticationForm, UserCreationForm

from .models import Profile

User = get_user_model()

class UsernameAuthenticationForm(AuthenticationForm):

"""Standard username/password login form (Django default)."""

username = forms.CharField(

max_length=150,

widget=forms.TextInput(attrs={"autocomplete": "username", "placeholder": "Username"}),

)

password = forms.CharField(

widget=forms.PasswordInput(attrs={"autocomplete": "current-password", "placeholder": "Password"}),

)

class RegisterForm(UserCreationForm):

"""Registration form.

- username is required (public identity)

- user chooses consumer or seller

- profile stores email + role flags (seeded at registration; rest optional)

Option A:

- Profile row is created via signal.

- This form seeds Profile fields after user creation.

"""

email = forms.EmailField(required=False)

register_as_seller = forms.BooleanField(

required=False,

initial=False,

help_text="Check this if you want to register as a seller (Stripe onboarding required later).",

)

class Meta:

model = User

fields = ("username", "email", "password1", "password2")

def clean_username(self):

username = (self.cleaned_data.get("username") or "").strip()

if User.objects.filter(username__iexact=username).exists():

raise forms.ValidationError("That username is already taken.")

return username

def save(self, commit: bool = True):

Create the user first

user = super().save(commit=False)

Seed user.email too (useful for auth flows, Stripe, admin, etc.)

email = (self.cleaned_data.get("email") or "").strip()

if hasattr(user, "email"):

user.email = email

if commit:

user.save()

Profile is created via signal; seed it with registration details.

profile = getattr(user, "profile", None)

if profile is not None:

profile.email = email

profile.is_seller = bool(self.cleaned_data.get("register_as_seller", False))

profile.save(update_fields=["email", "is_seller", "updated_at"])

return user

class ProfileForm(forms.ModelForm):

class Meta:

model = Profile

fields = [

"first_name",

"last_name",

"email",

"phone_1",

"phone_2",

"address_1",

"address_2",

"city",

"state",

"zip_code",

"avatar",

"is_seller", # allow opt-in; Stripe gating happens elsewhere

]

widgets = {

"first_name": forms.TextInput(attrs={"placeholder": "First name"}),

"last_name": forms.TextInput(attrs={"placeholder": "Last name"}),

"email": forms.EmailInput(attrs={"placeholder": "Email"}),

"phone_1": forms.TextInput(attrs={"placeholder": "Phone 1"}),

"phone_2": forms.TextInput(attrs={"placeholder": "Phone 2"}),

"address_1": forms.TextInput(attrs={"placeholder": "Address 1"}),

"address_2": forms.TextInput(attrs={"placeholder": "Address 2"}),

"city": forms.TextInput(attrs={"placeholder": "City"}),

"zip_code": forms.TextInput(attrs={"placeholder": "ZIP"}),

}

def __init__(self, *args, **kwargs):

self.user = kwargs.pop("user", None)

super().__init__(*args, **kwargs)

Owner/admin flags should not be editable here

if "is_owner" in self.fields:

self.fields.pop("is_owner")

accounts.models

from __future__ import annotations

from django.conf import settings

from django.core.validators import RegexValidator

from django.db import models

class Profile(models.Model):

"""Marketplace Profile.

Extends the configured AUTH_USER_MODEL with marketplace-specific profile data and role flags.

Roles:

- Consumer: default for any registered user

- Seller: can list products (requires Stripe onboarding later)

- Owner/Admin: full permissions; should be your account (can be enforced via superuser/staff too)

Notes:

- Public identity is username.

- Profile is created automatically via signal (Option A).

Seller identity:

- Some sellers are individuals; others are a "shop".

- `shop_name` is an optional *public* label used across the marketplace.

If blank, we fall back to username.

"""

user = models.OneToOneField(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="profile")

Contact / identity

first_name = models.CharField(max_length=150, blank=True)

last_name = models.CharField(max_length=150, blank=True)

Seller-facing identity (public)

shop_name = models.CharField(

max_length=80,

blank=True,

help_text="Optional public shop name. If blank, your username is shown.",

)

Used for correspondence; username is public

email = models.EmailField(blank=True)

phone_regex = RegexValidator(

```
regex=r"^[0-9\-\+\(\) ]{7,20}$",
message="Enter a valid phone number (digits and - + ( ) allowed).",
)
phone_1 = models.CharField(max_length=20, blank=True, validators=[phone_regex])
phone_2 = models.CharField(max_length=20, blank=True, validators=[phone_regex])
address_1 = models.CharField(max_length=255, blank=True)
address_2 = models.CharField(max_length=255, blank=True)
city = models.CharField(max_length=120, blank=True)
US_STATES = [
("AL", "Alabama"), ("AK", "Alaska"), ("AZ", "Arizona"), ("AR", "Arkansas"),
("CA", "California"), ("CO", "Colorado"), ("CT", "Connecticut"), ("DE", "Delaware"),
("FL", "Florida"), ("GA", "Georgia"), ("HI", "Hawaii"), ("ID", "Idaho"),
("IL", "Illinois"), ("IN", "Indiana"), ("IA", "Iowa"), ("KS", "Kansas"),
("KY", "Kentucky"), ("LA", "Louisiana"), ("ME", "Maine"), ("MD", "Maryland"),
("MA", "Massachusetts"), ("MI", "Michigan"), ("MN", "Minnesota"), ("MS", "Mississippi"),
("MO", "Missouri"), ("MT", "Montana"), ("NE", "Nebraska"), ("NV", "Nevada"),
("NH", "New Hampshire"), ("NJ", "New Jersey"), ("NM", "New Mexico"), ("NY", "New York"),
("NC", "North Carolina"), ("ND", "North Dakota"), ("OH", "Ohio"), ("OK", "Oklahoma"),
("OR", "Oregon"), ("PA", "Pennsylvania"), ("RI", "Rhode Island"), ("SC", "South Carolina"),
("SD", "South Dakota"), ("TN", "Tennessee"), ("TX", "Texas"), ("UT", "Utah"),
("VT", "Vermont"), ("VA", "Virginia"), ("WA", "Washington"), ("WV", "West Virginia"),
("WI", "Wisconsin"), ("WY", "Wyoming"),
("DC", "District of Columbia"),
]
state = models.CharField(max_length=2, blank=True, choices=US_STATES)
zip_code = models.CharField(max_length=10, blank=True)
avatar = models.ImageField(upload_to="avatars/", blank=True, null=True)
# Role flags
```

```
is_seller = models.BooleanField(default=False)
is_owner = models.BooleanField(default=False) # Owner/admin override in UI
# Stripe (legacy placeholders; primary source of truth is payments.SellerStripeAccount)
stripe_account_id = models.CharField(max_length=255, blank=True)
stripe_onboarding_complete = models.BooleanField(default=False)
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)
class Meta:
indexes = [
models.Index(fields=["is_seller"]),
models.Index(fields=["is_owner"]),
models.Index(fields=["shop_name"]),
]
def __str__(self) -> str:
return f"Profile<{self.user.username}>"
@property
def display_name(self) -> str:
# Public identity is username; name is optional
name = f"{self.first_name} {self.last_name}".strip()
return name or self.user.username
@property
def public_seller_name(self) -> str:
"""Public seller label used across the marketplace."""
return (self.shop_name or "").strip() or self.user.username
@property
def can_access_seller_dashboard(self) -> bool:
return self.is_owner or self.user.is_superuser or self.user.is_staff or self.is_seller
@property
```

def can_access_consumer_dashboard(self) -> bool:

return self.user.is_authenticated

@property

def can_access_admin_dashboard(self) -> bool:

return self.is_owner or self.user.is_superuser or self.user.is_staff

accounts.signals

from __future__ import annotations

from django.conf import settings

from django.db.models.signals import post_save

from django.dispatch import receiver

from .models import Profile

@receiver(post_save, sender=settings.AUTH_USER_MODEL)

def create_or_update_profile(sender, instance, created, **kwargs):

"""Ensure every user has a Profile.

- On create: create Profile and seed a few fields from the User object.

- On update: guarantee Profile exists (do not overwrite user-edited Profile fields).

"""

if created:

Profile.objects.create(

user=instance,

first_name=getattr(instance, "first_name", "") or "",

last_name=getattr(instance, "last_name", "") or "",

email=getattr(instance, "email", "") or "",

)

return

Profile.objects.get_or_create(user=instance)

accounts.urls

from django.urls import path

from . import views

app_name = "accounts"

urlpatterns = [

path("login/", views.login_view, name="login"),

path("logout/", views.logout_view, name="logout"),

path("register/", views.register_view, name="register"),

path("profile/", views.profile_view, name="profile"),

]

accounts.views

from __future__ import annotations

from django.contrib import messages

from django.contrib.auth import login, logout

from django.contrib.auth.decorators import login_required

from django.shortcuts import render, redirect

from django.urls import reverse

from .forms import RegisterForm, UsernameAuthenticationForm, ProfileForm

def login_view(request):

if request.user.is_authenticated:

return redirect("accounts:profile")

if request.method == "POST":

form = UsernameAuthenticationForm(request, data=request.POST)

if form.is_valid():

user = form.get_user()

login(request, user)

messages.success(request, "Welcome back.")

next_url = request.GET.get("next") or reverse("accounts:profile")

return redirect(next_url)

else:

```
form = UsernameAuthenticationForm(request)
return render(request, "accounts/login.html", {"form": form})
def logout_view(request):
logout(request)
messages.success(request, "You have been logged out.")
return redirect("accounts:login")
def register_view(request):
if request.user.is_authenticated:
return redirect("accounts:profile")
if request.method == "POST":
form = RegisterForm(request.POST)
if form.is_valid():
user = form.save()
login(request, user)
messages.success(request, "Account created.")
return redirect("accounts:profile")
else:
form = RegisterForm()
return render(request, "accounts/register.html", {"form": form})
@login_required
def profile_view(request):
# Profile is created via signal; assume it exists.
profile = request.user.profile
if request.method == "POST":
form = ProfileForm(request.POST, request.FILES, instance=profile, user=request.user)
if form.is_valid():
form.save()
messages.success(request, "Profile updated.")
```

return redirect("accounts:profile")

else:

form = ProfileForm(instance=profile, user=request.user)

return render(request, "accounts/profile.html", {"form": form, "profile": profile})

cart.cart

cart/cart.py

from __future__ import annotations

from dataclasses import dataclass

from decimal import Decimal

from typing import Dict, List

from products.models import Product

CART_SESSION_KEY = "hc3_cart_v1"

@dataclass(frozen=True)

class CartLine:

product: Product

quantity: int

@property

def unit_price(self) -> Decimal:

return product_unit_price(self.product)

@property

def line_total(self) -> Decimal:

return self.unit_price * self.quantity

def product_unit_price(product: Product) -> Decimal:

Free items are always 0.00

if getattr(product, "is_free", False):

return Decimal("0.00")

return Decimal(str(getattr(product, "price", "0.00")))

class Cart:

"""

Session-backed cart.

Data format in session:

{

"<product_id>": {"qty": 1}

}

"""

def __init__(self, request):

self.request = request

self.session = request.session

self.data: Dict[str, Dict[str, int]] = self.session.get(CART_SESSION_KEY, {})

def _save(self) -> None:

self.session[CART_SESSION_KEY] = self.data

self.session.modified = True

def clear(self) -> None:

self.data = {}

self._save()

def add(self, product: Product, quantity: int = 1) -> None:

if not product.is_active:

return

pid = str(product.pk)

Digital files forced to qty=1

if product.kind == Product.Kind.FILE:

quantity = 1

quantity = max(int(quantity), 1)

if pid in self.data:

if product.kind == Product.Kind.FILE:

self.data[pid]["qty"] = 1

```
else:
self.data[pid]["qty"] = max(1, int(self.data[pid]["qty"]) + quantity)
else:
self.data[pid] = {"qty": quantity}
self._save()
def set_quantity(self, product: Product, quantity: int) -> None:
pid = str(product.pk)
if pid not in self.data:
return
if product.kind == Product.Kind.FILE:
self.data[pid]["qty"] = 1
else:
q = int(quantity)
if q <= 0:
self.remove(product)
return
self.data[pid]["qty"] = q
self._save()
def remove(self, product: Product) -> None:
pid = str(product.pk)
if pid in self.data:
del self.data[pid]
self._save()
def product_ids(self) -> List[int]:
ids: List[int] = []
for k in self.data.keys():
try:
ids.append(int(k))
```

except ValueError:

continue

return ids

def lines(self) -> List[CartLine]:

ids = self.product_ids()

products = (

Product.objects.filter(pk__in=ids, is_active=True)

.select_related("category", "seller")

.prefetch_related("images")

)

by_id = {p.pk: p for p in products}

result: List[CartLine] = []

dirty = False

for pid_str, payload in list(self.data.items()):

try:

pid = int(pid_str)

except ValueError:

invalid key in session -> drop

del self.data[pid_str]

dirty = True

continue

product = by_id.get(pid)

if not product:

product no longer active or deleted -> drop it from session

del self.data[pid_str]

dirty = True

continue

qty = int(payload.get("qty", 1))

if product.kind == Product.Kind.FILE:

qty = 1

else:

qty = max(1, qty)

result.append(CartLine(product=product, quantity=qty))

if dirty:

self._save()

return result

def subtotal(self) -> Decimal:

total = Decimal("0.00")

for line in self.lines():

total += line.line_total

return total

def count_items(self) -> int:

count distinct lines, not quantities

return len(self.data)

cart.forms

from __future__ import annotations

from django import forms

class AddToCartForm(forms.Form):

product_id = forms.IntegerField(widget=forms.HiddenInput)

quantity = forms.IntegerField(min_value=1, initial=1, required=False)

class UpdateCartLineForm(forms.Form):

product_id = forms.IntegerField(widget=forms.HiddenInput)

quantity = forms.IntegerField(min_value=0, required=True) # 0 removes

cart.urls

from django.urls import path

from . import views

app_name = "cart"

urlpatterns = [

path("", views.cart_detail, name="detail"),

path("add/", views.cart_add, name="add"),

path("update/", views.cart_update, name="update"),

path("remove/<int:product_id>/", views.cart_remove, name="remove"),

path("clear/", views.cart_clear, name="clear"),

]

cart.views

cart/views.py

from __future__ import annotations

import logging

from typing import List, Tuple

from django.contrib import messages

from django.shortcuts import get_object_or_404, redirect, render

from django.views.decorators.http import require_POST

from payments.utils import seller_is_stripe_ready

from products.models import Product, ProductEngagementEvent

from products.permissions import is_owner_user

from .cart import Cart

logger = logging.getLogger(__name__)

==

Helpers

==

def _seller_block_reason(*, request, product: Product) -> str | None:

"""

Return a human-readable reason if a product cannot be purchased.

IMPORTANT:

- Owner bypass is based on request.user (not the product's seller).

- Seller readiness is enforced server-side for cart add/update and order placement.

"""

try:

if request.user.is_authenticated and is_owner_user(request.user):

return None

except Exception:

pass

seller = getattr(product, "seller", None)

if seller and not seller_is_stripe_ready(seller):

return "Seller hasn’t completed payout setup yet."

return None

def _log_add_to_cart_throttled(request, *, product: Product) -> None:

"""Log ADD_TO_CART with a short session throttle to avoid spam."""

try:

key = f"hc3_event_add_to_cart_{product.id}"

if request.session.get(key):

return

ProductEngagementEvent.objects.create(

product=product,

event_type=ProductEngagementEvent.EventType.ADD_TO_CART,

)

request.session[key] = True

request.session.modified = True

except Exception:

return

def _prune_blocked_items(request, cart: Cart) -> Tuple[List[str], List[str]]:

"""

Remove items that are no longer purchasable (seller not ready, inactive, etc.)

Returns (removed_product_titles, unready_seller_usernames).

"""

removed_titles: List[str] = []

unready: List[str] = []

for line in cart.lines():

product = line.product

if not getattr(product, "is_active", True):

cart.remove(product)

removed_titles.append(getattr(product, "title", str(product.pk)))

continue

reason = _seller_block_reason(request=request, product=product)

if reason:

cart.remove(product)

removed_titles.append(getattr(product, "title", str(product.pk)))

try:

unready.append(getattr(product.seller, "username", "Seller"))

except Exception:

unready.append("Seller")

de-dupe seller list while preserving order

seen = set()

unready_sellers: List[str] = []

for u in unready:

if u in seen:

continue

seen.add(u)

unready_sellers.append(u)

return removed_titles, unready_sellers

```
# ============================================================
# Views
# ============================================================
def cart_detail(request):
cart = Cart(request)
removed_titles, unready_sellers = _prune_blocked_items(request, cart)
if removed_titles:
messages.warning(
request,
"Some items were removed from your cart because they can’t be checked out right now: "
+ ", ".join(removed_titles),
)
cart_lines = cart.lines()
subtotal = cart.subtotal()
can_checkout = bool(cart_lines) and not bool(unready_sellers)
return render(
request,
"cart/cart_detail.html",
{
"cart": cart,
"cart_lines": cart_lines,
"subtotal": subtotal,
"unready_sellers": unready_sellers,
"can_checkout": can_checkout,
# Template helper (avoid hardcoding "FILE" in templates)
"KIND_FILE": getattr(Product.Kind, "FILE", "file"),
},
)
```

@require_POST

def cart_add(request):

cart = Cart(request)

product_id = (request.POST.get("product_id") or "").strip()

qty_raw = (request.POST.get("quantity") or "1").strip()

try:

quantity = int(qty_raw)

except Exception:

quantity = 1

product = get_object_or_404(

Product.objects.select_related("seller", "category").prefetch_related("images"),

pk=product_id,

is_active=True,

)

reason = _seller_block_reason(request=request, product=product)

if reason:

messages.error(request, reason)

return redirect(product.get_absolute_url())

cart.add(product, quantity=quantity)

_log_add_to_cart_throttled(request, product=product)

messages.success(request, "Added to cart.")

next_url = (request.POST.get("next") or "").strip()

if next_url:

return redirect(next_url)

return redirect("cart:detail")

@require_POST

def cart_update(request):

cart = Cart(request)

```
product_id = (request.POST.get("product_id") or "").strip()
qty_raw = (request.POST.get("quantity") or "1").strip()
try:
quantity = int(qty_raw)
except Exception:
quantity = 1
product = get_object_or_404(Product.objects.select_related("seller"), pk=product_id)
if not product.is_active:
cart.remove(product)
messages.info(request, "Item removed (no longer available).")
return redirect("cart:detail")
reason = _seller_block_reason(request=request, product=product)
if reason:
cart.remove(product)
messages.error(request, f"Removed from cart: {reason}")
return redirect("cart:detail")
cart.set_quantity(product, quantity)
messages.success(request, "Cart updated.")
return redirect("cart:detail")
@require_POST
def cart_remove(request, product_id: int):
cart = Cart(request)
product = get_object_or_404(Product, pk=product_id)
cart.remove(product)
messages.info(request, "Item removed.")
return redirect("cart:detail")
@require_POST
def cart_clear(request):
```

cart = Cart(request)

cart.clear()

messages.info(request, "Cart cleared.")

return redirect("cart:detail")

catalog.admin

from __future__ import annotations

from django import forms

from django.contrib import admin

from .models import Category, RootCategory, SubCategory

Forms

----

class RootCategoryAdminForm(forms.ModelForm):

class Meta:

model = RootCategory

fields = "__all__"

def clean_parent(self):

Root categories must never have a parent

return None

class SubCategoryAdminForm(forms.ModelForm):

class Meta:

model = SubCategory

fields = "__all__"

def __init__(self, *args, **kwargs):

super().__init__(*args, **kwargs)

Present "parent" as "Category" in the UI

self.fields["parent"].label = "Category"

Limit selectable parents to ROOT categories

qs = Category.objects.filter(parent__isnull=True)

If type is known, filter parents by that type too (MODEL vs FILE)

chosen_type = None

if self.data.get("type"):

chosen_type = self.data.get("type")

elif getattr(self.instance, "type", None):

chosen_type = self.instance.type

if chosen_type:

qs = qs.filter(type=chosen_type)

self.fields["parent"].queryset = qs.order_by("type", "sort_order", "name")

def clean_parent(self):

parent = self.cleaned_data.get("parent")

if parent is None:

raise forms.ValidationError("A Subcategory must belong to a Category.")

if parent.parent_id is not None:

raise forms.ValidationError("Subcategories can only be one level deep (pick a root Category).")

chosen_type = self.cleaned_data.get("type")

if chosen_type and parent.type != chosen_type:

raise forms.ValidationError("Subcategory type must match the selected Category type.")

return parent

Base Category admin (needed for autocomplete)

Hidden from admin index/menu

@admin.register(Category)

class CategoryHiddenAdmin(admin.ModelAdmin):

search_fields = ("name", "slug", "description")

list_filter = ("type", "is_active")

ordering = ("type", "sort_order", "name")

def has_module_permission(self, request):

Hide "Categories" (base model) from the sidebar/index

return False

Split UX: Categories vs Subcategories

@admin.register(RootCategory)

class RootCategoryAdmin(admin.ModelAdmin):

form = RootCategoryAdminForm

list_display = ("name", "type", "is_active", "sort_order", "updated_at")

list_filter = ("type", "is_active")

search_fields = ("name", "slug", "description")

list_editable = ("is_active", "sort_order")

prepopulated_fields = {"slug": ("name",)}

fieldsets = (

("Core", {"fields": ("type", "name", "slug", "description")}),

("Display", {"fields": ("is_active", "sort_order")}),

("Timestamps", {"fields": ("created_at", "updated_at")}),

)

readonly_fields = ("created_at", "updated_at")

def get_queryset(self, request):

qs = super().get_queryset(request)

return qs.filter(parent__isnull=True)

@admin.register(SubCategory)

class SubCategoryAdmin(admin.ModelAdmin):

form = SubCategoryAdminForm

list_display = ("name", "type", "parent", "is_active", "sort_order", "updated_at")

```
list_filter = ("type", "is_active")
search_fields = ("name", "slug", "description", "parent__name")
list_editable = ("is_active", "sort_order")
autocomplete_fields = ("parent",)
prepopulated_fields = {"slug": ("name",)}
fieldsets = (
("Core", {"fields": ("type", "name", "slug", "parent", "description")}),
("Display", {"fields": ("is_active", "sort_order")}),
("Timestamps", {"fields": ("created_at", "updated_at")}),
)
readonly_fields = ("created_at", "updated_at")
def get_queryset(self, request):
qs = super().get_queryset(request)
return qs.filter(parent__isnull=False)
```

```
catalog.context_processors
from __future__ import annotations
from .models import Category
def sidebar_categories(request):
"""
Provides two separate category trees for the global sidebar:
- model_categories: roots (type=MODEL)
- file_categories: roots (type=FILE)
Children will be accessed via .children in templates.
"""
model_categories = (
Category.objects.filter(type=Category.CategoryType.MODEL, parent__isnull=True, is_active=True)
.prefetch_related("children")
.order_by("sort_order", "name")
```

```
)
file_categories = (
Category.objects.filter(type=Category.CategoryType.FILE, parent__isnull=True, is_active=True)
.prefetch_related("children")
.order_by("sort_order", "name")
)
return {
"sidebar_model_categories": model_categories,
"sidebar_file_categories": file_categories,
}
catalog.models
from __future__ import annotations
from django.db import models
from django.urls import reverse
from django.utils.text import slugify
class Category(models.Model):
class CategoryType(models.TextChoices):
MODEL = "MODEL", "3D Models"
FILE = "FILE", "3D Files"
type = models.CharField(max_length=10, choices=CategoryType.choices)
name = models.CharField(max_length=120)
slug = models.SlugField(max_length=140)
# Root categories have parent = NULL.
# Subcategories have parent = a root Category.
parent = models.ForeignKey(
"self",
null=True,
blank=True,
```

```
on_delete=models.CASCADE,
related_name="children",
)
description = models.TextField(blank=True)
is_active = models.BooleanField(default=True)
sort_order = models.PositiveIntegerField(default=0)
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)
class Meta:
unique_together = (
("type", "parent", "slug"),
)
indexes = [
models.Index(fields=["type", "is_active", "sort_order"]),
models.Index(fields=["parent", "is_active", "sort_order"]),
models.Index(fields=["slug"]),
]
ordering = ["type", "sort_order", "name"]
def __str__(self) -> str:
if self.parent:
return f"{self.get_type_display()} :: {self.parent.name} > {self.name}"
return f"{self.get_type_display()} :: {self.name}"
def save(self, *args, **kwargs):
if not self.slug:
self.slug = slugify(self.name)[:140]
super().save(*args, **kwargs)
@property
def is_root(self) -> bool:
```

return self.parent_id is None

def get_absolute_url(self) -> str:

return reverse("catalog:category_detail", kwargs={"pk": self.pk})

Admin UX: Category vs Subcategory

Proxy models (NO DB changes)

----

class RootCategory(Category):

class Meta:

proxy = True

verbose_name = "Category"

verbose_name_plural = "Categories"

class SubCategory(Category):

class Meta:

proxy = True

verbose_name = "Subcategory"

verbose_name_plural = "Subcategories"

catalog.urls

from django.urls import path

from . import views

app_name = "catalog"

urlpatterns = [

path("", views.category_list, name="category_list"),

path("<int:pk>/", views.category_detail, name="category_detail"),

]

catalog.views

from __future__ import annotations

from django.db.models import Prefetch

```
from django.shortcuts import get_object_or_404, render
from .models import Category
from products.models import Product
def category_list(request):
"""
Browse categories (top-level)
"""
model_roots = (
Category.objects.filter(type=Category.CategoryType.MODEL, parent__isnull=True, is_active=True)
.prefetch_related("children")
.order_by("sort_order", "name")
)
file_roots = (
Category.objects.filter(type=Category.CategoryType.FILE, parent__isnull=True, is_active=True)
.prefetch_related("children")
.order_by("sort_order", "name")
)
return render(
request,
"catalog/category_list.html",
{"model_roots": model_roots, "file_roots": file_roots},
)
def category_detail(request, pk: int):
"""
Category page: show products for the category (and optionally its descendants).
MVP behavior:
- show products in this category + direct children
- show inactive products? NO (only active)
```

"""

category = get_object_or_404(Category.objects.select_related("parent"), pk=pk, is_active=True)

Include this category + direct children (MVP)

child_ids = list(category.children.filter(is_active=True).values_list("id", flat=True))

category_ids = [category.id] + child_ids

products_qs = (

Product.objects.filter(is_active=True, category_id__in=category_ids)

.select_related("category", "seller")

.prefetch_related("images")

.order_by("-created_at")

)

For page sidebar / nav: show children as quick chips

children = category.children.filter(is_active=True).order_by("sort_order", "name")

return render(

request,

"catalog/category_detail.html",

{

"category": category,

"children": children,

"products": products_qs,

},

)

core.admin

from __future__ import annotations

from django.contrib import admin

from django.http import HttpRequest

from django.urls import reverse

from django.shortcuts import redirect

from django.contrib import messages

from .models import SiteConfig

@admin.register(SiteConfig)

class SiteConfigAdmin(admin.ModelAdmin):

list_display = (

"id",

"marketplace_sales_percent",

"platform_fee_cents",

"default_currency",

"allowed_shipping_countries_csv",

"updated_at",

)

def has_add_permission(self, request: HttpRequest) -> bool:

singleton: allow add only if none exists

return not SiteConfig.objects.exists()

def changelist_view(self, request: HttpRequest, extra_context=None):

"""

Convenience: if exactly one SiteConfig exists, jump directly to its edit page.

"""

qs = SiteConfig.objects.all()

if qs.count() == 1:

obj = qs.first()

url = reverse("admin:core_siteconfig_change", args=[obj.pk])

return redirect(url)

return super().changelist_view(request, extra_context=extra_context)

core.apps

from __future__ import annotations

from django.apps import AppConfig

class CoreConfig(AppConfig):

default_auto_field = "django.db.models.BigAutoField"

name = "core"

def ready(self) -> None:

ensure signals register

from . import signals # noqa: F401

core.config

from __future__ import annotations

from decimal import Decimal

from typing import Optional

from django.core.cache import cache

from .models import SiteConfig

CACHE_KEY = "core:site_config:v1"

CACHE_TTL_SECONDS = 30 # short TTL so admin changes take effect quickly

def get_site_config(*, use_cache: bool = True) -> SiteConfig:

"""

Returns the singleton SiteConfig.

Fresh DB case: auto-creates one row with model defaults.

This avoids boot errors on a brand-new DB.

"""

if use_cache:

cached = cache.get(CACHE_KEY)

if isinstance(cached, SiteConfig):

return cached

obj = SiteConfig.objects.first()

if obj is None:

obj = SiteConfig.objects.create()

cache.set(CACHE_KEY, obj, CACHE_TTL_SECONDS)

return obj

def invalidate_site_config_cache() -> None:

cache.delete(CACHE_KEY)

def get_marketplace_sales_percent() -> Decimal:

cfg = get_site_config()

return Decimal(cfg.marketplace_sales_percent or Decimal("0"))

def get_marketplace_sales_rate() -> Decimal:

10.00 -> 0.10

pct = get_marketplace_sales_percent()

try:

return (pct / Decimal("100"))

except Exception:

return Decimal("0")

def get_platform_fee_cents() -> int:

cfg = get_site_config()

try:

return int(cfg.platform_fee_cents or 0)

except Exception:

return 0

def get_allowed_shipping_countries() -> list[str]:

cfg = get_site_config()

return cfg.allowed_shipping_countries

core.context_processors

from __future__ import annotations

from typing import Any

from payments.models import SellerStripeAccount

from products.permissions import is_owner_user, is_seller_user

def sidebar_flags(request) -> dict[str, Any]:

"""

Global sidebar flags used by templates/partials/sidebar_dashboard.html.

Keeps dashboards templates stable (no need to remember passing these in every view).

"""

user = getattr(request, "user", None)

if not user or not getattr(user, "is_authenticated", False):

return {

"user_is_owner": False,

"user_is_seller": False,

"seller_stripe_ready": None,

}

owner = bool(is_owner_user(user))

seller = bool(is_seller_user(user))

Only compute readiness if they're a seller (or owner who can see seller areas).

Owner may not have a SellerStripeAccount, so keep it None in that case.

stripe_ready = None

if seller:

acct = SellerStripeAccount.objects.filter(user=user).only(

"stripe_account_id",

"details_submitted",

"charges_enabled",

"payouts_enabled",

).first()

stripe_ready = bool(acct.is_ready) if acct else False

return {

"user_is_owner": owner,

"user_is_seller": seller,

"seller_stripe_ready": stripe_ready,

}

core.models

from __future__ import annotations

from decimal import Decimal

from typing import Any

from django.db import models

class SiteConfig(models.Model):

"""

DB-backed site settings (singleton).

STRICT RULE:

- Any site setting MUST live here (so it's editable via Django admin/dashboard).

- No "settings.py constants" for runtime-tunable business rules.

"""

Marketplace fee: percent of seller gross (e.g. 10.00 -> 10%)

marketplace_sales_percent = models.DecimalField(

max_digits=6,

decimal_places=2,

default=Decimal("10.00"),

help_text="Percent of sales withheld by the marketplace (e.g., 10.00 = 10%).",

)

Optional fixed platform fee in cents (kept here even if you start at 0)

platform_fee_cents = models.PositiveIntegerField(

default=0,

help_text="Optional fixed fee in cents added to each order (0 disables).",

)

Currency defaults

default_currency = models.CharField(

max_length=8,

default="usd",

help_text="Default currency (Stripe-style), e.g. 'usd'.",

)

Shipping configuration

Store as JSON to avoid Postgres ArrayField dependency issues.

allowed_shipping_countries = models.JSONField(

default=list,

blank=True,

help_text="List of allowed country codes for shipping (e.g. ['US']).",

)

created_at = models.DateTimeField(auto_now_add=True)

updated_at = models.DateTimeField(auto_now=True)

class Meta:

verbose_name = "Site Config"

verbose_name_plural = "Site Config"

def __str__(self) -> str:

return "SiteConfig"

@property

def allowed_shipping_countries_csv(self) -> str:

try:

codes = self.allowed_shipping_countries or []

if not isinstance(codes, list):

return ""

cleaned = [str(x).strip().upper() for x in codes if str(x).strip()]

return ",".join(cleaned)

except Exception:

return ""

def clean(self) -> None:

Normalize JSON list field and defaults.

try:

codes = self.allowed_shipping_countries

if not codes:

self.allowed_shipping_countries = ["US"]

elif isinstance(codes, list):

cleaned = [str(x).strip().upper() for x in codes if str(x).strip()]

self.allowed_shipping_countries = cleaned or ["US"]

else:

If someone put a non-list in JSONField, reset safely

self.allowed_shipping_countries = ["US"]

except Exception:

self.allowed_shipping_countries = ["US"]

Clamp percent to sane bounds

try:

pct = Decimal(self.marketplace_sales_percent or Decimal("0"))

except Exception:

pct = Decimal("0")

if pct < 0:

self.marketplace_sales_percent = Decimal("0.00")

elif pct > 100:

self.marketplace_sales_percent = Decimal("100.00")

def save(self, *args: Any, **kwargs: Any) -> None:

Ensure normalization runs even when changed in admin.

self.clean()

super().save(*args, **kwargs)

STRICT cache invalidation (no stale settings)

try:

from .config import invalidate_site_config_cache

invalidate_site_config_cache()

except Exception:

pass

core.signals

from __future__ import annotations

from django.db.models.signals import post_migrate

from django.dispatch import receiver

from .config import invalidate_site_config_cache

from .models import SiteConfig

@receiver(post_migrate)

def ensure_site_config(sender, **kwargs):

"""

Ensure exactly one SiteConfig exists.

Runs after migrations; safe on a brand-new database.

"""

try:

SiteConfig.objects.get_or_create(id=1, defaults={"allowed_shipping_countries": ["US"]})

except Exception:

If PK=1 already taken or DB behavior differs, fall back to "first or create".

if not SiteConfig.objects.exists():

SiteConfig.objects.create(allowed_shipping_countries=["US"])

invalidate_site_config_cache()

core.site_settings

from __future__ import annotations

from decimal import Decimal

from typing import Dict, Tuple

from django.db import transaction

from .models import SiteSetting

(default_value, description)

DEFAULTS: Dict[str, Tuple[str, str]] = {

Platform cut taken from sales (percent). Example: "10.0" => 10%

"marketplace_sales_percent": ("10.0", "Platform cut of each sale, as a percent (e.g. 10.0)."),

You mentioned adding a platform fee later (flat fee). Leave default at 0 for now.

"order_platform_fee_cents": ("0", "Optional flat fee per order in cents (0 disables)."),

}

def ensure_defaults_exist() -> None:

"""

Ensures all DEFAULTS keys exist in the DB.

Safe to call at runtime.

"""

Avoid wrapping in atomic unless you want strict consistency; this is fine.

for key, (val, desc) in DEFAULTS.items():

SiteSetting.objects.get_or_create(

key=key,

defaults={"value": val, "description": desc},

)

def get_str(key: str, default: str = "") -> str:

obj = SiteSetting.objects.filter(key=key).first()

if obj is None:

return default

return (obj.value or "").strip()

def get_int(key: str, default: int = 0) -> int:

obj = SiteSetting.objects.filter(key=key).first()

if obj is None:

return default

return obj.as_int(default=default)

def get_decimal(key: str, default: Decimal = Decimal("0")) -> Decimal:

obj = SiteSetting.objects.filter(key=key).first()

if obj is None:

return default

return obj.as_decimal(default=default)

def get_bool(key: str, default: bool = False) -> bool:

obj = SiteSetting.objects.filter(key=key).first()

if obj is None:

return default

return obj.as_bool(default=default)

def marketplace_sales_percent() -> Decimal:

"""

The sales cut percent (e.g. 10.0).

"""

ensure_defaults_exist()

return get_decimal("marketplace_sales_percent", default=Decimal(DEFAULTS["marketplace_sales_percent"][0]))

def marketplace_sales_rate() -> Decimal:

"""

The sales cut rate (e.g. 0.10).

"""

pct = marketplace_sales_percent()

try:

return (pct / Decimal("100"))

except Exception:

return Decimal("0.10")

core.views

from __future__ import annotations

```
from datetime import timedelta
from django.db.models import Avg, Count, F, FloatField, Q, Value
from django.db.models.functions import Coalesce
from django.shortcuts import render, redirect
from django.utils import timezone
from orders.models import Order
from payments.models import SellerStripeAccount
from products.models import Product, ProductEngagementEvent
from products.permissions import is_owner_user
HOME_BUCKET_SIZE = 8
TRENDING_WINDOW_DAYS = 30
def _base_home_qs():
return (
Product.objects.filter(is_active=True)
.select_related("seller", "category")
.prefetch_related("images")
)
def _annotate_rating(qs):
return qs.annotate(
avg_rating=Coalesce(Avg("reviews__rating"), Value(0.0), output_field=FloatField()),
review_count=Coalesce(Count("reviews", distinct=True), Value(0)),
)
def _annotate_trending(qs, *, since_days: int = TRENDING_WINDOW_DAYS):
since = timezone.now() - timedelta(days=since_days)
recent_purchases = Count(
"order_items",
filter=Q(
order_items__order__status=Order.Status.PAID,
```

```
order_items__order__paid_at__isnull=False,
order_items__order__paid_at__gte=since,
),
distinct=True,
)
recent_reviews = Count(
"reviews",
filter=Q(reviews__created_at__gte=since),
distinct=True,
)
recent_views = Count(
"engagement_events",
filter=Q(
engagement_events__event_type=ProductEngagementEvent.EventType.VIEW,
engagement_events__created_at__gte=since,
),
distinct=True,
)
recent_clicks = Count(
"engagement_events",
filter=Q(
engagement_events__event_type=ProductEngagementEvent.EventType.CLICK,
engagement_events__created_at__gte=since,
),
distinct=True,
)
recent_add_to_cart = Count(
"engagement_events",
```

```
filter=Q(
engagement_events__event_type=ProductEngagementEvent.EventType.ADD_TO_CART,
engagement_events__created_at__gte=since,
),
distinct=True,
)
qs = qs.annotate(
recent_purchases=Coalesce(recent_purchases, Value(0)),
recent_reviews=Coalesce(recent_reviews, Value(0)),
recent_views=Coalesce(recent_views, Value(0)),
recent_clicks=Coalesce(recent_clicks, Value(0)),
recent_add_to_cart=Coalesce(recent_add_to_cart, Value(0)),
)
qs = qs.annotate(
trending_score=(
Coalesce(F("recent_purchases"), Value(0)) * Value(6.0)
+ Coalesce(F("recent_add_to_cart"), Value(0)) * Value(3.0)
+ Coalesce(F("recent_clicks"), Value(0)) * Value(1.25)
+ Coalesce(F("recent_reviews"), Value(0)) * Value(2.0)
+ Coalesce(F("recent_views"), Value(0)) * Value(0.25)
+ Coalesce(F("avg_rating"), Value(0.0)) * Value(1.0)
)
)
return qs
def _seller_can_sell(product: Product) -> bool:
"""Single source of truth for buy-gating on the home page."""
try:
if product.seller and is_owner_user(product.seller):
```

```
return True
except Exception:
pass
try:
acct = getattr(product.seller, "stripe_connect", None)
if acct is not None:
return bool(acct.is_ready)
except Exception:
pass
try:
if not product.seller_id:
return False
return SellerStripeAccount.objects.filter(
user_id=product.seller_id,
stripe_account_id__gt="",
details_submitted=True,
charges_enabled=True,
payouts_enabled=True,
).exists()
except Exception:
return False
def _apply_can_buy_flag(products: list[Product]) -> None:
for p in products:
p.can_buy = _seller_can_sell(p)
def _apply_trending_badge_flag(products: list[Product], *, computed_ids: set[int] | None = None) -> None:
computed_ids = computed_ids or set()
for p in products:
p.trending_badge = bool(getattr(p, "is_trending", False) or (p.id in computed_ids))
```

```
def home(request):
# Logged-in users land on their smart dashboard hub.
if request.user.is_authenticated:
return redirect("dashboards:home")
qs = _base_home_qs()
qs = _annotate_rating(qs)
featured = list(qs.filter(is_featured=True).order_by("-created_at")[:HOME_BUCKET_SIZE])
new_items = list(qs.order_by("-created_at")[:HOME_BUCKET_SIZE])
manual_trending = list(qs.filter(is_trending=True).order_by("-created_at")[:HOME_BUCKET_SIZE])
manual_ids = {p.id for p in manual_trending}
trending_needed = max(0, HOME_BUCKET_SIZE - len(manual_trending))
computed_trending: list[Product] = []
computed_ids: set[int] = set()
if trending_needed > 0:
trending_qs = _annotate_trending(qs, since_days=TRENDING_WINDOW_DAYS).exclude(id__in=manual_ids)
computed_trending = list(
trending_qs.order_by("-trending_score", "-avg_rating", "-created_at")[:trending_needed]
)
computed_ids = {p.id for p in computed_trending if getattr(p, "trending_score", 0) > 0}
trending = manual_trending + computed_trending
exclude_ids = {p.id for p in featured} | {p.id for p in new_items} | {p.id for p in trending}
misc = list(qs.exclude(id__in=exclude_ids).order_by("-created_at")[:HOME_BUCKET_SIZE])
all_cards = featured + new_items + trending + misc
_apply_can_buy_flag(all_cards)
_apply_trending_badge_flag(all_cards, computed_ids=computed_ids)
return render(
request,
"core/home.html",
```

```
{
"featured": featured,
"trending": trending,
"new_items": new_items,
"misc": misc,
},
)
dashboards.urls
from django.urls import path
from . import views
app_name = "dashboards"
urlpatterns = [
path("", views.dashboard_home, name="home"),
path("consumer/", views.consumer_dashboard, name="consumer"),
path("seller/", views.seller_dashboard, name="seller"),
path("admin/", views.admin_dashboard, name="admin"),
]
dashboards.views
from __future__ import annotations
from datetime import timedelta
from decimal import Decimal
from django.contrib import messages
from django.contrib.auth.decorators import login_required
from django.db.models import Count, F, IntegerField, Sum
from django.db.models.expressions import ExpressionWrapper
from django.shortcuts import redirect, render
from django.urls import reverse
from django.utils import timezone
```

```
from core.config import get_site_config
from orders.models import Order, OrderItem
from payments.models import SellerStripeAccount, SellerBalanceEntry
from products.models import Product
from products.permissions import is_owner_user, is_seller_user
from payments.services import get_seller_balance_cents
DASH_RECENT_DAYS = 30
def _cents_to_dollars(cents: int) -> Decimal:
return (Decimal(int(cents or 0)) / Decimal("100")).quantize(Decimal("0.01"))
@login_required
def dashboard_home(request):
user = request.user
if is_owner_user(user):
return redirect("dashboards:admin")
if is_seller_user(user):
return redirect("dashboards:seller")
return redirect("dashboards:consumer")
@login_required
def consumer_dashboard(request):
user = request.user
orders = (
Order.objects.filter(buyer=user)
.prefetch_related("items", "items__product")
.order_by("-created_at")[:10]
)
totals = Order.objects.filter(buyer=user, status=Order.Status.PAID).aggregate(
total_spent_cents=Sum("total_cents"),
paid_count=Count("id"),
```

)

total_spent = _cents_to_dollars(int(totals.get("total_spent_cents") or 0))

return render(

request,

"dashboards/consumer_dashboard.html",

{

"orders": orders,

"total_spent": total_spent,

"paid_count": totals.get("paid_count") or 0,

},

)

@login_required

def seller_dashboard(request):

user = request.user

if not is_seller_user(user):

messages.info(request, "You don’t have access to the seller dashboard.")

return redirect("dashboards:consumer")

since = timezone.now() - timedelta(days=DASH_RECENT_DAYS)

stripe_obj, _ = SellerStripeAccount.objects.get_or_create(user=user)

balance_cents = get_seller_balance_cents(seller=user)

listings_total = Product.objects.filter(seller=user, is_active=True).count()

listings_inactive = Product.objects.filter(seller=user, is_active=False).count()

line_total_expr = ExpressionWrapper(

F("quantity") * F("unit_price_cents"),

output_field=IntegerField(),

)

recent_sales = (

OrderItem.objects.filter(

```
seller=user,
order__status=Order.Status.PAID,
order__paid_at__gte=since,
)
.select_related("order", "product")
.annotate(line_total_cents=line_total_expr)
.order_by("-created_at")[:15]
)
sales_totals = OrderItem.objects.filter(
seller=user,
order__status=Order.Status.PAID,
order__paid_at__gte=since,
).aggregate(
gross_cents=Sum(line_total_expr),
net_cents=Sum("seller_net_cents"),
order_count=Count("order_id", distinct=True),
sold_count=Sum("quantity"),
)
payout_available_cents = max(
0, int((sales_totals.get("net_cents") or 0) + balance_cents)
)
ledger_entries = (
SellerBalanceEntry.objects.filter(seller=user)
.order_by("-created_at")[:10]
)
return render(
request,
"dashboards/seller_dashboard.html",
```

{

"stripe": stripe_obj,

"ready": stripe_obj.is_ready,

"listings_total": listings_total,

"listings_inactive": listings_inactive,

"recent_sales": recent_sales,

"gross_revenue": _cents_to_dollars(int(sales_totals.get("gross_cents") or 0)),

"net_revenue": _cents_to_dollars(int(sales_totals.get("net_cents") or 0)),

"balance": _cents_to_dollars(balance_cents),

"payout_available": _cents_to_dollars(payout_available_cents),

"ledger_entries": ledger_entries,

"sold_count": sales_totals.get("sold_count") or 0,

"order_count": sales_totals.get("order_count") or 0,

"since_days": DASH_RECENT_DAYS,

},

)

@login_required

def admin_dashboard(request):

user = request.user

if not is_owner_user(user):

messages.info(request, "You don’t have access to the admin dashboard.")

return redirect("dashboards:consumer")

since = timezone.now() - timedelta(days=DASH_RECENT_DAYS)

cfg = get_site_config()

site_config_admin_url = reverse("admin:core_siteconfig_changelist")

products_total = Product.objects.count()

products_active = Product.objects.filter(is_active=True).count()

sellers_total = Product.objects.values("seller_id").distinct().count()

```
orders_paid = Order.objects.filter(status=Order.Status.PAID, paid_at__isnull=False).count()
orders_pending = Order.objects.filter(status=Order.Status.PENDING).count()
revenue_cents = (
Order.objects.filter(
status=Order.Status.PAID,
paid_at__isnull=False,
paid_at__gte=since,
).aggregate(total=Sum("subtotal_cents"))
).get("total") or 0
revenue_30 = _cents_to_dollars(int(revenue_cents))
line_total_expr = ExpressionWrapper(
F("quantity") * F("unit_price_cents"),
output_field=IntegerField(),
)
top_sellers = (
OrderItem.objects.filter(
order__status=Order.Status.PAID,
order__paid_at__isnull=False,
order__paid_at__gte=since,
)
.values("seller__username")
.annotate(
revenue_cents=Sum(line_total_expr),
qty=Sum("quantity"),
orders=Count("order_id", distinct=True),
)
.order_by("-revenue_cents")[:10]
)
```

```
top_sellers_display = []
for row in top_sellers:
top_sellers_display.append(
{
"seller__username": row.get("seller__username") or "",
"revenue": _cents_to_dollars(int(row.get("revenue_cents") or 0)),
"qty": row.get("qty") or 0,
"orders": row.get("orders") or 0,
}
)
return render(
request,
"dashboards/admin_dashboard.html",
{
"products_total": products_total,
"products_active": products_active,
"sellers_total": sellers_total,
"orders_paid": orders_paid,
"orders_pending": orders_pending,
"revenue_30": revenue_30,
"top_sellers": top_sellers_display,
"since_days": DASH_RECENT_DAYS,
"site_config_admin_url": site_config_admin_url,
"marketplace_sales_percent": getattr(cfg, "marketplace_sales_percent", 0) or 0,
"platform_fee_cents": int(getattr(cfg, "platform_fee_cents", 0) or 0),
},
)
orders.admin
```

```
# orders/admin.py
from __future__ import annotations
from dataclasses import dataclass
from decimal import Decimal
from typing import Any
from django.contrib import admin, messages
from django.db.models import (
Sum,
Count,
Exists,
OuterRef,
Value,
IntegerField,
BooleanField,
ExpressionWrapper,
F,
Case,
When,
Q,
)
from django.db.models.functions import Coalesce, Cast
from django.urls import reverse
from django.utils.html import format_html
from payments.models import SellerStripeAccount
from payments.services import get_seller_balance_cents
from .models import Order, OrderItem, OrderEvent, StripeWebhookEvent
from .stripe_service import create_transfers_for_paid_order
# IMPORTANT: must match the stored DB value (lowercase)
```

TRANSFER_EVENT_TYPE = OrderEvent.Type.TRANSFER_CREATED

========================

Helpers

========================

def cents_to_money(cents: int | None, currency: str = "usd") -> str:

if cents is None:

cents = 0

try:

amount = int(cents) / 100.0

except (TypeError, ValueError):

amount = 0.0

cur = (currency or "usd").upper()

return f"{cur}{amount:,.2f}"

def admin_order_change_url(order_id) -> str:

return reverse("admin:orders_order_change", args=[order_id])

========================

Admin Filters

========================

class PaidStateFilter(admin.SimpleListFilter):

title = "paid state"

parameter_name = "paid_state"

def lookups(self, request, model_admin):

return (("paid", "Paid"), ("unpaid", "Unpaid"))

def queryset(self, request, queryset):

val = self.value()

if val == "paid":

return queryset.filter(paid_at__isnull=False)

if val == "unpaid":

```
return queryset.filter(paid_at__isnull=True)
return queryset
class BuyerTypeFilter(admin.SimpleListFilter):
title = "buyer type"
parameter_name = "buyer_type"
def lookups(self, request, model_admin):
return (("user", "User account"), ("guest", "Guest checkout"))
def queryset(self, request, queryset):
val = self.value()
if val == "user":
return queryset.filter(buyer__isnull=False)
if val == "guest":
return queryset.filter(buyer__isnull=True).exclude(guest_email__exact="")
return queryset
class FulfillmentMixFilter(admin.SimpleListFilter):
title = "items"
parameter_name = "mix"
def lookups(self, request, model_admin):
return (
("shipping", "Has shippable items"),
("digital", "Has digital items"),
("physical_only", "Physical-only"),
("digital_only", "Digital-only"),
)
def queryset(self, request, queryset):
val = self.value()
if val == "shipping":
return queryset.filter(items__requires_shipping=True).distinct()
```

if val == "digital":

return queryset.filter(items__is_digital=True).distinct()

if val == "physical_only":

return queryset.filter(items__is_digital=False).distinct()

if val == "digital_only":

return (

queryset.filter(items__is_digital=True)

.exclude(items__is_digital=False)

.distinct()

)

return queryset

class PayoutStateFilter(admin.SimpleListFilter):

"""

Uses OrderEvent(type=transfer_created) as the authoritative "payout created" marker.

"""

title = "payout state"

parameter_name = "payout_state"

def lookups(self, request, model_admin):

return (

("unpaid", "Unpaid"),

("pending", "Paid, payout pending"),

("paid_out", "Paid out (transfer created)"),

("skipped_unready", "Payout skipped (seller not ready)"),

)

def queryset(self, request, queryset):

val = self.value()

if val == "unpaid":

return queryset.filter(paid_at__isnull=True)

if val == "pending":

return queryset.filter(paid_at__isnull=False, has_transfer_event=False, payout_skipped_unready_seller=False)

if val == "paid_out":

return queryset.filter(paid_at__isnull=False, has_transfer_event=True)

if val == "skipped_unready":

return queryset.filter(paid_at__isnull=False, has_transfer_event=False, payout_skipped_unready_seller=True)

return queryset

class ReconciliationFilter(admin.SimpleListFilter):

"""

Flags that show "something is off" for production reconciliation.

This relies on annotations done in OrderAdmin.get_queryset().

"""

title = "reconciliation"

parameter_name = "recon"

def lookups(self, request, model_admin):

return (

("ok", "OK"),

("totals_mismatch", "Totals mismatch (subtotal vs items gross)"),

("ledger_mismatch", "Ledger mismatch (expected fee/net vs stored)"),

("paid_missing_stripe", "Paid missing Stripe ids"),

("paid_missing_transfer", "Paid missing transfer event"),

("payout_skipped_unready", "Payout skipped (seller not ready)"),

)

def queryset(self, request, queryset):

val = self.value()

if val == "ok":

return queryset.filter(

totals_mismatch=False,

ledger_mismatch=False,

paid_missing_stripe_ids=False,

paid_missing_transfer_event=False,

payout_skipped_unready_seller=False,

)

if val == "totals_mismatch":

return queryset.filter(totals_mismatch=True)

if val == "ledger_mismatch":

return queryset.filter(ledger_mismatch=True)

if val == "paid_missing_stripe":

return queryset.filter(paid_missing_stripe_ids=True)

if val == "paid_missing_transfer":

return queryset.filter(paid_missing_transfer_event=True)

if val == "payout_skipped_unready":

return queryset.filter(payout_skipped_unready_seller=True)

return queryset

========================

Inlines

========================

class OrderItemInline(admin.TabularInline):

model = OrderItem

extra = 0

can_delete = False

fields = (

"id",

"product",

"seller",

"quantity",

"unit_price_cents",

"marketplace_fee_cents",

"seller_net_cents",

"is_digital",

"requires_shipping",

"created_at",

)

readonly_fields = fields

show_change_link = True

class OrderEventInline(admin.TabularInline):

model = OrderEvent

extra = 0

can_delete = False

fields = ("created_at", "type", "message")

readonly_fields = ("created_at", "type", "message")

ordering = ("-created_at",)

show_change_link = True

========================

Order Admin

========================

@dataclass(frozen=True)

class _SellerPayoutRow:

seller_id: str

seller_label: str

connect_ready: bool

gross_cents: int

net_cents: int

balance_cents: int

payout_cents: int

@admin.register(Order)

class OrderAdmin(admin.ModelAdmin):

date_hierarchy = "created_at"

inlines = [OrderItemInline, OrderEventInline]

list_select_related = ("buyer",)

list_display = (

"id",

"status",

"kind",

"currency",

"buyer_display",

"subtotal_money",

"total_money",

"items_gross_money",

"expected_fee_money",

"marketplace_fee_money",

"expected_net_money",

"seller_net_money",

"items_qty",

"seller_count",

"paid_at",

"payout_state_badge",

"recon_badge",

"created_at",

)

list_filter = (

"status",

"kind",

"currency",

PaidStateFilter,

BuyerTypeFilter,

FulfillmentMixFilter,

PayoutStateFilter,

ReconciliationFilter,

"paid_at",

"created_at",

)

search_fields = (

"id",

"guest_email",

"stripe_session_id",

"stripe_payment_intent_id",

"buyer__username",

"buyer__email",

)

raw_id_fields = ("buyer",)

readonly_fields = (

"id",

"order_token",

"created_at",

"updated_at",

"paid_at",

"subtotal_cents",

"tax_cents",

"shipping_cents",

```
"total_cents",
"marketplace_sales_percent_snapshot",
"platform_fee_cents_snapshot", # legacy; must remain 0
"payout_summary_html",
)
fieldsets = (
("Identity", {"fields": ("id", "status", "kind", "currency", "buyer", "guest_email", "order_token")}),
("Totals (cents)", {"fields": ("subtotal_cents", "tax_cents", "shipping_cents", "total_cents")}),
("Settings snapshots", {"fields": ("marketplace_sales_percent_snapshot", "platform_fee_cents_snapshot")}),
("Stripe", {"fields": ("stripe_session_id", "stripe_payment_intent_id", "paid_at")}),
("Payout summary", {"fields": ("payout_summary_html",)}),
(
"Shipping snapshot",
{
"fields": (
"shipping_name",
"shipping_phone",
"shipping_line1",
"shipping_line2",
"shipping_city",
"shipping_state",
"shipping_postal_code",
"shipping_country",
)
},
),
("Timestamps", {"fields": ("created_at", "updated_at")}),
)
```

```
actions = [
"add_reconciliation_warning_event",
"retry_payout_transfers",
]
def get_queryset(self, request):
qs = super().get_queryset(request)
# items gross: sum(quantity * unit_price_cents)
line_total_expr = ExpressionWrapper(
F("items__quantity") * F("items__unit_price_cents"),
output_field=IntegerField(),
)
qs = qs.annotate(
items_qty_agg=Coalesce(Sum("items__quantity"), Value(0), output_field=IntegerField()),
seller_count_agg=Coalesce(Count("items__seller", distinct=True), Value(0), output_field=IntegerField()),
items_gross_cents_agg=Coalesce(Sum(line_total_expr), Value(0), output_field=IntegerField()),
marketplace_fee_cents_agg=Coalesce(Sum("items__marketplace_fee_cents"), Value(0),
output_field=IntegerField()),
seller_net_cents_agg=Coalesce(Sum("items__seller_net_cents"), Value(0), output_field=IntegerField()),
)
# transfer marker
transfer_exists = OrderEvent.objects.filter(order_id=OuterRef("pk"), type=TRANSFER_EVENT_TYPE)
qs = qs.annotate(has_transfer_event=Exists(transfer_exists))
# payout skipped marker (from payout attempts)
# stripe_service creates WARNING with message: "transfer skipped seller=… (not ready)"
skipped_unready_exists = OrderEvent.objects.filter(
order_id=OuterRef("pk"),
type=OrderEvent.Type.WARNING,
message__icontains="transfer skipped",
).filter(message__icontains="not ready")
```

```
qs = qs.annotate(payout_skipped_unready_seller=Exists(skipped_unready_exists))
# -------- Expected fee/net with exact ROUND_HALF_UP (percent-only) --------
# marketplace_sales_percent_snapshot stored like 10.00 (Decimal)
# Convert to integer basis points: 10.00% => 1000 bps (1 bps = 0.01%)
# fee = round(gross * bps / 10000) => (gross*bps + 5000)//10000
bps = Cast(F("marketplace_sales_percent_snapshot") * Value(100), IntegerField())
expected_fee_cents = ExpressionWrapper(
(F("items_gross_cents_agg") * bps + Value(5000)) / Value(10000),
output_field=IntegerField(),
)
qs = qs.annotate(
expected_fee_cents_agg=Coalesce(expected_fee_cents, Value(0), output_field=IntegerField()),
).annotate(
expected_net_cents_agg=Coalesce(
F("items_gross_cents_agg") - F("expected_fee_cents_agg"),
Value(0),
output_field=IntegerField(),
),
)
# flags
qs = qs.annotate(
totals_mismatch=Case(
When(subtotal_cents=F("items_gross_cents_agg"), then=Value(False)),
default=Value(True),
output_field=BooleanField(),
),
ledger_mismatch=Case(
When(
```

```
marketplace_fee_cents_agg=F("expected_fee_cents_agg"),
seller_net_cents_agg=F("expected_net_cents_agg"),
then=Value(False),
),
default=Value(True),
output_field=BooleanField(),
),
)
# paid missing stripe ids (ignore FREE PI)
qs = qs.annotate(
paid_missing_stripe_ids=Case(
When(paid_at__isnull=True, then=Value(False)),
When(stripe_payment_intent_id__exact="FREE", then=Value(False)),
When(stripe_session_id__exact="", then=Value(True)),
When(stripe_payment_intent_id__exact="", then=Value(True)),
default=Value(False),
output_field=BooleanField(),
),
paid_missing_transfer_event=Case(
When(paid_at__isnull=True, then=Value(False)),
When(stripe_payment_intent_id__exact="FREE", then=Value(False)),
When(has_transfer_event=True, then=Value(False)),
# if skipped-unready, don't label as "missing transfer"; it's an explained state
When(payout_skipped_unready_seller=True, then=Value(False)),
default=Value(True),
output_field=BooleanField(),
),
)
```

```
return qs

# ----------------------

# display helpers

# ----------------------

@admin.display(description="buyer")

def buyer_display(self, obj: Order) -> str:

if obj.buyer_id:

return getattr(obj.buyer, "username", None) or getattr(obj.buyer, "email", "") or f"User<{obj.buyer_id}>"

return obj.guest_email or "—"

@admin.display(description="subtotal")

def subtotal_money(self, obj: Order) -> str:

return cents_to_money(obj.subtotal_cents, obj.currency)

@admin.display(description="total")

def total_money(self, obj: Order) -> str:

return cents_to_money(obj.total_cents, obj.currency)

@admin.display(description="items gross")

def items_gross_money(self, obj: Order) -> str:

return cents_to_money(int(getattr(obj, "items_gross_cents_agg", 0) or 0), obj.currency)

@admin.display(description="exp fee")

def expected_fee_money(self, obj: Order) -> str:

return cents_to_money(int(getattr(obj, "expected_fee_cents_agg", 0) or 0), obj.currency)

@admin.display(description="mkt fee (items)")

def marketplace_fee_money(self, obj: Order) -> str:

return cents_to_money(int(getattr(obj, "marketplace_fee_cents_agg", 0) or 0), obj.currency)

@admin.display(description="exp net")

def expected_net_money(self, obj: Order) -> str:

return cents_to_money(int(getattr(obj, "expected_net_cents_agg", 0) or 0), obj.currency)

@admin.display(description="seller net (items)")
```

```
def seller_net_money(self, obj: Order) -> str:
return cents_to_money(int(getattr(obj, "seller_net_cents_agg", 0) or 0), obj.currency)
@admin.display(description="qty")
def items_qty(self, obj: Order) -> int:
return int(getattr(obj, "items_qty_agg", 0) or 0)
@admin.display(description="sellers")
def seller_count(self, obj: Order) -> int:
return int(getattr(obj, "seller_count_agg", 0) or 0)
@admin.display(description="payout")
def payout_state_badge(self, obj: Order) -> str:
if not obj.paid_at:
return format_html("<span style='padding:2px 6px;border-radius:10px;background:#eee;'>UNPAID</span>")
if getattr(obj, "has_transfer_event", False):
return format_html("<span style='padding:2px 6px;border-radius:10px;background:#d1fae5;'>PAID OUT</span>")
if getattr(obj, "payout_skipped_unready_seller", False):
return format_html("<span style='padding:2px 6px;border-radius:10px;background:#fecaca;'>SKIPPED</span>")
return format_html("<span style='padding:2px 6px;border-radius:10px;background:#fde68a;'>PENDING</span>")
@admin.display(description="recon")
def recon_badge(self, obj: Order) -> str:
flags = []
if getattr(obj, "totals_mismatch", False):
flags.append("subtotal!=items")
if getattr(obj, "ledger_mismatch", False):
flags.append("fee/net!=expected")
if getattr(obj, "paid_missing_stripe_ids", False):
flags.append("paid missing stripe ids")
if getattr(obj, "paid_missing_transfer_event", False):
flags.append("paid missing transfer")
```

if getattr(obj, "payout_skipped_unready_seller", False):

flags.append("payout skipped (unready seller)")

if not flags:

return format_html("OK")

txt = "; ".join(flags)

return format_html("🚨 {}", txt)

----

payout summary (order change view)

@admin.display(description="Per-seller payout summary")

def payout_summary_html(self, obj: Order) -> str:

"""

Read-only operator view.

We intentionally compute this live, because it’s for diagnostics/admin.

(Order snapshots still protect the money math for payout creation.)

"""

items = list(

obj.items.select_related("seller").all()

)

if not items:

return "—"

group by seller

by_seller: dict[str, dict[str, Any]] = {}

for it in items:

sid = str(it.seller_id)

if sid not in by_seller:

label = getattr(it.seller, "username", "") or getattr(it.seller, "email", "") or sid

by_seller[sid] = {

"seller": it.seller,

"label": label,

"gross": 0,

"net": 0,

}

gross = int(it.quantity) * int(it.unit_price_cents)

by_seller[sid]["gross"] += max(0, gross)

by_seller[sid]["net"] += max(0, int(it.seller_net_cents or 0))

rows: list[_SellerPayoutRow] = []

for sid, d in by_seller.items():

seller = d["seller"]

acct = SellerStripeAccount.objects.filter(user=seller).first()

ready = bool(acct and acct.is_ready)

bal = int(get_seller_balance_cents(seller=seller) or 0)

payout = max(0, int(d["net"]) + bal)

rows.append(

_SellerPayoutRow(

seller_id=sid,

seller_label=str(d["label"]),

connect_ready=ready,

gross_cents=int(d["gross"]),

net_cents=int(d["net"]),

balance_cents=bal,

payout_cents=payout,

)

)

rows.sort(key=lambda r: r.seller_label.lower())

render

```
out = []
out.append("<div style='max-width:980px'>")
out.append("<table style='border-collapse:collapse;width:100%'>")
out.append(
"<thead><tr>"
"<th style='text-align:left;border-bottom:1px solid #ddd;padding:6px'>Seller</th>"
"<th style='text-align:left;border-bottom:1px solid #ddd;padding:6px'>Connect</th>"
"<th style='text-align:right;border-bottom:1px solid #ddd;padding:6px'>Gross</th>"
"<th style='text-align:right;border-bottom:1px solid #ddd;padding:6px'>Net</th>"
"<th style='text-align:right;border-bottom:1px solid #ddd;padding:6px'>Balance</th>"
"<th style='text-align:right;border-bottom:1px solid #ddd;padding:6px'>Payout (net+bal)</th>"
"</tr></thead>"
)
out.append("<tbody>")
for r in rows:
badge = (
"<span style='padding:2px 6px;border-radius:10px;background:#d1fae5'>READY</span>"
if r.connect_ready
else "<span style='padding:2px 6px;border-radius:10px;background:#fecaca'>NOT READY</span>"
)
out.append(
"<tr>"
f"<td style='padding:6px;border-bottom:1px solid #f2f2f2'>{r.seller_label}</td>"
f"<td style='padding:6px;border-bottom:1px solid #f2f2f2'>{badge}</td>"
f"<td style='padding:6px;border-bottom:1px solid #f2f2f2;text-align:right'>{cents_to_money(r.gross_cents, obj.currency)}</td>"
f"<td style='padding:6px;border-bottom:1px solid #f2f2f2;text-align:right'>{cents_to_money(r.net_cents, obj.currency)}</td>"
```

```
f"<td style='padding:6px;border-bottom:1px solid #f2f2f2;text-align:right'>{cents_to_money(r.balance_cents, obj.currency)}</td>"

f"<td style='padding:6px;border-bottom:1px solid #f2f2f2;text-align:right'><b>{cents_to_money(r.payout_cents, obj.currency)}</b></td>"

"</tr>"

)

out.append("</tbody></table>")

if obj.paid_at and not obj.events.filter(type=OrderEvent.Type.TRANSFER_CREATED).exists():

out.append(

"<div style='margin-top:8px'>"

"<b>Note:</b> Order is paid but no transfer_created event exists yet. "

"Use the admin action <i>Retry payout transfers</i> if appropriate."

"</div>"

)

out.append("</div>")

return format_html("".join(out))

# ---------------------

# admin actions

# ---------------------

@admin.action(description="Add reconciliation WARNING event (non-destructive)")

def add_reconciliation_warning_event(self, request, queryset):

created = 0

for o in queryset:

flags = []

if getattr(o, "totals_mismatch", False):

flags.append(

f"subtotal({o.subtotal_cents}) != items_gross({getattr(o, 'items_gross_cents_agg', 0)})"

)

if getattr(o, "ledger_mismatch", False):
```

```
flags.append(
f"items_fee({getattr(o, 'marketplace_fee_cents_agg', 0)})/items_net({getattr(o, 'seller_net_cents_agg', 0)}) "
f"!= expected_fee({getattr(o, 'expected_fee_cents_agg', 0)})/expected_net({getattr(o, 'expected_net_cents_agg', 0)})"
)
if getattr(o, "paid_missing_stripe_ids", False):
flags.append("paid but missing stripe_session_id or stripe_payment_intent_id")
if getattr(o, "paid_missing_transfer_event", False):
flags.append("paid but missing transfer_created event")
if getattr(o, "payout_skipped_unready_seller", False):
flags.append("payout skipped because at least one seller not ready")
if not flags:
continue
OrderEvent.objects.create(
order=o,
type=OrderEvent.Type.WARNING,
message="ADMIN RECON: " + " | ".join(flags),
)
created += 1
if created:
self.message_user(request, f"Created {created} WARNING event(s).", level=messages.WARNING)
else:
self.message_user(request, "No reconciliation issues found in selection.", level=messages.INFO)
@admin.action(description="Retry payout transfers (paid orders only)")
def retry_payout_transfers(self, request, queryset):
"""
Operator action:
- Only applies to PAID orders with a real Stripe PI
- Skips if transfer_created already exists
```

- Writes OrderEvent warnings on problems (via stripe_service)

"""

attempted = 0

succeeded = 0

skipped = 0

for o in queryset.select_related("buyer"):

if not o.paid_at or o.status != Order.Status.PAID:

skipped += 1

continue

pi = (o.stripe_payment_intent_id or "").strip()

if not pi or pi == "FREE":

skipped += 1

continue

if o.events.filter(type=OrderEvent.Type.TRANSFER_CREATED).exists():

skipped += 1

continue

attempted += 1

try:

create_transfers_for_paid_order(order=o, payment_intent_id=pi)

If a transfer was created, we'll have at least one event now.

if o.events.filter(type=OrderEvent.Type.TRANSFER_CREATED).exists():

succeeded += 1

except Exception:

Keep admin action resilient

OrderEvent.objects.create(

order=o,

type=OrderEvent.Type.WARNING,

message="ADMIN: retry payout transfers failed (see server logs).",

)

if attempted == 0:

self.message_user(request, "No eligible paid orders selected for payout retry.", level=messages.INFO)

return

self.message_user(

request,

f"Retry payouts: attempted={attempted}, succeeded={succeeded}, skipped={skipped}.",

level=messages.SUCCESS if succeeded else messages.WARNING,

)

========================

Order Item Admin

========================

class OrderPaidFilter(admin.SimpleListFilter):

title = "order paid"

parameter_name = "order_paid"

def lookups(self, request, model_admin):

return (("paid", "Paid"), ("unpaid", "Unpaid"))

def queryset(self, request, queryset):

val = self.value()

if val == "paid":

return queryset.filter(order__paid_at__isnull=False)

if val == "unpaid":

return queryset.filter(order__paid_at__isnull=True)

return queryset

class OrderPayoutFilter(admin.SimpleListFilter):

title = "order payout"

parameter_name = "order_payout"

def lookups(self, request, model_admin):

```
return (("pending", "Paid, payout pending"), ("paid_out", "Paid out (transfer created)"))
def queryset(self, request, queryset):
val = self.value()
if val in {"pending", "paid_out"}:
transfer_orders = OrderEvent.objects.filter(type=TRANSFER_EVENT_TYPE).values("order_id")
if val == "paid_out":
return queryset.filter(order_id__in=transfer_orders)
return queryset.exclude(order_id__in=transfer_orders).filter(order__paid_at__isnull=False)
return queryset
@admin.register(OrderItem)
class OrderItemAdmin(admin.ModelAdmin):
date_hierarchy = "created_at"
list_select_related = ("order", "product", "seller")
list_display = (
"id",
"order_link",
"order_status",
"order_paid_at",
"product",
"seller",
"quantity",
"unit_price_money",
"marketplace_fee_money",
"seller_net_money",
"is_digital",
"requires_shipping",
"created_at",
)
```

```
list_filter = (
"is_digital",
"requires_shipping",
"created_at",
"seller",
"order__status",
OrderPaidFilter,
OrderPayoutFilter,
)
search_fields = (
"id",
"order__id",
"product__title",
"seller__username",
"seller__email",
"order__guest_email",
)
readonly_fields = ("id", "created_at")
raw_id_fields = ("order", "product", "seller")
@admin.display(description="order", ordering="order__id")
def order_link(self, obj: OrderItem):
url = admin_order_change_url(obj.order_id)
return format_html("<a href='{}'>#{}</a>", url, obj.order_id)
@admin.display(description="order status", ordering="order__status")
def order_status(self, obj: OrderItem) -> str:
return str(obj.order.status)
@admin.display(description="paid at", ordering="order__paid_at")
def order_paid_at(self, obj: OrderItem):
```

```
return obj.order.paid_at or "—"
@admin.display(description="unit price")
def unit_price_money(self, obj: OrderItem) -> str:
return cents_to_money(obj.unit_price_cents, obj.order.currency)
@admin.display(description="mkt fee")
def marketplace_fee_money(self, obj: OrderItem) -> str:
return cents_to_money(obj.marketplace_fee_cents, obj.order.currency)
@admin.display(description="seller net")
def seller_net_money(self, obj: OrderItem) -> str:
return cents_to_money(obj.seller_net_cents, obj.order.currency)
# =========================
# Order Event Admin
# =========================
@admin.register(OrderEvent)
class OrderEventAdmin(admin.ModelAdmin):
date_hierarchy = "created_at"
list_display = ("created_at", "order_link", "type", "message")
list_filter = ("type", "created_at")
search_fields = ("order__id", "message", "type")
readonly_fields = ("id", "created_at", "order", "type", "message")
list_select_related = ("order",)
@admin.display(description="order", ordering="order__id")
def order_link(self, obj: OrderEvent):
url = admin_order_change_url(obj.order_id)
return format_html("<a href='{}'>#{}</a>", url, obj.order_id)
# =========================
# Stripe Webhook Event Admin (idempotency audit)
# =========================
```

@admin.register(StripeWebhookEvent)

class StripeWebhookEventAdmin(admin.ModelAdmin):

date_hierarchy = "created_at"

list_display = ("created_at", "event_type", "stripe_event_id")

list_filter = ("event_type", "created_at")

search_fields = ("stripe_event_id", "event_type")

readonly_fields = ("id", "stripe_event_id", "event_type", "created_at")

ordering = ("-created_at",)

orders.models

orders/models.py

from __future__ import annotations

import uuid

from decimal import Decimal

from typing import Optional

from django.conf import settings

from django.core.exceptions import ValidationError

from django.core.mail import send_mail

from django.db import models

from django.urls import reverse

from django.utils import timezone

def _site_base_url() -> str:

"""

Absolute base URL used in emails for guest access/download links.

Set SITE_BASE_URL in env (recommended), e.g. https://homecraft3d.com

"""

base = (getattr(settings, "SITE_BASE_URL", "") or "").strip().rstrip("/")

if base:

return base

return "http://localhost:8000"

def _send_guest_paid_email_with_downloads(order: "Order") -> None:

"""

Send guest email containing:

- order detail link (tokenized)

- digital asset download links (tokenized), if any

"""

if not order.guest_email:

return

base = _site_base_url()

order_link = f"{base}{reverse('orders:detail', kwargs={'order_id': order.pk})}?t={order.order_token}"

product_ids = list(order.items.values_list("product_id", flat=True))

if not product_ids:

assets = []

else:

from products.models import DigitalAsset # noqa

assets = list(

DigitalAsset.objects.filter(product_id__in=product_ids)

.select_related("product")

.order_by("product_id", "id")

)

lines: list[str] = []

lines.append("Thanks for your purchase at Home Craft 3D!")

lines.append("")

lines.append("Access your order here:")

lines.append(order_link)

lines.append("")

if assets:

```
lines.append("Your digital downloads (links are tied to your order):")
for a in assets:
try:
if a.product.kind != a.product.Kind.FILE:
continue
except Exception:
continue
fn = a.original_filename or (a.file.name.rsplit("/", 1)[-1] if a.file else "download")
dl = (
f"{base}"
f"{reverse('orders:download_asset', kwargs={'order_id': order.pk, 'asset_id': a.pk})}"
f"?t={order.order_token}"
)
lines.append(f"- {fn}: {dl}")
lines.append("")
lines.append("If you didn’t make this purchase, you can ignore this email.")
subject = f"Your Home Craft 3D order #{order.pk}"
body = "\n".join(lines)
try:
send_mail(
subject,
body,
getattr(settings, "DEFAULT_FROM_EMAIL", None),
[order.guest_email],
)
except Exception:
pass
class Order(models.Model):
```

class Status(models.TextChoices):

DRAFT = "draft", "Draft"

PENDING = "pending", "Pending"

PAID = "paid", "Paid"

CANCELED = "canceled", "Canceled"

REFUNDED = "refunded", "Refunded"

class Kind(models.TextChoices):

DIGITAL = "digital", "Digital only"

PHYSICAL = "physical", "Physical only"

MIXED = "mixed", "Mixed (digital + physical)"

id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

buyer = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.PROTECT,

null=True,

blank=True,

related_name="orders",

help_text="Registered buyer. Null means guest checkout.",

)

guest_email = models.EmailField(blank=True, default="")

order_token = models.UUIDField(default=uuid.uuid4, editable=False, db_index=True)

status = models.CharField(max_length=24, choices=Status.choices, default=Status.DRAFT)

kind = models.CharField(max_length=16, choices=Kind.choices, default=Kind.PHYSICAL)

currency = models.CharField(max_length=8, default="usd")

subtotal_cents = models.PositiveIntegerField(default=0)

tax_cents = models.PositiveIntegerField(default=0)

shipping_cents = models.PositiveIntegerField(default=0)

total_cents = models.PositiveIntegerField(default=0)

Snapshot settings (historical correctness)

marketplace_sales_percent_snapshot = models.DecimalField(

max_digits=5,

decimal_places=2,

default=Decimal("0.00"),

help_text="Marketplace % cut captured at order creation time.",

)

Legacy field: kept for compatibility, but MUST be 0 (platform fee not used).

platform_fee_cents_snapshot = models.PositiveIntegerField(

default=0,

help_text="Legacy flat fee snapshot (NOT USED). Keep at 0.",

)

stripe_session_id = models.CharField(max_length=255, blank=True, default="", db_index=True)

stripe_payment_intent_id = models.CharField(max_length=255, blank=True, default="")

paid_at = models.DateTimeField(null=True, blank=True, db_index=True)

shipping_name = models.CharField(max_length=255, blank=True, default="")

shipping_phone = models.CharField(max_length=64, blank=True, default="")

shipping_line1 = models.CharField(max_length=255, blank=True, default="")

shipping_line2 = models.CharField(max_length=255, blank=True, default="")

shipping_city = models.CharField(max_length=120, blank=True, default="")

shipping_state = models.CharField(max_length=120, blank=True, default="")

shipping_postal_code = models.CharField(max_length=32, blank=True, default="")

shipping_country = models.CharField(max_length=2, blank=True, default="")

created_at = models.DateTimeField(default=timezone.now, db_index=True)

updated_at = models.DateTimeField(auto_now=True)

class Meta:

indexes = [

models.Index(fields=["-created_at"]),

```
models.Index(fields=["status", "-created_at"]),
models.Index(fields=["buyer", "-created_at"]),
models.Index(fields=["kind", "-created_at"]),
models.Index(fields=["-paid_at"]),
]
def __str__(self) -> str:
return f"Order {self.pk} ({self.status})"
@property
def access_token(self) -> uuid.UUID:
return self.order_token
def ensure_access_token(self) -> None:
if not self.order_token:
self.order_token = uuid.uuid4()
self.save(update_fields=["order_token", "updated_at"])
def clean(self) -> None:
has_buyer = bool(self.buyer_id)
has_guest = bool((self.guest_email or "").strip())
if not has_buyer and not has_guest:
raise ValidationError("Order must have either a buyer or a guest_email.")
if has_buyer and has_guest:
self.guest_email = ""
super().clean()
@property
def is_guest(self) -> bool:
return self.buyer_id is None
@property
def requires_shipping(self) -> bool:
return self.items.filter(requires_shipping=True).exists()
```

```
def recompute_totals(self) -> None:
subtotal = 0
any_digital = False
any_physical = False
for oi in self.items.all():
subtotal += int(oi.line_total_cents)
if oi.is_digital:
any_digital = True
if oi.requires_shipping:
any_physical = True
self.subtotal_cents = int(subtotal)
self.total_cents = int(self.subtotal_cents + self.tax_cents + self.shipping_cents)
if any_digital and any_physical:
self.kind = self.Kind.MIXED
elif any_digital:
self.kind = self.Kind.DIGITAL
else:
self.kind = self.Kind.PHYSICAL
def set_shipping_from_stripe(
self,
*args,
name: str = "",
phone: str = "",
line1: str = "",
line2: str = "",
city: str = "",
state: str = "",
postal_code: str = "",
```

country: str = "",

) -> None:

self.shipping_name = name or ""

self.shipping_phone = phone or ""

self.shipping_line1 = line1 or ""

self.shipping_line2 = line2 or ""

self.shipping_city = city or ""

self.shipping_state = state or ""

self.shipping_postal_code = postal_code or ""

self.shipping_country = country or ""

self.save(

update_fields=[

"shipping_name",

"shipping_phone",

"shipping_line1",

"shipping_line2",

"shipping_city",

"shipping_state",

"shipping_postal_code",

"shipping_country",

"updated_at",

]

)

def _add_event(self, type_: str, message: str = "") -> None:

try:

OrderEvent.objects.create(order=self, type=type_, message=message or "")

except Exception:

pass

```
def mark_paid(
self,
*,
payment_intent_id: str = "",
session_id: str = "",
paid_at: Optional[timezone.datetime] = None,
note: str = "",
) -> bool:
changed = False
now = paid_at or timezone.now()
update_fields: list[str] = []
payment_intent_id = (payment_intent_id or "").strip()
session_id = (session_id or "").strip()
if session_id and not self.stripe_session_id:
self.stripe_session_id = session_id
update_fields.append("stripe_session_id")
if payment_intent_id and not self.stripe_payment_intent_id:
self.stripe_payment_intent_id = payment_intent_id
update_fields.append("stripe_payment_intent_id")
if self.status != self.Status.PAID:
self.status = self.Status.PAID
update_fields.append("status")
changed = True
if not self.paid_at:
self.paid_at = now
update_fields.append("paid_at")
changed = True
if update_fields:
```

update_fields.append("updated_at")

self.save(update_fields=update_fields)

if changed:

msg = note or ""

if payment_intent_id and payment_intent_id != "FREE":

msg = msg or f"Marked paid via Stripe PI {payment_intent_id}"

elif payment_intent_id == "FREE":

msg = msg or "Marked paid via FREE checkout"

self._add_event(OrderEvent.Type.PAID, msg)

if self.is_guest and (self.guest_email or "").strip():

_send_guest_paid_email_with_downloads(self)

return changed

class OrderItem(models.Model):

id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

order = models.ForeignKey(Order, on_delete=models.CASCADE, related_name="items")

product = models.ForeignKey(

"products.Product",

on_delete=models.PROTECT,

related_name="order_items",

)

seller = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.PROTECT,

related_name="sold_order_items",

help_text="Seller snapshot at time of purchase.",

)

quantity = models.PositiveIntegerField(default=1)

unit_price_cents = models.PositiveIntegerField(default=0)

is_digital = models.BooleanField(default=False)

requires_shipping = models.BooleanField(default=True)

marketplace_fee_cents = models.PositiveIntegerField(

default=0,

help_text="Marketplace fee on this line (percent-based).",

)

seller_net_cents = models.PositiveIntegerField(

default=0,

help_text="Seller net on this line (gross - marketplace_fee).",

)

created_at = models.DateTimeField(default=timezone.now)

class Meta:

indexes = [

models.Index(fields=["order", "created_at"]),

models.Index(fields=["product", "created_at"]),

models.Index(fields=["seller", "created_at"]),

]

def __str__(self) -> str:

return f"{self.quantity} × {self.product_id}"

@property

def line_total_cents(self) -> int:

return int(self.quantity) * int(self.unit_price_cents)

LineItem = OrderItem

class OrderEvent(models.Model):

class Type(models.TextChoices):

CREATED = "created", "Created"

STRIPE_SESSION_CREATED = "stripe_session_created", "Stripe session created"

PAID = "paid", "Paid"

CANCELED = "canceled", "Canceled"

REFUNDED = "refunded", "Refunded"

TRANSFER_CREATED = "transfer_created", "Transfer created"

WARNING = "warning", "Warning"

id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

order = models.ForeignKey(Order, on_delete=models.CASCADE, related_name="events")

type = models.CharField(max_length=64, choices=Type.choices)

message = models.TextField(blank=True, default="")

created_at = models.DateTimeField(default=timezone.now)

class Meta:

indexes = [models.Index(fields=["order", "-created_at"])]

def __str__(self) -> str:

return f"{self.type} ({self.created_at:%Y-%m-%d %H:%M})"

class StripeWebhookEvent(models.Model):

"""

Records processed Stripe webhook events for strict idempotency.

"""

id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

stripe_event_id = models.CharField(max_length=255, unique=True)

event_type = models.CharField(max_length=255, blank=True, default="")

created_at = models.DateTimeField(default=timezone.now)

class Meta:

indexes = [

models.Index(fields=["stripe_event_id"]),

models.Index(fields=["event_type", "-created_at"]),

]

def __str__(self) -> str:

return f"{self.event_type} ({self.stripe_event_id})"

orders.refunds

orders/refunds.py

from __future__ import annotations

from typing import Iterable

from django.db import transaction

from orders.models import Order, OrderItem, OrderEvent

from payments.models import SellerBalanceEntry

@transaction.atomic

def refund_order_items(

*,

order: Order,

items: Iterable[OrderItem],

reason: str,

refund_marketplace_fee: bool = False,

) -> None:

"""

Refunds specific order items.

Assumptions:

- Stripe refund is handled upstream (this function is ledger + audit only)

- Items belong to the given order

"""

if order.status not in {Order.Status.PAID, Order.Status.REFUNDED}:

raise ValueError("Cannot refund an unpaid order.")

items_list = list(items)

for item in items_list:

if item.order_id != order.id:

raise ValueError("OrderItem does not belong to this order.")

SellerBalanceEntry.objects.create(

seller=item.seller,

amount_cents=-int(item.seller_net_cents),

reason=SellerBalanceEntry.Reason.REFUND,

order=order,

order_item=item,

note=reason,

)

if refund_marketplace_fee and int(item.marketplace_fee_cents or 0) > 0:

SellerBalanceEntry.objects.create(

seller=item.seller,

amount_cents=int(item.marketplace_fee_cents),

reason=SellerBalanceEntry.Reason.ADJUSTMENT,

order=order,

order_item=item,

note="Marketplace fee refunded",

)

OrderEvent.objects.create(

order=order,

type=OrderEvent.Type.REFUNDED,

message=reason,

)

if len(items_list) == order.items.count():

order.status = Order.Status.REFUNDED

order.save(update_fields=["status", "updated_at"])

orders.services

orders/services.py

from __future__ import annotations

from dataclasses import dataclass

from decimal import Decimal, ROUND_HALF_UP

from typing import Iterable, Optional

from django.db import transaction

from core.config import get_site_config

from payments.utils import money_to_cents

from products.models import Product

from .models import LineItem, Order, OrderEvent

@dataclass(frozen=True)

class ShippingSnapshot:

name: str = ""

phone: str = ""

line1: str = ""

line2: str = ""

city: str = ""

state: str = ""

postal_code: str = ""

country: str = ""

def normalize_email(email: str) -> str:

return (email or "").strip().lower()

def _iter_cart_items(cart_or_items) -> Iterable:

if hasattr(cart_or_items, "lines") and callable(getattr(cart_or_items, "lines")):

return cart_or_items.lines()

return cart_or_items

def _cents_round(d: Decimal) -> int:

return int(d.quantize(Decimal("1"), rounding=ROUND_HALF_UP))

def _pct_to_rate(pct: Decimal) -> Decimal:

try:

return Decimal(pct) / Decimal("100")

except Exception:

return Decimal("0")

def _compute_marketplace_fee_cents(*, gross_cents: int, sales_rate: Decimal) -> int:

gross = Decimal(int(gross_cents))

fee = gross * (sales_rate or Decimal("0"))

return max(0, _cents_round(fee))

@transaction.atomic

def create_order_from_cart(

cart_or_items=None,

*,

cart_items=None,

buyer,

guest_email: str,

currency: str = "usd",

shipping: Optional[ShippingSnapshot] = None,

) -> Order:

"""

Create an Order + OrderItems from a session Cart (or an iterable of cart lines).

Rules:

- Snapshot SiteConfig percent onto Order at creation time

- Snapshot seller onto each OrderItem at purchase time

- Compute per-line ledger fields:

marketplace_fee_cents, seller_net_cents

Platform fee:

- NOT USED. Forced to 0 (legacy field remains).

"""

items_iterable = cart_items if cart_items is not None else cart_or_items

buyer_obj = buyer if getattr(buyer, "is_authenticated", False) else None

guest_email = normalize_email(guest_email)

if buyer_obj is None and not guest_email:

raise ValueError("Guest checkout requires a valid email address.")

cfg = get_site_config()

sales_pct = Decimal(getattr(cfg, "marketplace_sales_percent", Decimal("0.00")) or Decimal("0.00"))

order = Order.objects.create(

buyer=buyer_obj,

guest_email=guest_email if buyer_obj is None else "",

currency=(currency or "usd").lower(),

status=Order.Status.PENDING,

marketplace_sales_percent_snapshot=sales_pct,

platform_fee_cents_snapshot=0, # legacy: no platform fee

)

if shipping:

order.shipping_name = shipping.name

order.shipping_phone = shipping.phone

order.shipping_line1 = shipping.line1

order.shipping_line2 = shipping.line2

order.shipping_city = shipping.city

order.shipping_state = shipping.state

order.shipping_postal_code = shipping.postal_code

order.shipping_country = shipping.country

items = _iter_cart_items(items_iterable)

sales_rate = _pct_to_rate(order.marketplace_sales_percent_snapshot)

line_items: list[LineItem] = []

for item in items:

product = getattr(item, "product", None)

if product is None:

raise ValueError("Cart line missing product.")

seller = getattr(product, "seller", None)

if seller is None:

raise ValueError(f"Product {getattr(product, 'pk', '')} has no seller.")

qty = int(getattr(item, "quantity", 1) or 1)

if hasattr(item, "unit_price_cents"):

unit_price_cents = int(getattr(item, "unit_price_cents") or 0)

else:

unit_price = getattr(item, "unit_price", None)

unit_price_cents = money_to_cents(unit_price)

is_digital = bool(getattr(product, "kind", "") == Product.Kind.FILE)

requires_shipping = bool(getattr(product, "kind", "") == Product.Kind.MODEL)

if is_digital:

qty = 1

gross_cents = max(0, int(qty) * int(unit_price_cents))

marketplace_fee_cents = _compute_marketplace_fee_cents(gross_cents=gross_cents, sales_rate=sales_rate)

seller_net_cents = max(0, gross_cents - marketplace_fee_cents)

line_items.append(

LineItem(

order=order,

product=product,

seller=seller,

quantity=qty,

unit_price_cents=unit_price_cents,

is_digital=is_digital,

requires_shipping=requires_shipping,

marketplace_fee_cents=marketplace_fee_cents,

seller_net_cents=seller_net_cents,

)

)

LineItem.objects.bulk_create(line_items)

order.recompute_totals()

order.save(

update_fields=[

"subtotal_cents",

"tax_cents",

"shipping_cents",

"total_cents",

"kind",

"shipping_name",

"shipping_phone",

"shipping_line1",

"shipping_line2",

"shipping_city",

"shipping_state",

"shipping_postal_code",

"shipping_country",

"status",

"updated_at",

]

)

try:

OrderEvent.objects.create(order=order, type=OrderEvent.Type.CREATED)

except Exception:

pass

return order

@transaction.atomic

def mark_order_paid(*, order: Order, stripe_payment_intent_id: str = "") -> Order:

"""Legacy helper (prefer Order.mark_paid)."""

order.mark_paid(payment_intent_id=stripe_payment_intent_id)

return order

orders.stripe_service

orders/stripe_service.py

from __future__ import annotations

import hashlib

from typing import Any

import stripe

from django.conf import settings

from django.db import transaction

from django.db.models import F, IntegerField, Sum

from django.db.models.expressions import ExpressionWrapper

from django.urls import reverse

from core.config import get_allowed_shipping_countries

from payments.models import SellerStripeAccount, SellerBalanceEntry

from payments.services import get_seller_balance_cents

from payments.utils import seller_is_stripe_ready

from products.permissions import is_owner_user

from .models import Order, OrderEvent

def _stripe_init() -> None:

stripe.api_key = settings.STRIPE_SECRET_KEY

def _assert_order_sellers_stripe_ready(*, request, order: Order) -> None:

"""

Defense-in-depth: re-check readiness at checkout start.

IMPORTANT: use OrderItem.seller snapshot (not product__seller).

"""

if request.user.is_authenticated and is_owner_user(request.user):

return

bad: list[str] = []

for it in order.items.select_related("seller").all():

seller = it.seller

if not seller or not seller_is_stripe_ready(seller):

bad.append(getattr(seller, "username", str(getattr(seller, "pk", ""))))

if bad:

seen: set[str] = set()

bad_unique = [u for u in bad if not (u in seen or seen.add(u))]

raise ValueError("Seller not ready for checkout: " + ", ".join(bad_unique))

def _order_idem_key(order: Order) -> str:

base = f"{order.pk}:{order.total_cents}:{order.currency}:{order.kind}"

digest = hashlib.sha256(base.encode("utf-8")).hexdigest()[:12]

return f"checkout_session:{order.pk}:{digest}:v1"

def create_checkout_session_for_order(*, request, order: Order) -> stripe.checkout.Session:

_stripe_init()

_assert_order_sellers_stripe_ready(request=request, order=order)

line_items: list[dict[str, Any]] = []

for item in order.items.select_related("product").all():

product = item.product

name = getattr(product, "title", None) or getattr(product, "name", None) or f"Product {item.product_id}"

line_items.append(

{

"price_data": {

"currency": order.currency.lower(),

"product_data": {"name": str(name)},

"unit_amount": int(item.unit_price_cents),

},

"quantity": int(item.quantity),

}

)

success_url = request.build_absolute_uri(

reverse("orders:checkout_success") + "?session_id={CHECKOUT_SESSION_ID}"

)

cancel_path = reverse("orders:checkout_cancel", kwargs={"order_id": order.pk})

if order.is_guest:

cancel_path = f"{cancel_path}?t={order.order_token}"

cancel_url = request.build_absolute_uri(cancel_path)

kwargs: dict[str, Any] = {

"mode": "payment",

"line_items": line_items,

"success_url": success_url,

"cancel_url": cancel_url,

"client_reference_id": str(order.pk),

"metadata": {

"order_id": str(order.pk),

"order_token": str(order.order_token),

"kind": str(order.kind),

},

"payment_intent_data": {

"metadata": {

"order_id": str(order.pk),

"order_token": str(order.order_token),

}

},

"billing_address_collection": "required",

}

if order.is_guest and order.guest_email:

kwargs["customer_email"] = order.guest_email

if order.requires_shipping:

kwargs["shipping_address_collection"] = {"allowed_countries": get_allowed_shipping_countries()}

kwargs["phone_number_collection"] = {"enabled": True}

session = stripe.checkout.Session.create(

**kwargs,

idempotency_key=_order_idem_key(order),

)

if not order.stripe_session_id:

order.stripe_session_id = session.id

order.save(update_fields=["stripe_session_id", "updated_at"])

OrderEvent.objects.create(

order=order,

type=OrderEvent.Type.STRIPE_SESSION_CREATED,

message=f"session={session.id}",

)

return session

def verify_and_parse_webhook(payload: bytes, sig_header: str):

_stripe_init()

return stripe.Webhook.construct_event(

payload=payload,

sig_header=sig_header,

secret=settings.STRIPE_WEBHOOK_SECRET,

)

```
def _transfers_already_recorded(order: Order) -> bool:
return order.events.filter(type=OrderEvent.Type.TRANSFER_CREATED).exists()
@transaction.atomic
def create_transfers_for_paid_order(*, order: Order, payment_intent_id: str) -> None:
"""
Create Stripe transfers for a PAID order.
Ledger-aware:
- Applies seller balance
- Never overpays
- Carries negative balances forward
"""
payment_intent_id = (payment_intent_id or "").strip()
if not payment_intent_id or payment_intent_id == "FREE":
return
if _transfers_already_recorded(order):
return
_stripe_init()
line_total_expr = ExpressionWrapper(
F("quantity") * F("unit_price_cents"),
output_field=IntegerField(),
)
seller_rows = (
order.items.select_related("seller")
.values("seller_id")
.annotate(
gross_cents=Sum(line_total_expr),
net_cents=Sum("seller_net_cents"),
)
```

)

charge_id = ""

try:

pi = stripe.PaymentIntent.retrieve(payment_intent_id)

charge_id = str(getattr(pi, "latest_charge", "") or "")

except Exception:

charge_id = ""

for row in seller_rows:

seller_id = row.get("seller_id")

gross_cents = int(row.get("gross_cents") or 0)

net_cents = int(row.get("net_cents") or 0)

if gross_cents <= 0 or net_cents <= 0 or not seller_id:

continue

acct = SellerStripeAccount.objects.filter(user_id=seller_id).first()

if not acct or not acct.is_ready:

OrderEvent.objects.create(

order=order,

type=OrderEvent.Type.WARNING,

message=f"transfer skipped seller={seller_id} (not ready)",

)

continue

balance_cents = int(get_seller_balance_cents(seller=acct.user) or 0)

payout_cents = max(0, net_cents + balance_cents)

if payout_cents <= 0:

OrderEvent.objects.create(

order=order,

type=OrderEvent.Type.WARNING,

message=f"transfer skipped seller={seller_id} (balance={balance_cents})",

)

continue

kwargs: dict[str, Any] = {

"amount": int(payout_cents),

"currency": order.currency.lower(),

"destination": acct.stripe_account_id,

"transfer_group": str(order.pk),

"metadata": {

"order_id": str(order.pk),

"seller_id": str(seller_id),

"payment_intent_id": str(payment_intent_id),

"gross_cents": str(gross_cents),

"net_cents": str(net_cents),

"seller_balance_before": str(balance_cents),

},

}

if charge_id:

kwargs["source_transaction"] = charge_id

transfer = stripe.Transfer.create(

**kwargs,

idempotency_key=f"transfer:{order.pk}:{seller_id}:v4",

)

SellerBalanceEntry.objects.create(

seller=acct.user,

amount_cents=-int(payout_cents),

reason=SellerBalanceEntry.Reason.PAYOUT,

order=order,

note=f"Stripe transfer {transfer.id}",

```
)
OrderEvent.objects.create(
order=order,
type=OrderEvent.Type.TRANSFER_CREATED,
message=(
f"transfer={transfer.id} seller={seller_id} "
f"gross={gross_cents} net={net_cents} "
f"balance_before={balance_cents} payout={payout_cents}"
),
)
orders.urls
# orders/urls.py
from __future__ import annotations
from django.urls import include, path
from . import views
from . import webhooks
app_name = "orders"
urlpatterns = [
# Buyer checkout flow
path("place/", views.place_order, name="place"),
path("<uuid:order_id>/", views.order_detail, name="detail"),
path("<uuid:order_id>/checkout/start/", views.checkout_start, name="checkout_start"),
path("checkout/success/", views.checkout_success, name="checkout_success"),
path("<uuid:order_id>/checkout/cancel/", views.checkout_cancel, name="checkout_cancel"),
path("<uuid:order_id>/download/<uuid:asset_id>/", views.download_asset, name="download_asset"),
# Buyer: Purchases (paid-only downloads live here)
path("purchases/", views.purchases, name="purchases"),
# Buyer order history (legacy/all)
```

path("mine/", views.my_orders, name="my_orders"),

Seller fulfillment (current implementation is in orders/views.py)

path("seller/orders/", views.seller_orders_list, name="seller_orders_list"),

path("seller/orders/<uuid:order_id>/", views.seller_order_detail, name="seller_order_detail"),

Refunds (under Orders)

path("refunds/", include(("refunds.urls", "refunds"), namespace="refunds")),

Stripe webhook endpoint (cleanly separated)

path("webhooks/stripe/", webhooks.stripe_webhook, name="stripe_webhook"),

]

orders.views

orders/views.py

from __future__ import annotations

import logging

from django.conf import settings

from django.contrib import messages

from django.contrib.auth.decorators import login_required

from django.core.exceptions import ValidationError

from django.core.paginator import Paginator

from django.core.validators import validate_email

from django.http import FileResponse, Http404

from django.shortcuts import get_object_or_404, redirect, render

from django.urls import reverse

from django.views.decorators.http import require_POST

from cart.cart import Cart

from payments.utils import seller_is_stripe_ready

from products.models import DigitalAsset, Product

from products.permissions import is_owner_user, is_seller_user

from .models import Order

from .services import create_order_from_cart

from .stripe_service import create_checkout_session_for_order

logger = logging.getLogger(__name__)

def _token_from_request(request) -> str:

return (request.GET.get("t") or "").strip()

def _normalize_guest_email(raw: str) -> str:

email = (raw or "").strip().lower()

if not email:

return ""

try:

validate_email(email)

except ValidationError:

return ""

return email

def _user_can_access_order(request, order: Order) -> bool:

if request.user.is_authenticated and (request.user.is_staff or request.user.is_superuser):

return True

if getattr(order, "buyer_id", None):

return request.user.is_authenticated and request.user.id == order.buyer_id

t = _token_from_request(request)

return bool(t) and str(t) == str(getattr(order, "order_token", ""))

def _order_has_unready_sellers(order: Order) -> list[str]:

bad: list[str] = []

for item in order.items.select_related("seller").all():

seller = getattr(item, "seller", None)

if seller and not seller_is_stripe_ready(seller):

bad.append(getattr(seller, "username", str(getattr(seller, "pk", ""))))

seen: set[str] = set()

```
    out: list[str] = []
    for u in bad:
        if u in seen:
            continue
        seen.add(u)
        out.append(u)
    return out


def _cart_has_unready_sellers(cart: Cart) -> list[str]:
    bad: list[str] = []
    for line in cart.lines():
        product = getattr(line, "product", None)
        seller = getattr(product, "seller", None)
        if seller and not seller_is_stripe_ready(seller):
            bad.append(getattr(seller, "username", str(getattr(seller, "pk", ""))))
    seen: set[str] = set()
    out: list[str] = []
    for u in bad:
        if u in seen:
            continue
        seen.add(u)
        out.append(u)
    return out


@require_POST
def place_order(request):
    cart = Cart(request)
    if cart.count_items() == 0:
        messages.info(request, "Your cart is empty.")
        return redirect("cart:detail")
```

```
bad_sellers = _cart_has_unready_sellers(cart)
if bad_sellers:
messages.error(
request,
"One or more sellers in your cart haven’t completed payout setup yet: " + ", ".join(bad_sellers),
)
return redirect("cart:detail")
guest_email = ""
if not request.user.is_authenticated:
guest_email = _normalize_guest_email(request.POST.get("guest_email") or "")
if not guest_email:
messages.error(request, "Please enter a valid email to checkout as a guest.")
return redirect("cart:detail")
try:
order = create_order_from_cart(cart, buyer=request.user, guest_email=guest_email)
except ValueError as e:
messages.error(request, str(e) or "Your cart can’t be checked out right now.")
return redirect("cart:detail")
cart.clear()
messages.success(request, "Order created (pending payment).")
if order.is_guest:
return redirect(f"{reverse('orders:detail', kwargs={'order_id': order.pk})}?t={order.order_token}")
return redirect("orders:detail", order_id=order.pk)
def order_detail(request, order_id):
order = get_object_or_404(
Order.objects.prefetch_related(
"items",
"items__seller",
```

"items__refund_request",

"items__product",

"items__product__digital_assets",

),

pk=order_id,

)

if not _user_can_access_order(request, order):

if order.buyer_id and not request.user.is_authenticated:

return redirect("accounts:login")

raise Http404("Not found")

can_download = bool(order.status == Order.Status.PAID and _user_can_access_order(request, order))

return render(

request,

"orders/order_detail.html",

{

"order": order,

"order_token": _token_from_request(request),

"can_download": can_download,

"stripe_publishable_key": getattr(settings, "STRIPE_PUBLISHABLE_KEY", ""),

},

)

@require_POST

def checkout_start(request, order_id):

order = get_object_or_404(

Order.objects.prefetch_related("items", "items__seller", "items__product"),

pk=order_id,

)

if not _user_can_access_order(request, order):

```
if order.buyer_id and not request.user.is_authenticated:
return redirect("accounts:login")
raise Http404("Not found")
if order.status != Order.Status.PENDING:
messages.info(request, "This order is not payable.")
return redirect("orders:detail", order_id=order.pk)
if order.items.count() == 0:
messages.error(request, "Order has no items.")
return redirect("orders:detail", order_id=order.pk)
bad_sellers = _order_has_unready_sellers(order)
if bad_sellers and not (request.user.is_authenticated and is_owner_user(request.user)):
messages.error(
request,
"One or more sellers in this order haven’t completed payout setup yet: " + ", ".join(bad_sellers),
)
return redirect("orders:detail", order_id=order.pk)
if int(order.total_cents or 0) <= 0:
order.mark_paid(payment_intent_id="FREE")
messages.success(request, "Your order is complete.")
if order.is_guest:
return redirect(f"{reverse('orders:detail', kwargs={'order_id': order.pk})}?t={order.order_token}")
return redirect("orders:detail", order_id=order.pk)
session = create_checkout_session_for_order(request=request, order=order)
return redirect(session.url)
def checkout_success(request):
session_id = (request.GET.get("session_id") or "").strip()
if not session_id:
messages.info(request, "Checkout completed. If your order doesn't update immediately, refresh in a moment.")
```

```
return redirect("home")
order = Order.objects.filter(stripe_session_id=session_id).first()
order_detail_url = ""
if order:
if order.is_guest:
order_detail_url = f"{reverse('orders:detail', kwargs={'order_id': order.pk})}?t={order.order_token}"
else:
order_detail_url = reverse("orders:detail", kwargs={"order_id": order.pk})
return render(request, "orders/checkout_success.html", {"order": order, "order_detail_url": order_detail_url})
def checkout_cancel(request, order_id):
order = get_object_or_404(Order, pk=order_id)
messages.info(request, "Checkout canceled.")
t = _token_from_request(request)
if order.is_guest and t:
return redirect(f"{reverse('orders:detail', kwargs={'order_id': order.pk})}?t={t}")
return redirect("orders:detail", order_id=order.pk)
def download_asset(request, order_id, asset_id):
order = get_object_or_404(
Order.objects.prefetch_related("items", "items__product"),
pk=order_id,
)
if order.status != Order.Status.PAID:
raise Http404("Not found")
if not _user_can_access_order(request, order):
if order.buyer_id and not request.user.is_authenticated:
return redirect("accounts:login")
raise Http404("Not found")
asset = get_object_or_404(DigitalAsset.objects.select_related("product"), pk=asset_id)
```

```
order_product_ids = set(order.items.values_list("product_id", flat=True))
if asset.product_id not in order_product_ids:
raise Http404("Not found")
if asset.product.kind != Product.Kind.FILE:
raise Http404("Not found")
file_handle = asset.file.open("rb")
filename = asset.original_filename or asset.file.name.rsplit("/", 1)[-1]
return FileResponse(file_handle, as_attachment=True, filename=filename)
@login_required
def purchases(request):
qs = (
Order.objects.filter(buyer=request.user, status=Order.Status.PAID, paid_at__isnull=False)
.prefetch_related("items", "items__product", "items__product__digital_assets")
.order_by("-paid_at", "-created_at")
)
paginator = Paginator(qs, 20)
page = paginator.get_page(request.GET.get("page") or 1)
return render(
request,
"orders/purchases.html",
{
"page_obj": page,
"orders": page.object_list,
},
)
@login_required
def my_orders(request):
qs = (
```

```
Order.objects.filter(buyer=request.user)
.prefetch_related("items", "items__product")
.order_by("-created_at")
)
paginator = Paginator(qs, 20)
page = paginator.get_page(request.GET.get("page") or 1)
return render(request, "orders/my_orders.html", {"page_obj": page, "orders": page.object_list})
@login_required
def seller_orders_list(request):
user = request.user
if not (is_seller_user(user) or is_owner_user(user)):
messages.info(request, "You don’t have access to seller orders.")
return redirect("dashboards:consumer")
qs = (
Order.objects.filter(status=Order.Status.PAID, paid_at__isnull=False)
.prefetch_related("items", "items__product", "items__seller")
.order_by("-paid_at", "-created_at")
)
if not is_owner_user(user):
qs = qs.filter(items__seller=user).distinct()
paginator = Paginator(qs, 25)
page = paginator.get_page(request.GET.get("page") or 1)
return render(request, "orders/seller_orders_list.html", {"page_obj": page, "orders": page.object_list})
@login_required
def seller_order_detail(request, order_id):
user = request.user
if not (is_seller_user(user) or is_owner_user(user)):
messages.info(request, "You don’t have access to seller orders.")
```

return redirect("dashboards:consumer")

order = get_object_or_404(

Order.objects.prefetch_related("items", "items__product", "items__seller"),

pk=order_id,

)

if not is_owner_user(user):

if not order.items.filter(seller=user).exists():

return redirect("orders:seller_orders_list")

seller_items = order.items.select_related("product", "seller").all()

if not is_owner_user(user):

seller_items = seller_items.filter(seller=user)

return render(request, "orders/seller_order_detail.html", {"order": order, "seller_items": seller_items})

orders.webhooks

orders/webhooks.py

from __future__ import annotations

import logging

from dataclasses import dataclass

from django.db import transaction

from django.http import HttpRequest, HttpResponse, HttpResponseBadRequest

from django.views.decorators.csrf import csrf_exempt

from payments.models import SellerBalanceEntry

from .models import Order, OrderEvent, StripeWebhookEvent

from .stripe_service import create_transfers_for_paid_order, verify_and_parse_webhook

logger = logging.getLogger(__name__)

def _extract_shipping_from_session_obj(session_obj: dict) -> dict:

shipping_details = session_obj.get("shipping_details") or {}

customer_details = session_obj.get("customer_details") or {}

name = shipping_details.get("name") or customer_details.get("name") or ""

```
phone = customer_details.get("phone") or ""
addr = shipping_details.get("address") or customer_details.get("address") or {}
return {
"name": name,
"phone": phone,
"line1": addr.get("line1") or "",
"line2": addr.get("line2") or "",
"city": addr.get("city") or "",
"state": addr.get("state") or "",
"postal_code": addr.get("postal_code") or "",
"country": addr.get("country") or "",
}
def _get_order_id_from_event(event: dict) -> str:
obj = (event.get("data") or {}).get("object") or {}
metadata = obj.get("metadata") or {}
order_id = (metadata.get("order_id") or "").strip()
if order_id:
return order_id
order_id = (obj.get("client_reference_id") or "").strip()
if order_id:
return order_id
return ""
def _record_event_once(*, stripe_event_id: str, event_type: str) -> bool:
_, created = StripeWebhookEvent.objects.get_or_create(
stripe_event_id=stripe_event_id,
defaults={"event_type": event_type or ""},
)
return created
```

```
def _transfers_already_created(order: Order) -> bool:
return order.events.filter(type=OrderEvent.Type.TRANSFER_CREATED).exists()
@dataclass(frozen=True)
class _RefundAllocation:
order_item_id: str
seller_id: str
debit_cents: int
def _allocate_refund_across_items(*, order: Order, refund_total_cents: int) -> list[_RefundAllocation]:
"""
Allocate refund across eligible items proportional to seller_net_cents.
Integer-safe:
- share_i = floor(refund_total * net_i / total_net) for all but last
- last gets remainder
- each seller debit is capped at that line's net
"""
refund_total_cents = max(0, int(refund_total_cents or 0))
items_all = list(order.items.all())
if refund_total_cents <= 0 or not items_all:
return []
eligible = [it for it in items_all if int(getattr(it, "seller_net_cents", 0) or 0) > 0]
if not eligible:
return []
nets = [int(it.seller_net_cents or 0) for it in eligible]
total_net = sum(nets)
if total_net <= 0:
return []
allocations: list[_RefundAllocation] = []
remaining = refund_total_cents
```

for idx, it in enumerate(eligible):

net = int(it.seller_net_cents or 0)

if net <= 0:

continue

if idx == len(eligible) - 1:

share = remaining

else:

share = (refund_total_cents * net) // total_net

share = max(0, min(share, remaining))

remaining -= share

debit = min(share, net)

if debit <= 0:

continue

allocations.append(

_RefundAllocation(

order_item_id=str(it.pk),

seller_id=str(it.seller_id),

debit_cents=int(debit),

)

)

return allocations

def _maybe_mark_order_refunded(*, order: Order, refunded_total_cents: int, note: str) -> None:

refunded_total_cents = int(refunded_total_cents or 0)

if refunded_total_cents <= 0:

return

total = int(order.total_cents or 0)

if total > 0 and refunded_total_cents < total:

OrderEvent.objects.create(

```
order=order,
type=OrderEvent.Type.WARNING,
message=f"Partial refund observed ({refunded_total_cents}c of {total}c). {note}",
)
return
if order.status != Order.Status.REFUNDED:
order.status = Order.Status.REFUNDED
order.save(update_fields=["status", "updated_at"])
OrderEvent.objects.create(order=order, type=OrderEvent.Type.REFUNDED, message=note)
@csrf_exempt
def stripe_webhook(request: HttpRequest) -> HttpResponse:
payload = request.body
sig_header = request.headers.get("Stripe-Signature", "")
if not sig_header:
return HttpResponseBadRequest("Missing signature")
try:
event = verify_and_parse_webhook(payload, sig_header)
except Exception:
return HttpResponseBadRequest("Invalid signature")
stripe_event_id = (event.get("id") or "").strip()
event_type = (event.get("type") or "").strip()
if not stripe_event_id or not event_type:
return HttpResponse(status=200)
if not _record_event_once(stripe_event_id=stripe_event_id, event_type=event_type):
return HttpResponse(status=200)
obj = (event.get("data") or {}).get("object") or {}
order_id = _get_order_id_from_event(event)
if not order_id:
```

logger.warning("Stripe event %s (%s) missing order_id mapping", stripe_event_id, event_type)

return HttpResponse(status=200)

try:

with transaction.atomic():

order = Order.objects.select_for_update().get(pk=order_id)

if event_type == "checkout.session.completed":

session_id = (obj.get("id") or "").strip()

payment_intent_id = (obj.get("payment_intent") or "").strip()

updated_fields: list[str] = []

if session_id and not order.stripe_session_id:

order.stripe_session_id = session_id

updated_fields.append("stripe_session_id")

if payment_intent_id and not order.stripe_payment_intent_id:

order.stripe_payment_intent_id = payment_intent_id

updated_fields.append("stripe_payment_intent_id")

if updated_fields:

updated_fields.append("updated_at")

order.save(update_fields=updated_fields)

order.mark_paid(payment_intent_id=payment_intent_id, session_id=session_id)

ship = _extract_shipping_from_session_obj(obj)

if any([ship["line1"], ship["city"], ship["postal_code"], ship["country"]]):

order.set_shipping_from_stripe(**ship)

create_transfers_for_paid_order(order=order, payment_intent_id=payment_intent_id)

return HttpResponse(status=200)

if event_type in {"charge.refunded", "refund.created", "refund.updated"}:

refunded_cents = int(obj.get("amount_refunded") or obj.get("amount") or 0)

payout_created = _transfers_already_created(order)

if payout_created and refunded_cents > 0:

```
allocs = _allocate_refund_across_items(order=order, refund_total_cents=refunded_cents)
for a in allocs:
SellerBalanceEntry.objects.create(
seller_id=a.seller_id,
amount_cents=-int(a.debit_cents),
reason=SellerBalanceEntry.Reason.REFUND,
order=order,
order_item_id=a.order_item_id,
note=f"Stripe refund via {event_type} (event={stripe_event_id})",
)
OrderEvent.objects.create(
order=order,
type=OrderEvent.Type.WARNING,
message=(
f"Refund received after payout. Recorded seller debits "
f"(refund={refunded_cents}c, event={stripe_event_id}, type={event_type})."
),
)
else:
OrderEvent.objects.create(
order=order,
type=OrderEvent.Type.WARNING,
message=(
f"Refund received (refund={refunded_cents}c, type={event_type}, event={stripe_event_id}). "
f"No seller debits recorded (payout_created={payout_created})."
),
)
_maybe_mark_order_refunded(
```

```
order=order,
refunded_total_cents=refunded_cents,
note=f"Stripe refund observed ({event_type}, {refunded_cents}c, event={stripe_event_id})",
)
return HttpResponse(status=200)
if event_type in {"charge.dispute.created", "charge.dispute.updated"}:
status = (obj.get("status") or "").strip().lower()
OrderEvent.objects.create(
order=order,
type=OrderEvent.Type.WARNING,
message=f"Dispute event: {event_type} status={status or 'unknown'} event={stripe_event_id}",
)
return HttpResponse(status=200)
if event_type == "charge.dispute.closed":
status = (obj.get("status") or "").strip().lower()
payout_created = _transfers_already_created(order)
if status == "lost":
if payout_created:
for it in order.items.all():
net = int(it.seller_net_cents or 0)
if net <= 0:
continue
SellerBalanceEntry.objects.create(
seller_id=str(it.seller_id),
amount_cents=-net,
reason=SellerBalanceEntry.Reason.CHARGEBACK,
order=order,
order_item=it,
```

```
note=f"Chargeback lost (event={stripe_event_id})",
)
OrderEvent.objects.create(
order=order,
type=OrderEvent.Type.WARNING,
message=(
"Chargeback lost. Seller debited net (payout already created). "
"Dispute fee may require manual adjustment."
),
)
else:
OrderEvent.objects.create(
order=order,
type=OrderEvent.Type.WARNING,
message="Chargeback lost before payout. No seller debits recorded (no payout created).",
)
if order.status != Order.Status.REFUNDED:
order.status = Order.Status.REFUNDED
order.save(update_fields=["status", "updated_at"])
OrderEvent.objects.create(order=order, type=OrderEvent.Type.REFUNDED, message="Chargeback lost")
return HttpResponse(status=200)
OrderEvent.objects.create(
order=order,
type=OrderEvent.Type.WARNING,
message=f"Chargeback closed with status={status or 'unknown'} event={stripe_event_id}",
)
return HttpResponse(status=200)
return HttpResponse(status=200)
```

except Order.DoesNotExist:

return HttpResponse(status=200)

except Exception:

logger.exception(

"Stripe webhook processing failed event=%s type=%s order=%s",

stripe_event_id,

event_type,

order_id,

)

return HttpResponse(status=200)

payments.admin

payments/admin.py

from __future__ import annotations

from django.contrib import admin

from django.db.models import Sum

from .models import SellerBalanceEntry, SellerStripeAccount

@admin.register(SellerStripeAccount)

class SellerStripeAccountAdmin(admin.ModelAdmin):

list_display = (

"user",

"stripe_account_id",

"details_submitted",

"charges_enabled",

"payouts_enabled",

"current_balance",

"onboarding_started_at",

"onboarding_completed_at",

"updated_at",

)

search_fields = ("user__username", "user__email", "stripe_account_id")

list_filter = ("details_submitted", "charges_enabled", "payouts_enabled")

def current_balance(self, obj):

total = (

SellerBalanceEntry.objects.filter(seller=obj.user)

.aggregate(total=Sum("amount_cents"))

.get("total")

or 0

)

return f"${total / 100:,.2f}"

current_balance.short_description = "Balance"

@admin.register(SellerBalanceEntry)

class SellerBalanceEntryAdmin(admin.ModelAdmin):

list_display = (

"created_at",

"seller",

"amount_cents",

"reason",

"order",

"order_item",

"note",

)

list_filter = ("reason", "created_at")

search_fields = (

"seller__username",

"seller__email",

"order__id",

"note",

)

readonly_fields = (

"seller",

"amount_cents",

"reason",

"order",

"order_item",

"note",

"created_at",

)

payments.context_processors

from __future__ import annotations

from django.urls import NoReverseMatch, reverse

from payments.models import SellerStripeAccount

from products.permissions import is_owner_user, is_seller_user

def seller_stripe_status(request):

"""Global template context.

- seller_stripe_ready: True/False/None

None => not a seller (hide badge)

- has_connect_sync: bool (payments:connect_sync exists)

- user_is_owner: bool (owner/admin override)

- user_is_seller: bool (seller OR owner/admin)

Notes:

- These flags avoid templates touching `user.profile` directly, which can raise

RelatedObjectDoesNotExist if a Profile row isn't created yet.

"""

user = getattr(request, "user", None)

user_is_owner = False

user_is_seller = False

seller_stripe_ready = None

if user and getattr(user, "is_authenticated", False):

user_is_owner = is_owner_user(user)

user_is_seller = is_seller_user(user)

if user_is_owner:

seller_stripe_ready = True

elif user_is_seller:

acct = SellerStripeAccount.objects.filter(user=user).first()

seller_stripe_ready = bool(acct and acct.is_ready)

try:

reverse("payments:connect_sync")

has_connect_sync = True

except NoReverseMatch:

has_connect_sync = False

return {

"seller_stripe_ready": seller_stripe_ready,

"has_connect_sync": has_connect_sync,

"user_is_owner": user_is_owner,

"user_is_seller": user_is_seller,

}

payments.decorators

payments/decorators.py

from __future__ import annotations

from functools import wraps

from django.contrib import messages

from django.shortcuts import redirect

```
from products.permissions import is_owner_user
from payments.models import SellerStripeAccount
def stripe_ready_required(view_func):
"""
Gate seller uploads/publishing until Stripe Connect is fully enabled.
Owner/admin bypasses the gate.
"""
@wraps(view_func)
def _wrapped(request, *args, **kwargs):
if request.user.is_authenticated and is_owner_user(request.user):
return view_func(request, *args, **kwargs)
acct = SellerStripeAccount.objects.filter(user=request.user).first()
if not acct or not acct.is_ready:
messages.warning(
request,
"You must finish Stripe onboarding before you can create or modify listings.",
)
return redirect("payments:connect_status")
return view_func(request, *args, **kwargs)
return _wrapped
payments.models
# payments/models.py
from __future__ import annotations
import uuid
from django.conf import settings
from django.db import models
from django.utils import timezone
class SellerStripeAccount(models.Model):
```

"""

Stores Stripe Connect Express account linkage for a seller user.

"""

user = models.OneToOneField(

settings.AUTH_USER_MODEL,

on_delete=models.CASCADE,

related_name="stripe_connect",

)

stripe_account_id = models.CharField(

max_length=255, blank=True, default="", db_index=True

)

details_submitted = models.BooleanField(default=False)

charges_enabled = models.BooleanField(default=False)

payouts_enabled = models.BooleanField(default=False)

onboarding_started_at = models.DateTimeField(null=True, blank=True)

onboarding_completed_at = models.DateTimeField(null=True, blank=True)

created_at = models.DateTimeField(auto_now_add=True)

updated_at = models.DateTimeField(auto_now=True)

class Meta:

indexes = [

models.Index(fields=["stripe_account_id"]),

models.Index(

fields=["details_submitted", "charges_enabled", "payouts_enabled"]

),

]

def __str__(self) -> str:

return f"SellerStripeAccount<{self.user_id}> {self.stripe_account_id or 'unlinked'}"

@property

def is_ready(self) -> bool:

return bool(self.stripe_account_id) and self.details_submitted and self.charges_enabled and self.payouts_enabled

def mark_onboarding_started(self) -> None:

if not self.onboarding_started_at:

self.onboarding_started_at = timezone.now()

self.save(update_fields=["onboarding_started_at", "updated_at"])

def mark_onboarding_completed_if_ready(self) -> None:

if self.is_ready and not self.onboarding_completed_at:

self.onboarding_completed_at = timezone.now()

self.save(update_fields=["onboarding_completed_at", "updated_at"])

class SellerBalanceEntry(models.Model):

"""

Append-only ledger for seller balances.

amount_cents:

> 0 => platform owes seller

< 0 => seller owes platform

"""

class Reason(models.TextChoices):

PAYOUT = "payout", "Payout"

REFUND = "refund", "Refund"

CHARGEBACK = "chargeback", "Chargeback"

ADJUSTMENT = "adjustment", "Manual adjustment"

id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

seller = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.CASCADE,

related_name="balance_entries",

)

```
amount_cents = models.IntegerField(
help_text="Signed cents. Positive = owed to seller, negative = seller owes platform."
)
reason = models.CharField(max_length=32, choices=Reason.choices)
order = models.ForeignKey(
"orders.Order",
null=True,
blank=True,
on_delete=models.SET_NULL,
related_name="seller_balance_entries",
)
order_item = models.ForeignKey(
"orders.OrderItem",
null=True,
blank=True,
on_delete=models.SET_NULL,
related_name="seller_balance_entries",
)
note = models.TextField(blank=True, default="")
created_at = models.DateTimeField(default=timezone.now, db_index=True)
class Meta:
indexes = [
models.Index(fields=["seller", "-created_at"]),
models.Index(fields=["reason", "-created_at"]),
]
ordering = ("-created_at",)
def __str__(self) -> str:
return f"{self.seller_id}: {self.amount_cents} ({self.reason})"
```

payments.services

payments/services.py

from __future__ import annotations

from django.db.models import Sum

from payments.models import SellerBalanceEntry

def get_seller_balance_cents(*, seller) -> int:

"""

Returns signed balance.

Negative => seller owes platform.

"""

agg = SellerBalanceEntry.objects.filter(seller=seller).aggregate(

total=Sum("amount_cents")

)

return int(agg["total"] or 0)

payments.stripe_connect

from __future__ import annotations

import stripe

from django.conf import settings

from django.urls import reverse

def _base_url() -> str:

base = (getattr(settings, "SITE_BASE_URL", "") or "").strip()

if not base:

raise RuntimeError("SITE_BASE_URL is required for Stripe Connect links.")

return base.rstrip("/")

def configure_stripe() -> None:

stripe.api_key = settings.STRIPE_SECRET_KEY

def create_express_account(*, email: str, country: str = "US") -> stripe.Account:

configure_stripe()

return stripe.Account.create(

type="express",

country=country,

email=email,

capabilities={

"card_payments": {"requested": True},

"transfers": {"requested": True},

},

business_type="individual", # MVP default; Stripe will ask for details

)

def create_account_link(*, stripe_account_id: str) -> stripe.AccountLink:

"""

Returns a Stripe-hosted onboarding link for an Express account.

"""

configure_stripe()

refresh_url = f"{_base_url()}{reverse('payments:connect_refresh')}"

return_url = f"{_base_url()}{reverse('payments:connect_return')}"

return stripe.AccountLink.create(

account=stripe_account_id,

refresh_url=refresh_url,

return_url=return_url,

type="account_onboarding",

)

def retrieve_account(stripe_account_id: str) -> stripe.Account:

configure_stripe()

return stripe.Account.retrieve(stripe_account_id)

payments.urls

from __future__ import annotations

```
from django.urls import path
from . import views
app_name = "payments"
urlpatterns = [
path("connect/", views.connect_status, name="connect_status"),
path("connect/start/", views.connect_start, name="connect_start"),
path("connect/sync/", views.connect_sync, name="connect_sync"),
path("connect/refresh/", views.connect_refresh, name="connect_refresh"),
path("connect/return/", views.connect_return, name="connect_return"),
# Webhook (Connect)
path(
"stripe/connect/webhook/",
views.stripe_connect_webhook,
name="stripe_connect_webhook",
),
# Seller payouts / ledger
path("payouts/", views.payouts_dashboard, name="payouts_dashboard"),
]
payments.utils
from __future__ import annotations
from decimal import Decimal, ROUND_HALF_UP
from typing import Union
from payments.models import SellerStripeAccount
from products.permissions import is_owner_user
def seller_is_stripe_ready(seller_user) -> bool:
"""
True if seller can receive payouts (Stripe Connect fully enabled).
Owner/admin bypass is treated as ready.
```

"""

if seller_user and is_owner_user(seller_user):

return True

acct = SellerStripeAccount.objects.filter(user=seller_user).first()

return bool(acct and acct.is_ready)

MoneyLike = Union[Decimal, int, float, str, None]

def money_to_cents(value: MoneyLike) -> int:

"""Convert a currency amount (e.g. dollars) to integer cents safely.

Accepts Decimal, int, float, or numeric string. Uses ROUND_HALF_UP.

NOTE:

- If caller passes an int, we assume it is already cents (by convention).

"""

if value is None:

return 0

if isinstance(value, int):

return int(value)

dec = value if isinstance(value, Decimal) else Decimal(str(value))

cents = (dec * Decimal("100")).quantize(Decimal("1"), rounding=ROUND_HALF_UP)

return int(cents)

def cents_to_money(cents: int) -> Decimal:

return (Decimal(int(cents)) / Decimal("100")).quantize(Decimal("0.01"))

payments.views

from __future__ import annotations

from django.contrib import messages

from django.core.paginator import Paginator

from django.db.models import Q

from django.http import HttpResponse, HttpResponseBadRequest

from django.shortcuts import redirect, render

from django.views.decorators.csrf import csrf_exempt

from django.views.decorators.http import require_POST

from products.permissions import seller_required

from .models import SellerBalanceEntry, SellerStripeAccount

from .services import get_seller_balance_cents

from .stripe_connect import create_account_link, create_express_account, retrieve_account

def _seller_email_for_connect(user) -> str:

"""Pick a stable email for Stripe Connect.

Preference order:

1) user.email (if you have a custom user model with email)

2) user.profile.email (your Profile model stores contact email)

"""

email = (getattr(user, "email", "") or "").strip()

if email:

return email

profile = getattr(user, "profile", None)

if profile is not None:

email = (getattr(profile, "email", "") or "").strip()

if email:

return email

return ""

def _refresh_connect_status(obj: SellerStripeAccount) -> None:

"""Refresh Stripe Connect status fields from Stripe (best-effort)."""

if not obj.stripe_account_id:

return

acct = retrieve_account(obj.stripe_account_id)

obj.details_submitted = bool(acct.get("details_submitted"))

obj.charges_enabled = bool(acct.get("charges_enabled"))

```
obj.payouts_enabled = bool(acct.get("payouts_enabled"))
obj.save(
update_fields=[
"details_submitted",
"charges_enabled",
"payouts_enabled",
"updated_at",
]
)
obj.mark_onboarding_completed_if_ready()
@seller_required
def connect_status(request):
"""Seller-facing status page + CTA to start/continue Stripe onboarding."""
obj, _ = SellerStripeAccount.objects.get_or_create(user=request.user)
# Optional: light refresh on GET if linked but not ready yet.
if obj.stripe_account_id and not obj.is_ready:
try:
_refresh_connect_status(obj)
except Exception:
pass
context = {
"stripe": obj,
"ready": obj.is_ready,
}
return render(request, "payments/connect_status.html", context)
@seller_required
@require_POST
def connect_start(request):
```

"""Create Stripe Express account if needed, then redirect to onboarding link."""

obj, _ = SellerStripeAccount.objects.get_or_create(user=request.user)

if not obj.stripe_account_id:

email = _seller_email_for_connect(request.user)

if not email:

messages.error(

request,

"Your account is missing an email. Add one in your profile, then try again.",

)

return redirect("payments:connect_status")

acct = create_express_account(email=email, country="US")

obj.stripe_account_id = acct["id"]

obj.details_submitted = bool(acct.get("details_submitted"))

obj.charges_enabled = bool(acct.get("charges_enabled"))

obj.payouts_enabled = bool(acct.get("payouts_enabled"))

obj.save(

update_fields=[

"stripe_account_id",

"details_submitted",

"charges_enabled",

"payouts_enabled",

"updated_at",

]

)

obj.mark_onboarding_started()

link = create_account_link(stripe_account_id=obj.stripe_account_id)

return redirect(link["url"])

@seller_required

```
@require_POST
def connect_sync(request):
"""Manual refresh button for sellers (handy if webhook delivery is delayed)."""
obj, _ = SellerStripeAccount.objects.get_or_create(user=request.user)
if not obj.stripe_account_id:
messages.info(request, "You haven’t started Stripe onboarding yet.")
return redirect("payments:connect_status")
try:
_refresh_connect_status(obj)
except Exception:
messages.info(
request, "Couldn’t refresh Stripe status right now. Try again in a moment."
)
return redirect("payments:connect_status")
if obj.is_ready:
messages.success(request, "Stripe payouts are enabled. You’re ready to sell!")
else:
messages.info(
request,
"Stripe status refreshed. If anything is missing, click Continue to finish onboarding.",
)
return redirect("payments:connect_status")
@seller_required
def connect_refresh(request):
"""Stripe sends user here if they abandon or the session expires."""
messages.info(
request, "Your Stripe onboarding link expired. Click Continue to generate a new one."
)
```

return redirect("payments:connect_status")

@seller_required

def connect_return(request):

"""Stripe sends user here after onboarding. We refresh the account status."""

obj, _ = SellerStripeAccount.objects.get_or_create(user=request.user)

if obj.stripe_account_id:

try:

_refresh_connect_status(obj)

except Exception:

pass

if obj.is_ready:

messages.success(request, "Stripe payouts are enabled. You’re ready to sell!")

else:

messages.info(

request,

"Stripe setup saved. If anything is missing, click Continue to finish onboarding.",

)

return redirect("payments:connect_status")

@seller_required

def payouts_dashboard(request):

"""

Seller payouts / ledger page.

Shows:

- current signed balance (platform owes seller if positive; seller owes platform if negative)

- ledger entries (append-only)

- optional filters: reason, q (note/order id)

"""

seller = request.user

```
balance_cents = int(get_seller_balance_cents(seller=seller) or 0)
reason = (request.GET.get("reason") or "").strip()
q = (request.GET.get("q") or "").strip()
entries = SellerBalanceEntry.objects.filter(seller=seller).select_related(
"order", "order_item"
)
if reason:
entries = entries.filter(reason=reason)
if q:
entries = entries.filter(Q(note__icontains=q) | Q(order__id__icontains=q))
entries = entries.order_by("-created_at")
paginator = Paginator(entries, 50)
page = paginator.get_page(request.GET.get("page") or 1)
stripe_obj, _ = SellerStripeAccount.objects.get_or_create(user=seller)
context = {
"balance_cents": balance_cents,
"page_obj": page,
"entries": page.object_list,
"reason": reason,
"q": q,
"reasons": SellerBalanceEntry.Reason.choices,
"stripe": stripe_obj,
"stripe_ready": stripe_obj.is_ready,
}
return render(request, "payments/payouts_dashboard.html", context)
def _verify_and_parse_connect_webhook(payload: bytes, sig_header: str):
"""Verify Stripe webhook for Connect events."""
import stripe
```

```
from django.conf import settings
stripe.api_key = settings.STRIPE_SECRET_KEY
secret = getattr(settings, "STRIPE_CONNECT_WEBHOOK_SECRET", "")
if not secret:
raise RuntimeError("STRIPE_CONNECT_WEBHOOK_SECRET is not configured")
return stripe.Webhook.construct_event(
payload=payload,
sig_header=sig_header,
secret=secret,
)
@csrf_exempt
@require_POST
def stripe_connect_webhook(request):
"""Stripe webhook endpoint to keep Connect statuses updated."""
payload = request.body
sig_header = request.headers.get("Stripe-Signature", "")
if not sig_header:
return HttpResponseBadRequest("Missing signature")
try:
event = _verify_and_parse_connect_webhook(payload, sig_header)
except Exception:
return HttpResponseBadRequest("Invalid signature")
event_type = event.get("type", "")
data_object = (event.get("data") or {}).get("object") or {}
if event_type == "account.updated":
acct_id = data_object.get("id", "")
if acct_id:
obj = SellerStripeAccount.objects.filter(stripe_account_id=acct_id).first()
```

```
if obj:
obj.details_submitted = bool(data_object.get("details_submitted"))
obj.charges_enabled = bool(data_object.get("charges_enabled"))
obj.payouts_enabled = bool(data_object.get("payouts_enabled"))
obj.save(
update_fields=[
"details_submitted",
"charges_enabled",
"payouts_enabled",
"updated_at",
]
)
obj.mark_onboarding_completed_if_ready()
return HttpResponse(status=200)
```

```
products.admin
from __future__ import annotations
from django.contrib import admin
from .models import (
Product,
ProductImage,
ProductDigital,
ProductPhysical,
DigitalAsset,
ProductEngagementEvent,
)
class ProductImageInline(admin.TabularInline):
model = ProductImage
extra = 0
```

class DigitalAssetInline(admin.TabularInline):

model = DigitalAsset

extra = 0

@admin.register(Product)

class ProductAdmin(admin.ModelAdmin):

list_display = (

"id",

"title",

"kind",

"seller",

"category",

"is_active",

"is_featured",

"is_trending",

"created_at",

)

list_filter = ("kind", "is_active", "is_featured", "is_trending", "category")

search_fields = ("title", "slug", "seller__username", "short_description", "description")

prepopulated_fields = {"slug": ("title",)}

inlines = [ProductImageInline, DigitalAssetInline]

@admin.register(ProductDigital)

class ProductDigitalAdmin(admin.ModelAdmin):

list_display = ("id", "product", "file_count")

@admin.register(ProductPhysical)

class ProductPhysicalAdmin(admin.ModelAdmin):

list_display = ("id", "product", "material", "color")

@admin.register(ProductImage)

class ProductImageAdmin(admin.ModelAdmin):

list_display = ("id", "product", "is_primary", "sort_order", "created_at")

list_filter = ("is_primary",)

search_fields = ("product__title",)

@admin.register(DigitalAsset)

class DigitalAssetAdmin(admin.ModelAdmin):

list_display = ("id", "product", "original_filename", "created_at")

search_fields = ("product__title", "original_filename")

@admin.register(ProductEngagementEvent)

class ProductEngagementEventAdmin(admin.ModelAdmin):

list_display = ("id", "event_type", "product", "created_at")

list_filter = ("event_type",)

search_fields = ("product__title", "product__seller__username")

date_hierarchy = "created_at"

products.forms

from __future__ import annotations

from django import forms

from catalog.models import Category

from .models import Product, ProductImage, DigitalAsset

class ProductForm(forms.ModelForm):

class Meta:

model = Product

fields = [

"kind",

"title",

"slug",

"short_description",

"description",

"category",

"is_free",

"price",

"is_active",

"is_featured",

"is_trending",

]

widgets = {

"description": forms.Textarea(attrs={"rows": 6}),

}

def __init__(self, *args, **kwargs):

self.user = kwargs.pop("user", None)

super().__init__(*args, **kwargs)

For MVP: show only active categories, ordered nicely

self.fields["category"].queryset = Category.objects.filter(is_active=True).order_by("type", "parent_id", "sort_order", "name")

If kind is already known, narrow category choices to the matching tree

kind_value = None

if self.instance and self.instance.pk:

kind_value = self.instance.kind

else:

kind_value = self.data.get("kind") or self.initial.get("kind")

if kind_value in (Product.Kind.MODEL, Product.Kind.FILE):

expected_type = Category.CategoryType.MODEL if kind_value == Product.Kind.MODEL else Category.CategoryType.FILE

self.fields["category"].queryset = self.fields["category"].queryset.filter(type=expected_type)

Make slug optional; we auto-generate in save() if empty

self.fields["slug"].required = False

def clean(self):

cleaned = super().clean()

is_free = cleaned.get("is_free")

price = cleaned.get("price")

if is_free:

cleaned["price"] = 0

else:

allow empty price to error clearly

if price is None:

self.add_error("price", "Price is required unless the item is marked free.")

else:

try:

if price <= 0:

self.add_error("price", "Price must be greater than $0.00 unless the item is marked free.")

except TypeError:

self.add_error("price", "Enter a valid price.")

return cleaned

class ProductImageUploadForm(forms.ModelForm):

class Meta:

model = ProductImage

fields = ["image", "alt_text", "is_primary", "sort_order"]

class DigitalAssetUploadForm(forms.ModelForm):

class Meta:

model = DigitalAsset

fields = ["file", "original_filename"]

def save(self, commit=True):

obj = super().save(commit=False)

if not obj.original_filename and obj.file:

obj.original_filename = getattr(obj.file, "name", "") or ""

if commit:

obj.save()

return obj

products.models

from __future__ import annotations

from decimal import Decimal

from django.conf import settings

from django.core.validators import MinValueValidator

from django.db import models

from django.urls import reverse

from django.utils.text import slugify

class Product(models.Model):

class Kind(models.TextChoices):

MODEL = "MODEL", "3D Model (Physical)"

FILE = "FILE", "3D File (Digital)"

Ownership / publishing

seller = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.CASCADE,

related_name="products",

help_text="The user who owns this listing (seller).",

)

kind = models.CharField(max_length=10, choices=Kind.choices)

Core listing fields

title = models.CharField(max_length=160)

slug = models.SlugField(max_length=180)

short_description = models.CharField(max_length=280, blank=True)

description = models.TextField(blank=True)

Category: must match the correct tree for kind (validated in clean())

category = models.ForeignKey(

"catalog.Category",

on_delete=models.PROTECT,

related_name="products",

)

Pricing

price = models.DecimalField(

max_digits=10,

decimal_places=2,

default=Decimal("0.00"),

validators=[MinValueValidator(Decimal("0.00"))],

)

is_free = models.BooleanField(default=False)

Visibility

is_active = models.BooleanField(default=True)

Home page buckets (simple flags for MVP; can become computed later)

is_featured = models.BooleanField(default=False)

is_trending = models.BooleanField(default=False)

Timestamps

created_at = models.DateTimeField(auto_now_add=True)

updated_at = models.DateTimeField(auto_now=True)

class Meta:

unique_together = (("seller", "slug"),)

indexes = [

models.Index(fields=["kind", "is_active", "created_at"]),

models.Index(fields=["is_featured", "is_active"]),

models.Index(fields=["is_trending", "is_active"]),

models.Index(fields=["slug"]),

```
]
ordering = ["-created_at"]
def __str__(self) -> str:
return f"{self.title} ({self.get_kind_display()})" # type: ignore[attr-defined]
def save(self, *args, **kwargs):
if not self.slug:
self.slug = slugify(self.title)[:180]
super().save(*args, **kwargs)
def clean(self):
"""Enforce kind/category tree consistency and pricing rules."""
from django.core.exceptions import ValidationError
from catalog.models import Category
if self.category_id:
if self.kind == Product.Kind.MODEL and self.category.type != Category.CategoryType.MODEL:
raise ValidationError({"category": "Model products must use a 3D Models category."})
if self.kind == Product.Kind.FILE and self.category.type != Category.CategoryType.FILE:
raise ValidationError({"category": "File products must use a 3D Files category."})
if self.is_free:
self.price = Decimal("0.00")
if not self.is_free and self.price <= Decimal("0.00"):
raise ValidationError({"price": "Price must be greater than $0.00 unless the item is marked free."})
@property
def display_price(self) -> str:
return "Free" if self.is_free else f"${self.price:,.2f}"
def get_absolute_url(self) -> str:
return reverse("products:detail", kwargs={"pk": self.pk, "slug": self.slug})
@property
def primary_image(self):
```

return self.images.filter(is_primary=True).first() or self.images.order_by("sort_order", "id").first()

@property

def seller_public_name(self) -> str:

"""Public-facing seller display name (shop_name if set; else username).

Never raises if Profile is missing.

"""

try:

profile = self.seller.profile

except Exception:

profile = None

if profile is not None:

try:

name = (profile.shop_name or "").strip()

if name:

return name

except Exception:

pass

return getattr(self.seller, "username", "Seller")

class ProductImage(models.Model):

product = models.ForeignKey(Product, on_delete=models.CASCADE, related_name="images")

image = models.ImageField(upload_to="product_images/")

alt_text = models.CharField(max_length=160, blank=True)

is_primary = models.BooleanField(default=False)

sort_order = models.PositiveIntegerField(default=0)

created_at = models.DateTimeField(auto_now_add=True)

class Meta:

indexes = [

models.Index(fields=["product", "is_primary", "sort_order"]),

```
]
ordering = ["sort_order", "id"]
def __str__(self) -> str:
return f"Image<{self.product_id}>#{self.pk}"
class ProductDigital(models.Model):
"""
Extension table for digital file products.
"""
product = models.OneToOneField(Product, on_delete=models.CASCADE, related_name="digital")
# Optional fields (MVP)
license_text = models.TextField(blank=True)
file_count = models.PositiveIntegerField(default=0)
def __str__(self) -> str:
return f"Digital<{self.product_id}>"
class DigitalAsset(models.Model):
"""
Individual downloadable asset (STL/OBJ/3MF/etc).
In MVP we store the file; later we can gate download via paid orders.
"""
product = models.ForeignKey(Product, on_delete=models.CASCADE, related_name="digital_assets")
file = models.FileField(upload_to="digital_assets/")
original_filename = models.CharField(max_length=255, blank=True)
created_at = models.DateTimeField(auto_now_add=True)
class Meta:
ordering = ["id"]
def __str__(self) -> str:
return f"Asset<{self.product_id}>#{self.pk}"
class ProductPhysical(models.Model):
```

"""

Extension table for physical printed model products.

"""

product = models.OneToOneField(Product, on_delete=models.CASCADE, related_name="physical")

MVP placeholders

material = models.CharField(max_length=120, blank=True)

color = models.CharField(max_length=120, blank=True)

width_mm = models.PositiveIntegerField(null=True, blank=True)

height_mm = models.PositiveIntegerField(null=True, blank=True)

depth_mm = models.PositiveIntegerField(null=True, blank=True)

def __str__(self) -> str:

return f"Physical<{self.product_id}>"

class ProductEngagementEvent(models.Model):

"""

Lightweight event log to make Trending feel real on day 1.

Event types:

- VIEW (product detail page view)

- ADD_TO_CART (cart add action)

- CLICK (click from a product card; recorded when detail loads with ?src=...)

"""

class EventType(models.TextChoices):

VIEW = "VIEW", "View"

ADD_TO_CART = "ADD_TO_CART", "Add to cart"

CLICK = "CLICK", "Click"

product = models.ForeignKey(

Product,

on_delete=models.CASCADE,

related_name="engagement_events",

)

event_type = models.CharField(max_length=20, choices=EventType.choices)

created_at = models.DateTimeField(auto_now_add=True)

class Meta:

indexes = [

models.Index(fields=["product", "event_type", "created_at"]),

models.Index(fields=["event_type", "created_at"]),

]

ordering = ["-created_at"]

def __str__(self) -> str:

return f"{self.event_type} product={self.product_id} at {self.created_at}"

products.permissions

from __future__ import annotations

from functools import wraps

from typing import Callable

from django.contrib.auth.decorators import login_required

from django.http import HttpRequest, HttpResponse

from django.shortcuts import redirect

from django.urls import reverse

def _get_profile(user):

Works with a typical OneToOne Profile named `profile`

return getattr(user, "profile", None)

def is_owner_user(user) -> bool:

if not user or not user.is_authenticated:

return False

Owner/admin override paths

if getattr(user, "is_superuser", False) or getattr(user, "is_staff", False):

return True

profile = _get_profile(user)

return bool(getattr(profile, "is_owner", False))

def is_seller_user(user) -> bool:

if not user or not user.is_authenticated:

return False

if is_owner_user(user):

return True

profile = _get_profile(user)

return bool(getattr(profile, "is_seller", False))

def seller_required(view_func: Callable[[HttpRequest, ...], HttpResponse]):

"""

Requires:

- authenticated

- seller OR owner/admin

"""

@login_required

@wraps(view_func)

def _wrapped(request: HttpRequest, *args, **kwargs):

if not is_seller_user(request.user):

return redirect(reverse("home"))

return view_func(request, *args, **kwargs)

return _wrapped

products.urls_seller

from django.urls import path

from . import views_seller

urlpatterns = [

path("", views_seller.seller_product_list, name="seller_list"),

path("new/", views_seller.seller_product_create, name="seller_create"),

path("<int:pk>/edit/", views_seller.seller_product_edit, name="seller_edit"),

path("<int:pk>/images/", views_seller.seller_product_images, name="seller_images"),

path("<int:pk>/images/<int:image_id>/delete/", views_seller.seller_product_image_delete, name="seller_image_delete"),

path("<int:pk>/assets/", views_seller.seller_product_assets, name="seller_assets"),

path("<int:pk>/assets/<int:asset_id>/delete/", views_seller.seller_product_asset_delete, name="seller_asset_delete"),

path("<int:pk>/toggle-active/", views_seller.seller_product_toggle_active, name="seller_toggle_active"),

]

products.urls

from django.urls import path

from . import views

from . import views_seller

app_name = "products"

urlpatterns = [

Public browsing

path("", views.product_list, name="list"),

path("models/", views.models_list, name="models"),

path("files/", views.files_list, name="files"),

Engagement redirect (logs CLICK then redirects to detail)

path("go/<int:pk>/<slug:slug>/", views.product_go, name="go"),

Canonical detail

path("<int:pk>/<slug:slug>/", views.product_detail, name="detail"),

Seller area

path("seller/", views_seller.seller_product_list, name="seller_list"),

path("seller/new/", views_seller.seller_product_create, name="seller_create"),

path("seller/<int:pk>/edit/", views_seller.seller_product_edit, name="seller_edit"),

path("seller/<int:pk>/images/", views_seller.seller_product_images, name="seller_images"),

path("seller/<int:pk>/images/<int:image_id>/delete/", views_seller.seller_product_image_delete, name="seller_image_delete"),

path("seller/<int:pk>/assets/", views_seller.seller_product_assets, name="seller_assets"),

path("seller/<int:pk>/assets/<int:asset_id>/delete/", views_seller.seller_product_asset_delete, name="seller_asset_delete"),

path("seller/<int:pk>/toggle-active/", views_seller.seller_product_toggle_active, name="seller_toggle_active"),

]

products.views_seller

from __future__ import annotations

from django.contrib import messages

from django.db.models import Q

from django.http import Http404

from django.shortcuts import get_object_or_404, redirect, render

from payments.models import SellerStripeAccount

from payments.decorators import stripe_ready_required

from .forms import ProductForm, ProductImageUploadForm, DigitalAssetUploadForm

from .models import Product, ProductImage, DigitalAsset

from .permissions import seller_required, is_owner_user

def _can_edit_product(user, product: Product) -> bool:

if is_owner_user(user):

return True

return product.seller_id == user.id

def _get_owned_product_or_404(request, pk: int) -> Product:

product = get_object_or_404(Product, pk=pk)

if not _can_edit_product(request.user, product):

raise Http404("Not found")

return product

@seller_required

def seller_product_list(request):

"""

Seller dashboard: list your products.

Owner/admin sees all products.

NOTE: This is NOT gated by Stripe readiness. Sellers can still view what they have.

"""

qs = Product.objects.select_related("category", "seller").prefetch_related("images").order_by("-created_at")

if not is_owner_user(request.user):

qs = qs.filter(seller=request.user)

q = (request.GET.get("q") or "").strip()

if q:

qs = qs.filter(Q(title__icontains=q) | Q(short_description__icontains=q) | Q(description__icontains=q))

stripe_account = None

stripe_ready = True

if not is_owner_user(request.user):

stripe_account = SellerStripeAccount.objects.filter(user=request.user).first()

stripe_ready = bool(stripe_account and stripe_account.is_ready)

return render(

request,

"products/seller/product_list.html",

{

"products": qs,

"q": q,

"stripe_account": stripe_account,

"stripe_ready": stripe_ready,

},

)

@seller_required

@stripe_ready_required

```
def seller_product_create(request):
if request.method == "POST":
form = ProductForm(request.POST, user=request.user)
if form.is_valid():
product = form.save(commit=False)
product.seller = request.user
product.save()
messages.success(request, "Product created. Next: add images (and digital assets if applicable).")
return redirect("products:seller_images", pk=product.pk)
else:
form = ProductForm(user=request.user)
return render(request, "products/seller/product_form.html", {"form": form, "mode": "create"})
@seller_required
@stripe_ready_required
def seller_product_edit(request, pk: int):
product = _get_owned_product_or_404(request, pk)
if request.method == "POST":
form = ProductForm(request.POST, instance=product, user=request.user)
if form.is_valid():
form.save()
messages.success(request, "Product updated.")
return redirect("products:seller_list")
else:
form = ProductForm(instance=product, user=request.user)
return render(
request,
"products/seller/product_form.html",
{"form": form, "mode": "edit", "product": product},
```

)

@seller_required

@stripe_ready_required

def seller_product_images(request, pk: int):

product = _get_owned_product_or_404(request, pk)

if request.method == "POST":

form = ProductImageUploadForm(request.POST, request.FILES)

if form.is_valid():

img: ProductImage = form.save(commit=False)

img.product = product

img.save()

If primary set, unset others

if img.is_primary:

ProductImage.objects.filter(product=product).exclude(pk=img.pk).update(is_primary=False)

messages.success(request, "Image uploaded.")

return redirect("products:seller_images", pk=product.pk)

else:

form = ProductImageUploadForm()

images = product.images.all().order_by("sort_order", "id")

return render(

request,

"products/seller/product_images.html",

{"product": product, "form": form, "images": images},

)

@seller_required

@stripe_ready_required

def seller_product_image_delete(request, pk: int, image_id: int):

product = _get_owned_product_or_404(request, pk)

img = get_object_or_404(ProductImage, pk=image_id, product=product)

if request.method == "POST":

was_primary = img.is_primary

img.delete()

if was_primary:

Make the next image primary automatically (nice UX)

next_img = ProductImage.objects.filter(product=product).order_by("sort_order", "id").first()

if next_img:

next_img.is_primary = True

next_img.save(update_fields=["is_primary"])

messages.success(request, "Image deleted.")

return redirect("products:seller_images", pk=product.pk)

return redirect("products:seller_images", pk=product.pk)

@seller_required

@stripe_ready_required

def seller_product_assets(request, pk: int):

product = _get_owned_product_or_404(request, pk)

if product.kind != Product.Kind.FILE:

messages.info(request, "This is not a digital file product. Assets are only for FILE listings.")

return redirect("products:seller_list")

if request.method == "POST":

form = DigitalAssetUploadForm(request.POST, request.FILES)

if form.is_valid():

asset: DigitalAsset = form.save(commit=False)

asset.product = product

asset.save()

messages.success(request, "Digital asset uploaded.")

return redirect("products:seller_assets", pk=product.pk)

else:

form = DigitalAssetUploadForm()

assets = product.digital_assets.all().order_by("id")

return render(

request,

"products/seller/product_assets.html",

{"product": product, "form": form, "assets": assets},

)

@seller_required

@stripe_ready_required

def seller_product_asset_delete(request, pk: int, asset_id: int):

product = _get_owned_product_or_404(request, pk)

asset = get_object_or_404(DigitalAsset, pk=asset_id, product=product)

if request.method == "POST":

asset.delete()

messages.success(request, "Digital asset deleted.")

return redirect("products:seller_assets", pk=product.pk)

return redirect("products:seller_assets", pk=product.pk)

@seller_required

@stripe_ready_required

def seller_product_toggle_active(request, pk: int):

product = _get_owned_product_or_404(request, pk)

if request.method == "POST":

product.is_active = not product.is_active

product.save(update_fields=["is_active"])

messages.success(request, f"Listing is now {'active' if product.is_active else 'inactive'}.")

return redirect("products:seller_list")

products.views

```
from __future__ import annotations
from datetime import timedelta
from django.db.models import Avg, Count, F, FloatField, Q, Value
from django.db.models.functions import Coalesce
from django.http import HttpRequest, HttpResponse
from django.shortcuts import get_object_or_404, redirect, render
from django.utils import timezone
from orders.models import Order
from payments.models import SellerStripeAccount
from products.permissions import is_owner_user
from .models import Product, ProductEngagementEvent
MIN_REVIEWS_TOP_RATED = 3
TRENDING_WINDOW_DAYS = 30
VIEW_THROTTLE_MINUTES = 10
CLICK_THROTTLE_MINUTES = 5
TRENDING_BADGE_TOP_N = 12
def _base_qs():
return (
Product.objects.filter(is_active=True)
.select_related("category", "seller")
.prefetch_related("images")
)
def _annotate_rating(qs):
qs = qs.annotate(
avg_rating=Coalesce(Avg("reviews__rating"), Value(0.0), output_field=FloatField()),
review_count=Coalesce(Count("reviews", distinct=True), Value(0)),
)
# Seller reputation (purchased-only seller reviews)
```

```
qs = qs.annotate(
seller_avg_rating=Coalesce(
Avg("seller__seller_reviews_received__rating"),
Value(0.0),
output_field=FloatField(),
),
seller_review_count=Coalesce(
Count("seller__seller_reviews_received", distinct=True),
Value(0),
),
)
return qs
def _annotate_trending(qs, *, since_days: int = TRENDING_WINDOW_DAYS):
since = timezone.now() - timedelta(days=since_days)
recent_purchases = Count(
"order_items",
filter=Q(
order_items__order__status=Order.Status.PAID,
order_items__order__paid_at__isnull=False,
order_items__order__paid_at__gte=since,
),
distinct=True,
)
recent_reviews = Count(
"reviews",
filter=Q(reviews__created_at__gte=since),
distinct=True,
)
```

```
recent_views = Count(
"engagement_events",
filter=Q(
engagement_events__event_type=ProductEngagementEvent.EventType.VIEW,
engagement_events__created_at__gte=since,
),
distinct=True,
)
recent_clicks = Count(
"engagement_events",
filter=Q(
engagement_events__event_type=ProductEngagementEvent.EventType.CLICK,
engagement_events__created_at__gte=since,
),
distinct=True,
)
recent_add_to_cart = Count(
"engagement_events",
filter=Q(
engagement_events__event_type=ProductEngagementEvent.EventType.ADD_TO_CART,
engagement_events__created_at__gte=since,
),
distinct=True,
)
qs = qs.annotate(
recent_purchases=Coalesce(recent_purchases, Value(0)),
recent_reviews=Coalesce(recent_reviews, Value(0)),
recent_views=Coalesce(recent_views, Value(0)),
```

```
recent_clicks=Coalesce(recent_clicks, Value(0)),
recent_add_to_cart=Coalesce(recent_add_to_cart, Value(0)),
)
qs = qs.annotate(
trending_score=(
Coalesce(F("recent_purchases"), Value(0)) * Value(6.0)
+ Coalesce(F("recent_add_to_cart"), Value(0)) * Value(3.0)
+ Coalesce(F("recent_clicks"), Value(0)) * Value(1.25)
+ Coalesce(F("recent_reviews"), Value(0)) * Value(2.0)
+ Coalesce(F("recent_views"), Value(0)) * Value(0.25)
+ Coalesce(F("avg_rating"), Value(0.0)) * Value(1.0)
)
)
return qs
def _apply_trending_badge_flag(products: list[Product], *, computed_ids: set[int] | None = None) -> None:
computed_ids = computed_ids or set()
for p in products:
p.trending_badge = bool(getattr(p, "is_trending", False) or (p.id in computed_ids))
def _seller_can_sell(product: Product) -> bool:
try:
if product.seller and is_owner_user(product.seller):
return True
except Exception:
pass
try:
acct = getattr(product.seller, "stripe_connect", None)
if acct is not None:
return bool(acct.is_ready)
```

except Exception:

pass

try:

if not product.seller_id:

return False

return SellerStripeAccount.objects.filter(

user_id=product.seller_id,

stripe_account_id__gt="",

details_submitted=True,

charges_enabled=True,

payouts_enabled=True,

).exists()

except Exception:

return False

def _product_list_common(request: HttpRequest, *, kind: str | None, page_title: str) -> HttpResponse:

qs = _base_qs()

if kind in (Product.Kind.MODEL, Product.Kind.FILE):

qs = qs.filter(kind=kind)

if not kind:

kind_filter = (request.GET.get("kind") or "").strip().upper()

if kind_filter in (Product.Kind.MODEL, Product.Kind.FILE):

qs = qs.filter(kind=kind_filter)

q = (request.GET.get("q") or "").strip()

if q:

qs = qs.filter(Q(title__icontains=q) | Q(short_description__icontains=q) | Q(description__icontains=q))

sort = (request.GET.get("sort") or "new").strip().lower()

qs = _annotate_rating(qs)

trending_fallback = False

top_fallback = False

computed_ids: set[int] = set()

if sort == "trending":

qs = _annotate_trending(qs, since_days=TRENDING_WINDOW_DAYS)

qs = qs.order_by("-trending_score", "-avg_rating", "-created_at")

top_rows = list(qs.filter(trending_score__gt=0).values_list("id", flat=True)[:TRENDING_BADGE_TOP_N])

computed_ids = set(top_rows)

elif sort == "top":

filtered = qs.filter(review_count__gte=MIN_REVIEWS_TOP_RATED).order_by(

"-avg_rating", "-review_count", "-created_at"

)

first = list(filtered.values_list("id", flat=True)[:1])

if first:

qs = filtered

top_fallback = False

else:

qs = qs.order_by("-avg_rating", "-review_count", "-created_at")

top_fallback = True

else:

qs = qs.order_by("-created_at")

products = list(qs)

if sort == "trending":

any_signal = any(getattr(p, "trending_score", 0) > 0 for p in products)

trending_fallback = not any_signal

_apply_trending_badge_flag(products, computed_ids=computed_ids)

for p in products:

p.can_buy = _seller_can_sell(p)

return render(

request,

"products/product_list.html",

{

"products": products,

"q": q,

"kind": (kind or (request.GET.get("kind") or "")).strip().upper(),

"page_title": page_title,

"sort": sort,

"min_reviews_top_rated": MIN_REVIEWS_TOP_RATED,

"trending_fallback": trending_fallback,

"top_fallback": top_fallback,

},

)

def product_list(request: HttpRequest) -> HttpResponse:

return _product_list_common(request, kind=None, page_title="Browse Products")

def models_list(request: HttpRequest) -> HttpResponse:

return _product_list_common(request, kind=Product.Kind.MODEL, page_title="Browse 3D Models")

def files_list(request: HttpRequest) -> HttpResponse:

return _product_list_common(request, kind=Product.Kind.FILE, page_title="Browse 3D Files")

def _log_event_throttled(request: HttpRequest, *, product: Product, event_type: str, minutes: int) -> None:

try:

key = f"hc3_event_{event_type.lower()}_{product.id}"

now = timezone.now()

last_iso = request.session.get(key)

if last_iso:

try:

last_dt = timezone.datetime.fromisoformat(last_iso)

if timezone.is_naive(last_dt):

```
last_dt = timezone.make_aware(last_dt, timezone.get_current_timezone())
if now - last_dt < timedelta(minutes=minutes):
return
except Exception:
pass
ProductEngagementEvent.objects.create(product=product, event_type=event_type)
request.session[key] = now.isoformat()
except Exception:
return
def product_go(request: HttpRequest, pk: int, slug: str) -> HttpResponse:
product = get_object_or_404(
Product.objects.filter(is_active=True).select_related("category", "seller"),
pk=pk,
slug=slug,
)
_log_event_throttled(
request,
product=product,
event_type=ProductEngagementEvent.EventType.CLICK,
minutes=5,
)
return redirect("products:detail", pk=product.pk, slug=product.slug)
def product_detail(request: HttpRequest, pk: int, slug: str) -> HttpResponse:
product = get_object_or_404(
Product.objects.filter(is_active=True)
.select_related("category", "seller")
.prefetch_related("images"),
pk=pk,
```

slug=slug,

)

_log_event_throttled(

request,

product=product,

event_type=ProductEngagementEvent.EventType.VIEW,

minutes=10,

)

can_buy = _seller_can_sell(product)

from reviews.models import Review, SellerReview

review_qs = Review.objects.filter(product=product).select_related("buyer").order_by("-created_at")

summary = review_qs.aggregate(avg=Avg("rating"), count=Count("id"))

avg_rating = summary.get("avg") or 0

review_count = summary.get("count") or 0

recent_reviews = list(review_qs[:5])

seller_qs = SellerReview.objects.filter(seller_id=product.seller_id)

seller_summary = seller_qs.aggregate(avg=Avg("rating"), count=Count("id"))

seller_avg_rating = seller_summary.get("avg") or 0

seller_review_count = seller_summary.get("count") or 0

more_like_this = (

_base_qs()

.filter(kind=product.kind)

.exclude(pk=product.pk)

.filter(is_active=True)

)

same_cat = more_like_this.filter(category=product.category)

if same_cat.exists():

more_like_this = same_cat

```
more_like_this = _annotate_rating(more_like_this).order_by("-created_at")[:8]
more_like_this_list = list(more_like_this)
_apply_trending_badge_flag(more_like_this_list, computed_ids=set())
# ------------------------
# Q&A Tab (A)
# ------------------------
from qa.models import ProductQuestionThread
qa_threads = (
ProductQuestionThread.objects.filter(product=product, deleted_at__isnull=True)
.select_related("buyer", "product", "product__seller")
.prefetch_related("messages", "messages__author")
.order_by("-updated_at", "-created_at")
)
qa_threads_list = list(qa_threads[:20])
qa_thread_count = ProductQuestionThread.objects.filter(product=product, deleted_at__isnull=True).count()
return render(
request,
"products/product_detail.html",
{
"product": product,
"more_like_this": more_like_this_list,
"avg_rating": avg_rating,
"review_count": review_count,
"recent_reviews": recent_reviews,
"seller_avg_rating": seller_avg_rating,
"seller_review_count": seller_review_count,
"can_buy": can_buy,
# Q&A
```

"qa_threads": qa_threads_list,

"qa_thread_count": qa_thread_count,

},

)

qa.admin

from django.contrib import admin

from .models import ProductQuestionMessage, ProductQuestionReport, ProductQuestionThread

@admin.register(ProductQuestionThread)

class ProductQuestionThreadAdmin(admin.ModelAdmin):

list_display = ("id", "product", "buyer", "created_at", "updated_at", "deleted_at")

search_fields = ("product__title", "buyer__username", "subject")

list_filter = ("created_at", "deleted_at")

raw_id_fields = ("product", "buyer")

@admin.register(ProductQuestionMessage)

class ProductQuestionMessageAdmin(admin.ModelAdmin):

list_display = ("id", "thread", "author", "created_at", "deleted_at")

search_fields = ("thread__product__title", "author__username", "body")

list_filter = ("created_at", "deleted_at")

raw_id_fields = ("thread", "author", "deleted_by")

@admin.register(ProductQuestionReport)

class ProductQuestionReportAdmin(admin.ModelAdmin):

list_display = ("id", "status", "reason", "message", "reporter", "created_at")

search_fields = ("message__body", "reporter__username", "details")

list_filter = ("status", "reason", "created_at")

raw_id_fields = ("message", "reporter", "resolved_by")

qa.apps

from django.apps import AppConfig

class QaConfig(AppConfig):

default_auto_field = "django.db.models.BigAutoField"

name = "qa"

verbose_name = "Product Q&A"

qa.forms

from __future__ import annotations

from django import forms

from .models import ProductQuestionReport

class ThreadCreateForm(forms.Form):

subject = forms.CharField(

required=False,

max_length=180,

widget=forms.TextInput(attrs={"class": "form-control", "placeholder": "Optional subject"}),

)

body = forms.CharField(

required=True,

widget=forms.Textarea(

attrs={"class": "form-control", "rows": 3, "placeholder": "Ask a question..."}

),

)

class ReplyForm(forms.Form):

body = forms.CharField(

required=True,

widget=forms.Textarea(

attrs={"class": "form-control", "rows": 2, "placeholder": "Write a reply..."}

),

)

class ReportForm(forms.Form):

reason = forms.ChoiceField(

```
choices=ProductQuestionReport.Reason.choices,
widget=forms.Select(attrs={"class": "form-select"}),
)
details = forms.CharField(
required=False,
widget=forms.Textarea(
attrs={"class": "form-control", "rows": 2, "placeholder": "Optional details (recommended)"}
),
)
qa.models
from __future__ import annotations
from datetime import timedelta
from django.conf import settings
from django.db import models
from django.utils import timezone
class ProductQuestionThread(models.Model):
"""A buyer-initiated Q&A thread for a product.
Visibility: displayed on the product page.
Posting: logged-in users only.
Reply permissions: only thread.buyer and product.seller.
"""
product = models.ForeignKey(
"products.Product",
on_delete=models.CASCADE,
related_name="qa_threads",
)
buyer = models.ForeignKey(
settings.AUTH_USER_MODEL,
```

on_delete=models.PROTECT,

related_name="qa_threads",

help_text="Thread starter (buyer).",

)

subject = models.CharField(max_length=180, blank=True, default="")

created_at = models.DateTimeField(default=timezone.now, db_index=True)

updated_at = models.DateTimeField(auto_now=True)

Soft delete (primarily for staff cleanup). When a product is unlisted, product detail is hidden anyway.

deleted_at = models.DateTimeField(null=True, blank=True)

class Meta:

ordering = ["-created_at"]

indexes = [

models.Index(fields=["product", "-created_at"]),

models.Index(fields=["buyer", "-created_at"]),

models.Index(fields=["-created_at"]),

]

def __str__(self) -> str:

return f"QAThread<{self.pk}> product={self.product_id} buyer={self.buyer_id}"

@property

def is_deleted(self) -> bool:

return bool(self.deleted_at)

class ProductQuestionMessage(models.Model):

"""A message within a thread.

Soft-delete rules (locked spec):

- author can delete within 30 minutes

- after 30 minutes: staff only (upon request)

Reports do NOT auto-hide (v1). Staff reviews via queue.

"""

DELETE_WINDOW_MINUTES = 30

thread = models.ForeignKey(

ProductQuestionThread,

on_delete=models.CASCADE,

related_name="messages",

)

author = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.PROTECT,

related_name="qa_messages",

)

body = models.TextField()

created_at = models.DateTimeField(default=timezone.now, db_index=True)

soft delete

deleted_at = models.DateTimeField(null=True, blank=True)

deleted_by = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.SET_NULL,

null=True,

blank=True,

related_name="qa_messages_deleted",

)

class Meta:

ordering = ["created_at"]

indexes = [

models.Index(fields=["thread", "created_at"]),

models.Index(fields=["author", "-created_at"]),

models.Index(fields=["-created_at"]),

]

def __str__(self) -> str:

return f"QAMessage<{self.pk}> thread={self.thread_id} author={self.author_id}"

@property

def is_deleted(self) -> bool:

return bool(self.deleted_at)

def can_author_delete_now(self) -> bool:

if self.is_deleted:

return False

return timezone.now() - self.created_at <= timedelta(minutes=self.DELETE_WINDOW_MINUTES)

class ProductQuestionReport(models.Model):

"""Report record for a Q&A message.

Locked spec:

- dropdown reasons + optional text

- never auto-hide in v1

- staff queue for review

"""

class Reason(models.TextChoices):

SPAM = "spam", "Spam"

HARASSMENT = "harassment", "Harassment"

HATE = "hate", "Hate"

VIOLENCE = "violence", "Violence"

SEXUAL = "sexual", "Sexual content"

ILLEGAL = "illegal", "Illegal content"

OTHER = "other", "Other"

class Status(models.TextChoices):

OPEN = "open", "Open"

RESOLVED = "resolved", "Resolved"

message = models.ForeignKey(

ProductQuestionMessage,

on_delete=models.CASCADE,

related_name="reports",

)

reporter = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.PROTECT,

related_name="qa_reports",

)

reason = models.CharField(max_length=24, choices=Reason.choices)

details = models.TextField(blank=True, default="")

status = models.CharField(max_length=12, choices=Status.choices, default=Status.OPEN, db_index=True)

created_at = models.DateTimeField(default=timezone.now, db_index=True)

resolved_at = models.DateTimeField(null=True, blank=True)

resolved_by = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.SET_NULL,

null=True,

blank=True,

related_name="qa_reports_resolved",

)

class Meta:

ordering = ["-created_at"]

indexes = [

models.Index(fields=["status", "-created_at"]),

models.Index(fields=["reporter", "-created_at"]),

models.Index(fields=["-created_at"]),

]

def __str__(self) -> str:

return f"QAReport<{self.pk}> {self.status}"

qa.services

from __future__ import annotations

from dataclasses import dataclass

from django.contrib.auth import get_user_model

from django.core.exceptions import PermissionDenied, ValidationError

from django.db import transaction

from django.utils import timezone

from products.models import Product

from .models import ProductQuestionMessage, ProductQuestionReport, ProductQuestionThread

User = get_user_model()

def _is_staff(user) -> bool:

return bool(getattr(user, "is_staff", False) or getattr(user, "is_superuser", False))

def _thread_participants(thread: ProductQuestionThread) -> tuple[int, int]:

buyer_id = int(thread.buyer_id)

seller_id = int(thread.product.seller_id)

return buyer_id, seller_id

def can_post_in_thread(*, user, thread: ProductQuestionThread) -> bool:

if not user or not getattr(user, "is_authenticated", False):

return False

if _is_staff(user):

return True

buyer_id, seller_id = _thread_participants(thread)

return int(user.id) in {buyer_id, seller_id}

def can_create_thread(*, user, product: Product) -> bool:

return bool(user and getattr(user, "is_authenticated", False) and product and product.is_active)

@dataclass(frozen=True)

class ThreadCreateResult:

thread: ProductQuestionThread

first_message: ProductQuestionMessage

@transaction.atomic

def create_thread(*, product: Product, buyer, subject: str, body: str) -> ThreadCreateResult:

if not can_create_thread(user=buyer, product=product):

raise PermissionDenied("You must be logged in to ask a question.")

subject = (subject or "").strip()[:180]

body = (body or "").strip()

if not body:

raise ValidationError("Message cannot be blank.")

thread = ProductQuestionThread.objects.create(

product=product,

buyer=buyer,

subject=subject,

)

msg = ProductQuestionMessage.objects.create(

thread=thread,

author=buyer,

body=body,

)

return ThreadCreateResult(thread=thread, first_message=msg)

@transaction.atomic

def add_reply(*, thread: ProductQuestionThread, author, body: str) -> ProductQuestionMessage:

if not can_post_in_thread(user=author, thread=thread):

raise PermissionDenied("You do not have permission to reply in this thread.")

body = (body or "").strip()

if not body:

raise ValidationError("Reply cannot be blank.")

msg = ProductQuestionMessage.objects.create(thread=thread, author=author, body=body)

keep thread fresh

ProductQuestionThread.objects.filter(pk=thread.pk).update(updated_at=timezone.now())

return msg

@transaction.atomic

def soft_delete_message(*, message: ProductQuestionMessage, actor) -> None:

if message.is_deleted:

return

if not actor or not getattr(actor, "is_authenticated", False):

raise PermissionDenied("Not allowed.")

is_author = int(actor.id) == int(message.author_id)

if is_author and message.can_author_delete_now():

pass

elif _is_staff(actor):

pass

else:

raise PermissionDenied("This message can only be deleted by its author within 30 minutes, or by staff.")

message.deleted_at = timezone.now()

message.deleted_by = actor

message.save(update_fields=["deleted_at", "deleted_by"])

@transaction.atomic

def create_report(*, message: ProductQuestionMessage, reporter, reason: str, details: str = "") -> ProductQuestionReport:

if not reporter or not getattr(reporter, "is_authenticated", False):

raise PermissionDenied("You must be logged in to report a message.")

if message.is_deleted:

raise ValidationError("Cannot report a deleted message.")

```
reason = (reason or "").strip()
details = (details or "").strip()
report = ProductQuestionReport.objects.create(
message=message,
reporter=reporter,
reason=reason,
details=details,
)
return report
@transaction.atomic
def resolve_report(*, report: ProductQuestionReport, actor) -> None:
if not _is_staff(actor):
raise PermissionDenied("Staff only.")
if report.status == ProductQuestionReport.Status.RESOLVED:
return
report.status = ProductQuestionReport.Status.RESOLVED
report.resolved_at = timezone.now()
report.resolved_by = actor
report.save(update_fields=["status", "resolved_at", "resolved_by"])
qa.urls
from django.urls import path
from . import views
app_name = "qa"
urlpatterns = [
path("product/<int:product_id>/threads/new/", views.thread_create, name="thread_create"),
path("threads/<int:thread_id>/reply/", views.reply_create, name="reply_create"),
path("messages/<int:message_id>/delete/", views.message_delete, name="message_delete"),
path("messages/<int:message_id>/report/", views.message_report, name="message_report"),
```

staff queue

path("staff/reports/", views.staff_reports_queue, name="staff_reports"),

path("staff/reports/<int:report_id>/resolve/", views.staff_resolve_report, name="staff_report_resolve"),

]

qa.views

from __future__ import annotations

from django.contrib import messages

from django.contrib.auth.decorators import login_required, user_passes_test

from django.http import Http404

from django.shortcuts import get_object_or_404, redirect, render

from django.urls import reverse

from django.views.decorators.http import require_POST

from products.models import Product

from .forms import ReplyForm, ReportForm, ThreadCreateForm

from .models import ProductQuestionMessage, ProductQuestionReport, ProductQuestionThread

from .services import add_reply, create_report, create_thread, resolve_report, soft_delete_message

def _is_staff(user) -> bool:

return bool(getattr(user, "is_staff", False) or getattr(user, "is_superuser", False))

@require_POST

@login_required

def thread_create(request, product_id: int):

product = get_object_or_404(Product.objects.filter(is_active=True), pk=product_id)

form = ThreadCreateForm(request.POST)

if not form.is_valid():

messages.error(request, "Please correct the form.")

return redirect(product.get_absolute_url() + "#qa")

try:

create_thread(

product=product,

buyer=request.user,

subject=form.cleaned_data.get("subject", ""),

body=form.cleaned_data["body"],

)

messages.success(request, "Question posted.")

except Exception as e:

messages.error(request, str(e) or "Unable to post question.")

return redirect(product.get_absolute_url() + "#qa")

@require_POST

@login_required

def reply_create(request, thread_id: int):

thread = get_object_or_404(

ProductQuestionThread.objects.select_related("product", "product__seller", "buyer"),

pk=thread_id,

deleted_at__isnull=True,

)

form = ReplyForm(request.POST)

if not form.is_valid():

messages.error(request, "Please correct the reply.")

return redirect(thread.product.get_absolute_url() + "#qa")

try:

add_reply(thread=thread, author=request.user, body=form.cleaned_data["body"])

messages.success(request, "Reply posted.")

except Exception as e:

messages.error(request, str(e) or "Unable to reply.")

return redirect(thread.product.get_absolute_url() + "#qa")

@require_POST

```
@login_required
def message_delete(request, message_id: int):
msg = get_object_or_404(
ProductQuestionMessage.objects.select_related("thread", "thread__product"),
pk=message_id,
)
try:
soft_delete_message(message=msg, actor=request.user)
messages.success(request, "Message deleted.")
except Exception as e:
messages.error(request, str(e) or "Unable to delete message.")
return redirect(msg.thread.product.get_absolute_url() + "#qa")
@require_POST
@login_required
def message_report(request, message_id: int):
msg = get_object_or_404(
ProductQuestionMessage.objects.select_related("thread", "thread__product"),
pk=message_id,
deleted_at__isnull=True,
)
form = ReportForm(request.POST)
if not form.is_valid():
messages.error(request, "Please correct the report.")
return redirect(msg.thread.product.get_absolute_url() + "#qa")
try:
create_report(
message=msg,
reporter=request.user,
```

```
reason=form.cleaned_data["reason"],
details=form.cleaned_data.get("details", ""),
)
messages.success(request, "Report submitted. Staff will review it.")
except Exception as e:
messages.error(request, str(e) or "Unable to submit report.")
return redirect(msg.thread.product.get_absolute_url() + "#qa")
@user_passes_test(_is_staff)
def staff_reports_queue(request):
qs = (
ProductQuestionReport.objects.select_related(
"message",
"message__thread",
"message__thread__product",
"reporter",
)
.filter(status=ProductQuestionReport.Status.OPEN)
.order_by("-created_at")
)
return render(request, "qa/staff_reports_queue.html", {"reports": qs})
@user_passes_test(_is_staff)
@require_POST
def staff_resolve_report(request, report_id: int):
report = get_object_or_404(ProductQuestionReport.objects.select_related("message"), pk=report_id)
try:
resolve_report(report=report, actor=request.user)
messages.success(request, "Report resolved.")
except Exception as e:
```

messages.error(request, str(e) or "Unable to resolve report.")

Bounce back to queue

return redirect("qa:staff_reports")

refunds.admin

refunds/admin.py

from __future__ import annotations

from django.contrib import admin, messages

from django.urls import reverse

from django.utils.html import format_html

from .models import RefundRequest

@admin.register(RefundRequest)

class RefundRequestAdmin(admin.ModelAdmin):

list_display = (

"id",

"status",

"reason",

"order_link",

"order_item_link",

"seller",

"buyer_or_guest",

"total_refund_display",

"created_at",

)

list_filter = ("status", "reason", "created_at", "seller")

search_fields = (

"id",

"order__id",

"order_item__id",

"seller__username",

"buyer__username",

"requester_email",

"stripe_refund_id",

)

ordering = ("-created_at",)

actions = ("admin_trigger_refund",)

readonly_fields = (

"id",

"order",

"order_item",

"seller",

"buyer",

"requester_email",

"reason",

"notes",

"status",

"line_subtotal_cents_snapshot",

"tax_cents_allocated_snapshot",

"shipping_cents_allocated_snapshot",

"total_refund_cents_snapshot",

"stripe_refund_id",

"refunded_at",

"seller_decided_at",

"seller_decision_note",

"created_at",

"updated_at",

)

```
fieldsets = (
("Identity", {"fields": ("id", "status", "reason", "created_at", "updated_at")}),
("Parties", {"fields": ("order", "order_item", "seller", "buyer", "requester_email")}),
("Buyer request", {"fields": ("notes",)}),
(
"Snapshots (source of truth)",
{
"fields": (
"line_subtotal_cents_snapshot",
"tax_cents_allocated_snapshot",
"shipping_cents_allocated_snapshot",
"total_refund_cents_snapshot",
)
},
),
("Seller decision", {"fields": ("seller_decided_at", "seller_decision_note")}),
("Stripe", {"fields": ("stripe_refund_id", "refunded_at")}),
)
def order_link(self, obj: RefundRequest) -> str:
url = reverse("admin:orders_order_change", args=[obj.order_id])
return format_html('<a href="{}">{}</a>', url, obj.order_id)
order_link.short_description = "Order"
def order_item_link(self, obj: RefundRequest) -> str:
url = reverse("admin:orders_orderitem_change", args=[obj.order_item_id])
return format_html('<a href="{}">{}</a>', url, obj.order_item_id)
order_item_link.short_description = "Order item"
def buyer_or_guest(self, obj: RefundRequest) -> str:
if obj.buyer_id:
```

return getattr(obj.buyer, "username", str(obj.buyer_id))

return f"Guest ({(obj.requester_email or '').strip()})"

buyer_or_guest.short_description = "Buyer"

def total_refund_display(self, obj: RefundRequest) -> str:

cents = int(obj.total_refund_cents_snapshot or 0)

return f"${cents / 100:.2f}"

total_refund_display.short_description = "Refund total"

@admin.action(description="Trigger Stripe refund (DANGEROUS) for APPROVED requests")

def admin_trigger_refund(self, request, queryset):

"""

Safety valve:

- Only APPROVED, not already refunded

- Uses service layer so invariants stay centralized

"""

from .services import trigger_refund # local import

count_ok = 0

count_skip = 0

for rr in queryset.select_related("order", "seller"):

if rr.status != RefundRequest.Status.APPROVED or rr.stripe_refund_id or rr.refunded_at:

count_skip += 1

continue

try:

trigger_refund(rr=rr, actor_user=request.user, allow_staff_safety_valve=True)

count_ok += 1

except Exception as e:

count_skip += 1

messages.error(request, f"Refund {rr.pk}: {e}")

if count_ok:

messages.success(request, f"Processed {count_ok} refund(s).")

if count_skip:

messages.info(request, f"Skipped {count_skip} refund(s).")

refunds.apps

from __future__ import annotations

from django.apps import AppConfig

class RefundsConfig(AppConfig):

default_auto_field = "django.db.models.BigAutoField"

name = "refunds"

verbose_name = "Refunds"

refunds.forms

from __future__ import annotations

from django import forms

from .models import RefundRequest

class RefundRequestCreateForm(forms.Form):

reason = forms.ChoiceField(

choices=RefundRequest.Reason.choices,

widget=forms.Select(attrs={"class": "form-select"}),

)

notes = forms.CharField(

required=False,

widget=forms.Textarea(

attrs={

"class": "form-control",

"rows": 3,

"placeholder": "Optional details (recommended)",

}

),

```
)
# Used only for guest confirmation in the create view
guest_email = forms.EmailField(
required=False,
widget=forms.EmailInput(attrs={"class": "form-control", "placeholder": "you@example.com"}),
)
class SellerDecisionForm(forms.Form):
decision_note = forms.CharField(
required=False,
widget=forms.Textarea(
attrs={
"class": "form-control",
"rows": 3,
"placeholder": "Optional note to buyer",
}
),
)
refunds.models
from __future__ import annotations
import uuid
from dataclasses import dataclass
from django.conf import settings
from django.core.exceptions import ValidationError
from django.db import models
from django.utils import timezone
class RefundRequest(models.Model):
"""
Refund requests are FULL refunds per PHYSICAL line item only (per locked spec).
```

Digital products are never refundable in v1.

"""

class Status(models.TextChoices):

REQUESTED = "requested", "Requested"

APPROVED = "approved", "Approved"

DECLINED = "declined", "Declined"

REFUNDED = "refunded", "Refunded"

CANCELED = "canceled", "Canceled"

class Reason(models.TextChoices):

DAMAGED = "damaged", "Item arrived damaged"

NOT_AS_DESCRIBED = "not_as_described", "Not as described"

LATE = "late", "Arrived too late"

WRONG_ITEM = "wrong_item", "Wrong item received"

OTHER = "other", "Other"

id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

Denormalize for easier queries + integrity

order = models.ForeignKey("orders.Order", on_delete=models.CASCADE, related_name="refund_requests")

order_item = models.OneToOneField(

"orders.OrderItem",

on_delete=models.CASCADE,

related_name="refund_request",

help_text="At most one refund request per order line item.",

)

Snapshots for permission + display (do NOT depend on product->seller later)

seller = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.PROTECT,

related_name="refund_requests_received",

)

buyer = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.PROTECT,

null=True,

blank=True,

related_name="refund_requests_made",

)

requester_email = models.EmailField(blank=True, default="")

reason = models.CharField(max_length=32, choices=Reason.choices)

notes = models.TextField(blank=True, default="")

status = models.CharField(max_length=16, choices=Status.choices, default=Status.REQUESTED)

Snapshot amounts at creation (FULL refund for this line item)

line_subtotal_cents_snapshot = models.PositiveIntegerField(default=0)

tax_cents_allocated_snapshot = models.PositiveIntegerField(default=0)

shipping_cents_allocated_snapshot = models.PositiveIntegerField(default=0)

total_refund_cents_snapshot = models.PositiveIntegerField(default=0)

Stripe refund tracking (partial refunds are allowed; we refund this line amount)

stripe_refund_id = models.CharField(max_length=255, blank=True, default="")

refunded_at = models.DateTimeField(null=True, blank=True)

seller_decided_at = models.DateTimeField(null=True, blank=True)

seller_decision_note = models.TextField(blank=True, default="")

created_at = models.DateTimeField(default=timezone.now, db_index=True)

updated_at = models.DateTimeField(auto_now=True)

class Meta:

indexes = [

models.Index(fields=["status", "-created_at"]),

models.Index(fields=["seller", "status", "-created_at"]),

models.Index(fields=["buyer", "status", "-created_at"]),

models.Index(fields=["order", "-created_at"]),

]

def __str__(self) -> str:

return f"RefundRequest<{self.pk}> {self.status}"

def clean(self) -> None:

super().clean()

Physical-only enforcement

oi = self.order_item

if oi.is_digital or not oi.requires_shipping:

raise ValidationError("Refund requests are only allowed for physical items.")

Digital products are non-refundable

try:

if oi.product.kind == oi.product.Kind.FILE:

raise ValidationError("Digital products are not refundable.")

except Exception:

If product/kind isn't available for some reason, fail open here;

physical-guard above is still the hard gate.

pass

Must be tied to the same order

if oi.order_id != self.order_id:

raise ValidationError("OrderItem must belong to the specified Order.")

@property

def is_decided(self) -> bool:

return self.status in {

self.Status.APPROVED,

self.Status.DECLINED,

self.Status.REFUNDED,

self.Status.CANCELED,

}

@property

def is_refundable_now(self) -> bool:

return self.status == self.Status.APPROVED and not self.stripe_refund_id and not self.refunded_at

@dataclass(frozen=True)

class AllocatedLineRefund:

line_subtotal_cents: int

tax_cents_allocated: int

shipping_cents_allocated: int

total_refund_cents: int

def as_dict(self) -> dict[str, int]:

return {

"line_subtotal_cents": int(self.line_subtotal_cents),

"tax_cents_allocated": int(self.tax_cents_allocated),

"shipping_cents_allocated": int(self.shipping_cents_allocated),

"total_refund_cents": int(self.total_refund_cents),

}

refunds.services

refunds/services.py

from __future__ import annotations

import logging

from decimal import Decimal, ROUND_HALF_UP

from django.core.exceptions import PermissionDenied, ValidationError

from django.db import transaction

from django.utils import timezone

from orders.models import Order, OrderEvent, OrderItem

from products.permissions import is_owner_user

```
from .models import AllocatedLineRefund, RefundRequest
from .stripe_service import create_stripe_refund_for_request
logger = logging.getLogger(__name__)
# ============================================================
# Allocation helpers
# ============================================================
def _safe_int(x) -> int:
try:
return int(x or 0)
except Exception:
return 0
def _allocate_tax_for_item(*, order: Order, item: OrderItem) -> int:
"""
Allocate order.tax_cents proportionally across ALL order items by line_total.
"""
total_tax = _safe_int(order.tax_cents)
if total_tax <= 0:
return 0
items = list(order.items.all())
if not items:
return 0
denom = sum(_safe_int(i.line_total_cents) for i in items)
if denom <= 0:
return 0
share = Decimal(_safe_int(item.line_total_cents)) / Decimal(denom)
alloc = (Decimal(total_tax) * share).quantize(Decimal("1"), rounding=ROUND_HALF_UP)
return max(0, int(alloc))
def _allocate_shipping_for_item(*, order: Order, item: OrderItem) -> int:
```

"""

Allocate order.shipping_cents across shippable (requires_shipping=True) items by line_total.

"""

total_shipping = _safe_int(order.shipping_cents)

if total_shipping <= 0:

return 0

shippable = [i for i in order.items.all() if bool(getattr(i, "requires_shipping", False))]

if not shippable:

return 0

denom = sum(_safe_int(i.line_total_cents) for i in shippable)

if denom <= 0:

return 0

share = Decimal(_safe_int(item.line_total_cents)) / Decimal(denom)

alloc = (Decimal(total_shipping) * share).quantize(Decimal("1"), rounding=ROUND_HALF_UP)

return max(0, int(alloc))

def compute_allocated_line_refund(*, order: Order, item: OrderItem) -> AllocatedLineRefund:

"""

FULL refund per physical line item:

- line subtotal = item.line_total_cents

- plus allocated tax

- plus allocated shipping (only across shippable lines)

"""

line_subtotal = _safe_int(item.line_total_cents)

tax_alloc = _allocate_tax_for_item(order=order, item=item)

ship_alloc = _allocate_shipping_for_item(order=order, item=item)

total = max(0, line_subtotal + tax_alloc + ship_alloc)

return AllocatedLineRefund(

line_subtotal_cents=line_subtotal,

tax_cents_allocated=tax_alloc,

shipping_cents_allocated=ship_alloc,

total_refund_cents=total,

)

==

Core service functions

==

@transaction.atomic

def create_refund_request(

*,

order: Order,

item: OrderItem,

requester_user,

requester_email: str,

reason: str,

notes: str = "",

) -> RefundRequest:

"""

Create a refund request for a PAID order + PHYSICAL line item only.

One refund request per item (OneToOne).

"""

if order.pk != item.order_id:

raise ValidationError("Order item does not belong to this order.")

if order.status != Order.Status.PAID or not getattr(order, "paid_at", None):

raise ValidationError("Refund requests are available only for paid orders.")

Physical-only enforcement (locked spec)

if bool(getattr(item, "is_digital", False)) or not bool(getattr(item, "requires_shipping", False)):

raise ValidationError("Refund requests are only allowed for physical items.")

Seller snapshot must exist

if not getattr(item, "seller_id", None):

raise ValidationError("Order item is missing seller snapshot.")

One per item (reverse OneToOne raises DoesNotExist)

try:

_ = item.refund_request

raise ValidationError("A refund request already exists for this item.")

except RefundRequest.DoesNotExist:

pass

except Exception:

raise ValidationError("Unable to verify existing refund status for this item.")

alloc = compute_allocated_line_refund(order=order, item=item)

rr = RefundRequest.objects.create(

order=order,

order_item=item,

seller_id=item.seller_id,

buyer_id=getattr(order, "buyer_id", None) or None,

requester_email=(requester_email or "").strip().lower(),

reason=reason,

notes=(notes or "").strip(),

status=RefundRequest.Status.REQUESTED,

line_subtotal_cents_snapshot=_safe_int(alloc.line_subtotal_cents),

tax_cents_allocated_snapshot=_safe_int(alloc.tax_cents_allocated),

shipping_cents_allocated_snapshot=_safe_int(alloc.shipping_cents_allocated),

total_refund_cents_snapshot=_safe_int(alloc.total_refund_cents),

)

rr.full_clean()

rr.save(update_fields=["updated_at"])

try:

OrderEvent.objects.create(

order=order,

type=OrderEvent.Type.WARNING,

message=f"Refund requested rr={rr.pk} item={item.pk} seller={item.seller_id}",

)

except Exception:

pass

return rr

@transaction.atomic

def seller_decide(*, rr: RefundRequest, seller_user, approve: bool, note: str = "") -> RefundRequest:

"""

Seller (or owner/staff) approves/declines.

"""

if rr.status != RefundRequest.Status.REQUESTED:

raise ValidationError("This refund request is not awaiting a decision.")

if not seller_user or not getattr(seller_user, "is_authenticated", False):

raise PermissionDenied("Authentication required.")

allow seller themselves, owner, or staff/superuser

if not (

is_owner_user(seller_user)

or bool(getattr(seller_user, "is_staff", False))

or bool(getattr(seller_user, "is_superuser", False))

):

if rr.seller_id != seller_user.id:

raise PermissionDenied("You do not have permission to decide this refund request.")

rr.status = RefundRequest.Status.APPROVED if approve else RefundRequest.Status.DECLINED

rr.seller_decided_at = timezone.now()

```
rr.seller_decision_note = (note or "").strip()
rr.save(update_fields=["status", "seller_decided_at", "seller_decision_note", "updated_at"])
try:
OrderEvent.objects.create(
order=rr.order,
type=OrderEvent.Type.WARNING,
message=f"Refund {rr.status} rr={rr.pk} by={seller_user.pk}",
)
except Exception:
pass
return rr
@transaction.atomic
def trigger_refund(*, rr: RefundRequest, actor_user, allow_staff_safety_valve: bool = True) -> RefundRequest:
"""
Trigger the Stripe refund after approval.
- Uses rr.total_refund_cents_snapshot as the source of truth.
"""
if rr.status != RefundRequest.Status.APPROVED:
raise ValidationError("Refund must be approved before it can be processed.")
if not rr.is_refundable_now:
raise ValidationError("This refund is not refundable right now.")
if not actor_user or not getattr(actor_user, "is_authenticated", False):
raise PermissionDenied("Authentication required.")
is_staff = bool(getattr(actor_user, "is_staff", False) or getattr(actor_user, "is_superuser", False))
is_owner = is_owner_user(actor_user)
if rr.seller_id == actor_user.id:
pass
elif is_owner:
```

```
pass
elif allow_staff_safety_valve and is_staff:
pass
else:
raise PermissionDenied("You do not have permission to process this refund.")
refund_id = create_stripe_refund_for_request(rr=rr)
rr.stripe_refund_id = refund_id
rr.refunded_at = timezone.now()
rr.status = RefundRequest.Status.REFUNDED
rr.save(update_fields=["stripe_refund_id", "refunded_at", "status", "updated_at"])
try:
OrderEvent.objects.create(
order=rr.order,
type=OrderEvent.Type.REFUNDED,
message=f"Refund processed rr={rr.pk} stripe_refund={refund_id}",
)
except Exception:
pass
return rr
refunds.stripe_service
# refunds/stripe_service.py
from __future__ import annotations
import stripe
from django.conf import settings
from .models import RefundRequest
def _init_stripe() -> None:
key = getattr(settings, "STRIPE_SECRET_KEY", None)
if not key:
```

raise RuntimeError("STRIPE_SECRET_KEY is not configured.")

stripe.api_key = key

def create_stripe_refund_for_request(*, rr: RefundRequest) -> str:

"""

Create a Stripe refund for this request.

Source of truth:

- refund amount = rr.total_refund_cents_snapshot

- target payment = rr.order.stripe_payment_intent_id

Idempotency:

- idempotency_key=f"refundreq-{rr.pk}"

"""

_init_stripe()

order = rr.order

payment_intent = (getattr(order, "stripe_payment_intent_id", "") or "").strip()

FREE checkouts have no Stripe refund

if payment_intent == "FREE":

raise ValueError("Order was a free checkout; no Stripe refund is possible.")

if not payment_intent:

raise ValueError("Order has no Stripe payment intent id; cannot refund.")

amount = int(getattr(rr, "total_refund_cents_snapshot", 0) or 0)

if amount <= 0:

raise ValueError("Refund amount must be > 0.")

refund = stripe.Refund.create(

payment_intent=payment_intent,

amount=amount,

metadata={

"refund_request_id": str(rr.pk),

"order_id": str(order.pk),

"order_item_id": str(rr.order_item_id),

"seller_id": str(rr.seller_id),

},

idempotency_key=f"refundreq-{rr.pk}",

)

refund_id = str(getattr(refund, "id", "") or "").strip()

if not refund_id:

raise ValueError("Stripe did not return a refund id.")

return refund_id

refunds.urls

refunds/urls.py

from __future__ import annotations

from django.urls import path

from . import views

app_name = "refunds"

urlpatterns = [

Buyer/Guest: create request (physical-only)

IMPORTANT: keep this ABOVE the "<uuid:refund_id>/" route to avoid URL shadowing.

path("new/<uuid:order_id>/<uuid:item_id>/", views.buyer_create, name="buyer_create"),

Seller: queue + detail + actions

path("seller/", views.seller_queue, name="seller_queue"),

path("seller/<uuid:refund_id>/", views.seller_detail, name="seller_detail"),

path("seller/<uuid:refund_id>/approve/", views.seller_approve, name="seller_approve"),

path("seller/<uuid:refund_id>/decline/", views.seller_decline, name="seller_decline"),

path("seller/<uuid:refund_id>/refund/", views.seller_trigger_refund, name="seller_trigger_refund"),

Staff safety valve

path("staff/", views.staff_queue, name="staff_queue"),

path("staff/<uuid:refund_id>/refund/", views.staff_trigger_refund, name="staff_trigger_refund"),

Buyer: list + detail (keep detail last)

path("", views.buyer_list, name="buyer_list"),

path("<uuid:refund_id>/", views.buyer_detail, name="buyer_detail"),

]

refunds.views

refunds/views.py

from __future__ import annotations

import logging

from django.contrib import messages

from django.contrib.auth.decorators import login_required, user_passes_test

from django.core.exceptions import PermissionDenied, ValidationError

from django.http import Http404, HttpRequest, HttpResponse

from django.shortcuts import get_object_or_404, redirect, render

from django.urls import reverse

from django.views.decorators.http import require_POST

from orders.models import Order, OrderItem

from products.permissions import is_owner_user, is_seller_user

from .forms import RefundRequestCreateForm, SellerDecisionForm

from .models import RefundRequest

from .services import create_refund_request, seller_decide, trigger_refund

logger = logging.getLogger(__name__)

==

Helpers

==

def _token_from_request(request: HttpRequest) -> str:

return (request.GET.get("t") or "").strip()

def _is_staff(user) -> bool:

return bool(getattr(user, "is_staff", False) or getattr(user, "is_superuser", False))

def _user_can_access_order(request: HttpRequest, order: Order) -> bool:

"""

Mirrors orders.views enforcement:

- staff/superuser always allowed

- buyer can access their own order

- guest can access via token ?t=<order.order_token>

"""

if request.user.is_authenticated and _is_staff(request.user):

return True

if getattr(order, "buyer_id", None):

return request.user.is_authenticated and request.user.id == order.buyer_id

t = _token_from_request(request)

return bool(t) and str(t) == str(getattr(order, "order_token", ""))

def _require_seller_or_staff(request: HttpRequest, rr: RefundRequest) -> None:

"""

Only:

- rr.seller

- owner

- staff

can act on seller-side refund request.

"""

user = request.user

if not user.is_authenticated:

raise Http404("Not found")

if is_owner_user(user) or _is_staff(user):

return

if rr.seller_id != user.id:

raise Http404("Not found")

```
def _redirect_order_detail(order: Order, token: str = "") -> str:
"""
Token-preserving redirect for guest orders.
"""
if getattr(order, "buyer_id", None):
return reverse("orders:detail", kwargs={"order_id": order.pk})
token = (token or "").strip()
if token:
return f"{reverse('orders:detail', kwargs={'order_id': order.pk})}?t={token}"
return reverse("orders:detail", kwargs={"order_id": order.pk})
def _order_token_for_redirect(request: HttpRequest, order: Order) -> str:
"""
Prefer token passed in request (?t=), else use order.order_token.
"""
t = _token_from_request(request)
if t:
return t
try:
return str(order.order_token)
except Exception:
return ""
def _get_item_refund_request(item: OrderItem) -> RefundRequest | None:
"""
Reverse OneToOne raises DoesNotExist; do not use hasattr().
"""
try:
return item.refund_request
except Exception:
```

return None

==

Buyer: list + detail

==

@login_required

def buyer_list(request: HttpRequest) -> HttpResponse:

qs = (

RefundRequest.objects.select_related(

"order",

"order_item",

"order_item__product",

"seller",

"buyer",

)

.filter(buyer=request.user)

.order_by("-created_at")

)

return render(request, "refunds/buyer_list.html", {"refunds": qs})

def buyer_detail(request: HttpRequest, refund_id) -> HttpResponse:

"""

Detail supports:

- logged-in buyer (rr.buyer)

- staff

- guest via valid order token (?t= on the underlying order)

"""

rr = get_object_or_404(

RefundRequest.objects.select_related(

"order",

"order_item",

"order_item__product",

"seller",

"buyer",

),

pk=refund_id,

)

Staff override

if request.user.is_authenticated and _is_staff(request.user):

return render(request, "refunds/buyer_detail.html", {"rr": rr})

Buyer path (logged-in)

if rr.buyer_id:

if not request.user.is_authenticated:

return redirect("accounts:login")

if rr.buyer_id != request.user.id:

raise Http404("Not found")

return render(request, "refunds/buyer_detail.html", {"rr": rr})

Guest path: must have token access to the underlying order.

order = rr.order

if not _user_can_access_order(request, order):

if order.buyer_id and not request.user.is_authenticated:

return redirect("accounts:login")

raise Http404("Not found")

return render(request, "refunds/buyer_detail.html", {"rr": rr})

==

Buyer/Guest: create request

==

def buyer_create(request: HttpRequest, order_id, item_id) -> HttpResponse:

"""

Create refund request for a PHYSICAL line item only.

Guests must access via tokenized order link and confirm checkout email matches order.guest_email.

"""

order = get_object_or_404(Order.objects.select_related("buyer"), pk=order_id)

token = _order_token_for_redirect(request, order)

if order.status != Order.Status.PAID or not getattr(order, "paid_at", None):

messages.info(request, "Refund requests are available only for paid orders.")

return redirect(_redirect_order_detail(order, token))

if not _user_can_access_order(request, order):

if order.buyer_id and not request.user.is_authenticated:

return redirect("accounts:login")

raise Http404("Not found")

item = get_object_or_404(

OrderItem.objects.select_related("product", "seller", "order"),

pk=item_id,

order=order,

)

existing = _get_item_refund_request(item)

if existing:

url = reverse("refunds:buyer_detail", kwargs={"refund_id": existing.pk})

if order.is_guest and token:

url = f"{url}?t={token}"

return redirect(url)

requester_user = request.user if request.user.is_authenticated else None

is_guest = bool(order.is_guest)

guest_email_prefill = (order.guest_email or "").strip().lower()

initial = {"guest_email": guest_email_prefill} if is_guest else None

form = RefundRequestCreateForm(request.POST or None, initial=initial)

Resolve requester email / permissions

if not is_guest:

Logged-in buyer only (or staff)

if not request.user.is_authenticated:

return redirect("accounts:login")

if request.user.id != order.buyer_id and not _is_staff(request.user):

raise Http404("Not found")

requester_email = (getattr(request.user, "email", "") or "").strip().lower()

else:

Guest: must confirm email == checkout email

requester_email = ""

if request.method == "POST":

requester_email = (form.data.get("guest_email") or "").strip().lower()

else:

requester_email = guest_email_prefill

if not requester_email or requester_email != guest_email_prefill:

if request.method == "POST":

form.add_error("guest_email", "Please confirm the email used at checkout.")

return render(

request,

"refunds/request_create.html",

{

"order": order,

"item": item,

"form": form,

"is_guest": True,

"guest_email": requester_email,

"order_token": token,

},

)

if request.method == "POST":

if form.is_valid():

try:

rr = create_refund_request(

order=order,

item=item,

requester_user=requester_user,

requester_email=requester_email,

reason=form.cleaned_data["reason"],

notes=form.cleaned_data.get("notes", "") or "",

)

except (ValidationError, PermissionDenied, ValueError) as e:

messages.error(request, str(e) or "Unable to create refund request.")

return redirect(_redirect_order_detail(order, token))

except Exception:

logger.exception("Refund request creation failed order=%s item=%s", order.pk, item.pk)

messages.error(request, "Unable to create refund request.")

return redirect(_redirect_order_detail(order, token))

messages.success(request, "Refund request submitted.")

url = reverse("refunds:buyer_detail", kwargs={"refund_id": rr.pk})

if order.is_guest and token:

url = f"{url}?t={token}"

return redirect(url)

return render(

request,

"refunds/request_create.html",

{

"order": order,

"item": item,

"form": form,

"is_guest": is_guest,

"guest_email": guest_email_prefill if is_guest else "",

"order_token": token,

},

)

==

Seller: queue + detail + actions

==

@login_required

def seller_queue(request: HttpRequest) -> HttpResponse:

user = request.user

if not (is_seller_user(user) or is_owner_user(user) or _is_staff(user)):

messages.info(request, "You don’t have access to seller refunds.")

return redirect("dashboards:consumer")

qs = (

RefundRequest.objects.select_related(

"order",

"order_item",

"order_item__product",

"seller",

"buyer",

)

.order_by("-created_at")

)

if not (is_owner_user(user) or _is_staff(user)):

qs = qs.filter(seller=user)

return render(request, "refunds/seller_queue.html", {"refunds": qs})

@login_required

def seller_detail(request: HttpRequest, refund_id) -> HttpResponse:

rr = get_object_or_404(

RefundRequest.objects.select_related(

"order",

"order_item",

"order_item__product",

"seller",

"buyer",

),

pk=refund_id,

)

_require_seller_or_staff(request, rr)

decision_form = SellerDecisionForm(initial={"decision_note": rr.seller_decision_note})

return render(request, "refunds/seller_detail.html", {"rr": rr, "decision_form": decision_form})

@login_required

@require_POST

def seller_approve(request: HttpRequest, refund_id) -> HttpResponse:

rr = get_object_or_404(

RefundRequest.objects.select_related("seller", "order"),

pk=refund_id,

)

_require_seller_or_staff(request, rr)

form = SellerDecisionForm(request.POST)

```
note = ""
if form.is_valid():
note = (form.cleaned_data.get("decision_note", "") or "").strip()
try:
seller_decide(rr=rr, seller_user=request.user, approve=True, note=note)
messages.success(request, "Refund approved. You can now trigger the Stripe refund.")
except (ValidationError, PermissionDenied, ValueError) as e:
messages.error(request, str(e) or "Unable to approve refund.")
except Exception:
logger.exception("Seller approve failed rr=%s user=%s", rr.pk, request.user.pk)
messages.error(request, "Unable to approve refund.")
return redirect("refunds:seller_detail", refund_id=rr.pk)
@login_required
@require_POST
def seller_decline(request: HttpRequest, refund_id) -> HttpResponse:
rr = get_object_or_404(
RefundRequest.objects.select_related("seller", "order"),
pk=refund_id,
)
_require_seller_or_staff(request, rr)
form = SellerDecisionForm(request.POST)
note = ""
if form.is_valid():
note = (form.cleaned_data.get("decision_note", "") or "").strip()
try:
seller_decide(rr=rr, seller_user=request.user, approve=False, note=note)
messages.success(request, "Refund declined.")
except (ValidationError, PermissionDenied, ValueError) as e:
```

```
messages.error(request, str(e) or "Unable to decline refund.")
except Exception:
logger.exception("Seller decline failed rr=%s user=%s", rr.pk, request.user.pk)
messages.error(request, "Unable to decline refund.")
return redirect("refunds:seller_detail", refund_id=rr.pk)
@login_required
@require_POST
def seller_trigger_refund(request: HttpRequest, refund_id) -> HttpResponse:
rr = get_object_or_404(
RefundRequest.objects.select_related("seller", "order"),
pk=refund_id,
)
_require_seller_or_staff(request, rr)
try:
trigger_refund(rr=rr, actor_user=request.user, allow_staff_safety_valve=True)
messages.success(request, "Refund processed via Stripe.")
except (ValidationError, PermissionDenied, ValueError) as e:
messages.error(request, str(e) or "Unable to process refund.")
except Exception:
logger.exception("Seller trigger refund failed rr=%s user=%s", rr.pk, request.user.pk)
messages.error(request, "Unable to process refund.")
return redirect("refunds:seller_detail", refund_id=rr.pk)
# ============================================================
# Staff: safety valve queue + action
# ============================================================
@user_passes_test(_is_staff)
def staff_queue(request: HttpRequest) -> HttpResponse:
qs = (
```

RefundRequest.objects.select_related(

"order",

"order_item",

"order_item__product",

"seller",

"buyer",

)

.order_by("-created_at")

)

return render(request, "refunds/staff_queue.html", {"refunds": qs})

@user_passes_test(_is_staff)

@require_POST

def staff_trigger_refund(request: HttpRequest, refund_id) -> HttpResponse:

rr = get_object_or_404(

RefundRequest.objects.select_related("seller", "order"),

pk=refund_id,

)

try:

trigger_refund(rr=rr, actor_user=request.user, allow_staff_safety_valve=True)

messages.success(request, "Refund processed via Stripe (staff safety valve).")

except (ValidationError, PermissionDenied, ValueError) as e:

messages.error(request, str(e) or "Unable to process refund.")

except Exception:

logger.exception("Staff trigger refund failed rr=%s staff=%s", rr.pk, request.user.pk)

messages.error(request, "Unable to process refund.")

return redirect("refunds:staff_queue")

reviews.admin

from django.contrib import admin

from .models import Review

@admin.register(Review)

class ReviewAdmin(admin.ModelAdmin):

list_display = ("id", "product", "buyer", "rating", "created_at")

list_filter = ("rating", "created_at")

search_fields = ("product__title", "buyer__username", "title", "body")

readonly_fields = ("created_at", "updated_at")

reviews.forms

from __future__ import annotations

from django import forms

from .models import Review, SellerReview

class ReviewForm(forms.ModelForm):

class Meta:

model = Review

fields = ["rating", "title", "body"]

widgets = {

"rating": forms.NumberInput(attrs={"min": 1, "max": 5, "class": "form-control"}),

"title": forms.TextInput(attrs={"class": "form-control"}),

"body": forms.Textarea(attrs={"class": "form-control", "rows": 5}),

}

class SellerReviewForm(forms.ModelForm):

class Meta:

model = SellerReview

fields = ["rating", "title", "body"]

widgets = {

"rating": forms.NumberInput(attrs={"min": 1, "max": 5, "class": "form-control"}),

"title": forms.TextInput(attrs={"class": "form-control"}),

"body": forms.Textarea(attrs={"class": "form-control", "rows": 5}),

```
}
reviews.models
from __future__ import annotations
from django.conf import settings
from django.core.validators import MaxValueValidator, MinValueValidator
from django.db import models
class Review(models.Model):
"""Buyer review for a purchased product."""
product = models.ForeignKey(
"products.Product",
on_delete=models.CASCADE,
related_name="reviews",
)
order_item = models.OneToOneField(
"orders.OrderItem",
on_delete=models.CASCADE,
related_name="review",
help_text="Enforces one review per purchased line item.",
)
buyer = models.ForeignKey(
settings.AUTH_USER_MODEL,
on_delete=models.CASCADE,
related_name="reviews",
)
rating = models.PositiveSmallIntegerField(
validators=[MinValueValidator(1), MaxValueValidator(5)],
help_text="1–5 stars",
)
```

title = models.CharField(max_length=120, blank=True, default="")

body = models.TextField(blank=True, default="")

created_at = models.DateTimeField(auto_now_add=True)

updated_at = models.DateTimeField(auto_now=True)

class Meta:

ordering = ["-created_at"]

indexes = [

models.Index(fields=["product", "created_at"]),

models.Index(fields=["buyer", "created_at"]),

models.Index(fields=["rating"]),

]

def __str__(self) -> str:

return f"Review<{self.product_id}> by {self.buyer_id} ({self.rating}/5)"

class SellerReview(models.Model):

"""Purchased-only rating for a seller.

Rules:

- Only an authenticated buyer (Order.buyer) can rate.

- Order must be PAID.

- Order must include at least one OrderItem for that seller.

We bind to the Order itself (not a specific item) so a buyer can leave ONE

seller rating per order for that seller.

"""

seller = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.CASCADE,

related_name="seller_reviews_received",

)

buyer = models.ForeignKey(

settings.AUTH_USER_MODEL,

on_delete=models.CASCADE,

related_name="seller_reviews_written",

)

order = models.ForeignKey(

"orders.Order",

on_delete=models.CASCADE,

related_name="seller_reviews",

)

rating = models.PositiveSmallIntegerField(

validators=[MinValueValidator(1), MaxValueValidator(5)],

help_text="1–5 stars",

)

title = models.CharField(max_length=120, blank=True, default="")

body = models.TextField(blank=True, default="")

created_at = models.DateTimeField(auto_now_add=True)

updated_at = models.DateTimeField(auto_now=True)

class Meta:

ordering = ["-created_at"]

constraints = [

models.UniqueConstraint(fields=["buyer", "seller", "order"], name="uniq_seller_review_per_order"),

]

indexes = [

models.Index(fields=["seller", "created_at"]),

models.Index(fields=["buyer", "created_at"]),

models.Index(fields=["rating"]),

]

def __str__(self) -> str:

return f"SellerReview<seller={self.seller_id} buyer={self.buyer_id} order={self.order_id}> ({self.rating}/5)"

reviews.services

from __future__ import annotations

from django.core.exceptions import PermissionDenied

from orders.models import Order, OrderItem

def get_reviewable_order_item_or_403(*, user, order_item_id: int) -> OrderItem:

"""Return an OrderItem the user is allowed to review, else raise PermissionDenied.

Rules:

- User must be authenticated.

- OrderItem must exist.

- The parent order must be PAID.

- The order's buyer must be the requesting user.

Notes:

- One review per OrderItem is enforced in models via OneToOneField.

"""

if not user or not getattr(user, "is_authenticated", False):

raise PermissionDenied("Authentication required")

item = (

OrderItem.objects.select_related("order", "product")

.filter(id=order_item_id)

.first()

)

if not item:

raise PermissionDenied("Order item not found")

order = item.order

if not order or order.status != Order.Status.PAID:

raise PermissionDenied("Order not paid")

if getattr(order, "buyer_id", None) != getattr(user, "id", None):

raise PermissionDenied("Not your order")

return item

def get_rateable_seller_order_or_403(*, user, order_id: int, seller_id: int) -> Order:

"""Return an Order the user can use to rate a seller, else raise PermissionDenied.

Rules:

- User must be authenticated.

- Order must exist and be PAID.

- Order.buyer must be the requesting user.

- Order must include at least one OrderItem with product.seller_id == seller_id.

"""

if not user or not getattr(user, "is_authenticated", False):

raise PermissionDenied("Authentication required")

order = Order.objects.filter(id=order_id).first()

if not order:

raise PermissionDenied("Order not found")

if order.status != Order.Status.PAID:

raise PermissionDenied("Order not paid")

if getattr(order, "buyer_id", None) != getattr(user, "id", None):

raise PermissionDenied("Not your order")

has_seller_item = (

OrderItem.objects.filter(order_id=order.id, product__seller_id=seller_id)

.exists()

)

if not has_seller_item:

raise PermissionDenied("Seller not in this order")

return order

reviews.urls

from django.urls import path

from . import views

app_name = "reviews"

urlpatterns = [

path("product/<int:product_id>/", views.product_reviews, name="product_reviews"),

path("order-item/<int:order_item_id>/new/", views.review_create_for_order_item, name="review_for_item"),

Purchased-only seller rating

path("seller/<int:order_id>/<int:seller_id>/new/", views.seller_review_create, name="seller_review_new"),

]

reviews.views

from __future__ import annotations

from django.contrib import messages

from django.core.exceptions import PermissionDenied

from django.http import Http404

from django.shortcuts import get_object_or_404, redirect, render

from django.views.decorators.http import require_http_methods

from .forms import ReviewForm, SellerReviewForm

from .models import Review, SellerReview

from .services import get_rateable_seller_order_or_403, get_reviewable_order_item_or_403

def product_reviews(request, product_id: int):

qs = (

Review.objects.select_related("buyer")

.filter(product_id=product_id)

.order_by("-created_at")

)

from django.db.models import Avg, Count

summary = qs.aggregate(avg=Avg("rating"), count=Count("id"))

avg_rating = summary.get("avg") or 0

review_count = summary.get("count") or 0

return render(

request,

"reviews/product_reviews.html",

{"reviews": qs, "avg_rating": avg_rating, "review_count": review_count, "product_id": product_id},

)

@require_http_methods(["GET", "POST"])

def review_create_for_order_item(request, order_item_id: int):

try:

item = get_reviewable_order_item_or_403(user=request.user, order_item_id=order_item_id)

except PermissionDenied:

raise Http404("Not found")

if hasattr(item, "review"):

messages.info(request, "You already reviewed this item.")

return redirect(item.product.get_absolute_url())

if request.method == "POST":

form = ReviewForm(request.POST)

if form.is_valid():

review: Review = form.save(commit=False)

review.product = item.product

review.order_item = item

review.buyer = request.user

review.save()

messages.success(request, "Thanks — your review was posted.")

return redirect(item.product.get_absolute_url())

else:

form = ReviewForm()

return render(

request,

"reviews/review_form.html",

{"form": form, "item": item, "product": item.product},

)

@require_http_methods(["GET", "POST"])

def seller_review_create(request, order_id: int, seller_id: int):

"""Create a seller rating for a seller within a specific PAID order."""

try:

order = get_rateable_seller_order_or_403(user=request.user, order_id=order_id, seller_id=seller_id)

except PermissionDenied:

raise Http404("Not found")

Prevent duplicates per order

existing = SellerReview.objects.filter(order_id=order.id, seller_id=seller_id, buyer_id=request.user.id).first()

if existing:

messages.info(request, "You already rated this seller for this order.")

return redirect("orders:detail", order_id=order.id)

if request.method == "POST":

form = SellerReviewForm(request.POST)

if form.is_valid():

sr: SellerReview = form.save(commit=False)

sr.order = order

sr.seller_id = seller_id

sr.buyer = request.user

sr.save()

messages.success(request, "Thanks — your seller rating was posted.")

return redirect("orders:detail", order_id=order.id)

else:

form = SellerReviewForm()

return render(

request,

"reviews/seller_review_form.html",

{"form": form, "order": order, "seller_id": seller_id},

)