```
from datetime import datetime, timedelta
from decimal import ROUND_HALF_UP, Decimal
from typing import TYPE_CHECKING, NotRequired, TypedDict

import pghistory
from celery.exceptions import ImproperlyConfigured
from django.apps import apps
from django.contrib.auth import get_user_model
from django.contrib.contenttypes.fields import GenericForeignKey
from django.contrib.contenttypes.models import ContentType
from django.contrib.postgres.aggregates import ArrayAgg
from django.contrib.postgres.fields import ArrayField
from django.core.signing import dumps
from django.db.models import (
    CASCADE,
    SET_NULL,
    BooleanField,
    CharField,
    Count,
    F,
    FloatField,
    ForeignKey,
    Index,
    JSONField,
    ManyToManyField,
    Model,
    OneToOneField,
    PositiveSmallIntegerField,
    Q,
    QuerySet,
    TextChoices,
    TextField,
    UniqueConstraint,
    URLField,
)
from django.db.models.expressions import OuterRef, Subquery
from django.db.models.query import Prefetch
from django.db.utils import IntegrityError
from django.utils.translation import gettext_lazy as _

from apps.account.models import OtpLog
from apps.assignment.models import Assignment
from apps.assignment.models import Grade as AssignmentGrade
from apps.common.error import ErrorCode
from apps.common.models import BooleanNowField, LearningObjectMixin,
OrderableMixin, TimeStampedMixin
from apps.common.util import AccessDate, OtpTokenDict
from apps.competency.models import Certificate, CertificateAward,
CertificateAwardDataDict
from apps.content.models import Media
from apps.course.trigger import course_create_grading_policy,
lessonmedia_unifier
from apps.discussion.models import Discussion
from apps.discussion.models import Grade as DiscussionGrade
from apps.exam.models import Exam
from apps.exam.models import Grade as ExamGrade
from apps.operation.models import FAQ, Announcement,
AnnouncementRead, Category, FAQItem, HonorCode, Instructor
from apps.survey.models import Survey

if TYPE_CHECKING:
    from django.contrib.auth.models import AbstractUser as User
else:
    User = get_user_model()


ASSESSIBLE_MODELS = [Exam, Assignment, Discussion]
ASSESSIBLE_MODEL_MAP = {(M._meta.app_label,
M._meta.model.__name__.lower()): M for M in ASSESSIBLE_MODELS}
ASSESSIBLE_GRADE_MODELS = {Exam: ExamGrade, Assignment:
AssignmentGrade, Discussion: DiscussionGrade}

TEMPLATE_SCHEDULES = {
    "start_today_email": {"offset_days": 0, "time": "09:00"},
    "weekly_progress_email": {"cron": "0 10 * * 1"},
    "exam_today_email": {"offset_days": 14, "time": "09:00"},
    "end_today_email": {"offset_days": 30, "time": "18:00"},
    "grade_completed_email": {"offset_days": 1, "time": "10:00"},
    "certificate_issued_email": {"offset_days": 0, "time": "15:00"},
}


class SessionDict(TypedDict):
    access_date: AccessDate
    course: Course
    engagement: NotRequired["Engagement"]
    otp_token: NotRequired[str]
    # stats: NotRequired["ScoreStatsDict"]


@pghistory.track()
class MessagePreset(Model):
    title = CharField(_("Title"), max_length=255, unique=True)
    description = TextField(_("Description"), blank=True,
default="")
    templates = ArrayField(CharField(max_length=50),
verbose_name=_("Templates"), blank=True, default=list)

    class Meta:
        verbose_name = _("Message Preset")
        verbose_name_plural = _("Message Presets")

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs):
        if self.templates:
            self.templates = list(dict.fromkeys(self.templates))
        super().save(*args, **kwargs)


@pghistory.track()
class Course(LearningObjectMixin):
    class LevelChoices(TextChoices):
        BEGINNER = "beginner", _("Beginner")
        INTERMEDIATE = "intermediate", _("Intermediate")
        ADVANCED = "advanced", _("Advanced")
        COMMON = "common", _("Common")

    owner = ForeignKey(User, CASCADE, verbose_name=_("Owner"))
    objective = TextField(_("Objective"), blank=True, default="")
    preview_url = URLField(_("Preview URL"), blank=True, null=True)
    effort_hours = PositiveSmallIntegerField(_("Effort Hours"))
    level = CharField(_("Level"), max_length=20,
choices=LevelChoices.choices)
    faq = ForeignKey(FAQ, SET_NULL, null=True, blank=True,
verbose_name=_("FAQ"))
    honor_code = ForeignKey(HonorCode, CASCADE,
verbose_name=_("Honor Code"))
    message_preset = ForeignKey(MessagePreset, SET_NULL, null=True,
blank=True, verbose_name=_("Message Preset"))

    instructors = ManyToManyField(Instructor,
through="CourseInstructor", blank=True,
verbose_name=_("Instructors"))
    announcements = ManyToManyField(Announcement,
through="CourseAnnouncement", blank=True,
verbose_name=_("Announcements"))  # fmt: skip
    surveys = ManyToManyField(Survey, through="CourseSurvey",
blank=True, verbose_name=_("Surveys"))

    categories = ManyToManyField(Category, blank=True,
verbose_name=_("Categories"))
    related_courses = ManyToManyField("self", blank=True,
symmetrical=False, verbose_name=_("Related Courses"))
    certificates = ManyToManyField(Certificate, blank=True,
verbose_name=_("Certificates"))

    class Meta(LearningObjectMixin.Meta):
        verbose_name = _("Course")
        verbose_name_plural = _("Courses")
        constraints = [UniqueConstraint(fields=["owner", "title"],
name="course_course_ow_ti_uniq")]

    if TYPE_CHECKING:
        lesson_set: "QuerySet[Lesson]"
        assessment_set: "QuerySet[Assessment]"
        gradingpolicy: "GradingPolicy"
        grading_criteria: list[GradingCriterionDict]

    def __str__(self):
        return f"{self.title} ({self.pk})"
    @classmethod
    async def get_session(cls, *, course_id: str, learner_id: str,
access_date: AccessDate):
        course = (
            await cls.objects
            .select_related("owner", "gradingpolicy", "honor_code")
            .prefetch_related(
                Prefetch(
                    "lesson_set",

queryset=Lesson.objects.order_by("ordering").prefetch_related(
                        Prefetch(
                            "medias",

queryset=Media.objects.annotate(ordering=F("lessonmedia__ordering"))
.order_by(
                                "lessonmedia__ordering"
                            ),
                        )
                    ),
                )
            )
            .aget(id=course_id)
        )
        course.grading_criteria = await
course.gradingpolicy.grading_criteria(access_date)
        session = SessionDict(access_date=access_date,
course=course)

        for lesson in course.lesson_set.all():
            lesson.start_date = access_date["start"] +
timedelta(days=lesson.start_offset)
            lesson.end_date = (
                lesson.start_date +
timedelta(days=lesson.end_offset)
                if lesson.end_offset is not None
                else access_date["end"]
            )

        engagement = (
            await Engagement.objects
            .select_related("gradebook")
            .filter(course=course, learner_id=learner_id,
active=True)
            .afirst()
        )
        if not engagement:
            if course.verification_required:
                session["otp_token"] = dumps(
                    OtpTokenDict(consumer_id=course.id,
app_label="course", model="course", user_id=learner_id)
                )
            return session
        session["engagement"] = engagement
        return session

    @classmethod
    async def get_detail(cls, id: str):
        return (
            await cls.objects
            .select_related("owner")
            .prefetch_related(
                Prefetch("faq__faqitem_set",
FAQItem.objects.filter(active=True).order_by("ordering")),
                Prefetch("categories",
Category.objects.order_by("id")),
                Prefetch(
                    "certificates",

Certificate.objects.select_related("issuer").filter(active=True).ord
er_by("-created"),
                ),
                Prefetch(
                    "instructors",
                    Instructor.objects
                    .annotate(lead=F("courseinstructor__lead"))
                    .filter(active=True)
                    .order_by("courseinstructor__ordering"),
                ),
                Prefetch("related_courses",
Course.objects.order_by("-modified")),
            )
            .aget(id=id)
        )

    def get_announcements(self, learner_id: str):
        return (
            self.announcements
            .annotate(
                read=Subquery(

AnnouncementRead.objects.filter(announcement=OuterRef("pk"),
user_id=learner_id).values("read")[:1]
                )
            )
            .filter(public=True)
            .order_by("courseannouncement__ordering", "-pinned")
        )

    @classmethod
    async def content_effective_date(
        cls, *, course_id: str, content_id: str, app_label: str,
model: str, access_date: AccessDate
    ):
        if ASSESSIBLE_MODEL_MAP.get((app_label, model)):
            accessible = await Assessment.objects.aget(
                course_id=course_id, item_id=content_id,
item_type__app_label=app_label, item_type__model=model
            )
        elif app_label == Media._meta.app_label and model ==
Media._meta.model.__name__.lower():
            # unique by lessonmedia trigger
            accessible = await
Lesson.objects.aget(course_id=course_id,
lessonmedia__media_id=content_id)
        else:
            raise ValueError(ErrorCode.UNKNOWN_COURSE_CONTENT)

        start = access_date["start"] +
timedelta(days=accessible.start_offset)
        end = start + timedelta(days=accessible.end_offset) if
accessible.end_offset is not None else access_date["end"]
        return AccessDate(start=start, end=end,
archive=access_date["archive"])

    @classmethod
    async def issue_context_key(cls, *, course_id: str, user_id:
str):
        en = await Engagement.objects.only("pk",
"course_id").aget(course_id=course_id, learner_id=user_id,
active=True)
        return en.issue_context_key()


@pghistory.track()
class CourseInstructor(OrderableMixin):
    course = ForeignKey(Course, CASCADE, verbose_name=_("Course"))
    instructor = ForeignKey(Instructor, CASCADE,
verbose_name=_("Instructor"))
    lead = BooleanField(_("Lead"), default=False)

    ordering_group = ("course",)

    class Meta(OrderableMixin.Meta):
        verbose_name = _("Course Instructor")
        verbose_name_plural = _("Course Instructors")
        constraints = [UniqueConstraint(fields=["course",
"instructor"], name="course_courseinstructor_co_in_uniq")]


@pghistory.track()
class CourseAnnouncement(OrderableMixin):
    course = ForeignKey(Course, CASCADE, verbose_name=_("Course"))
    announcement = ForeignKey(Announcement, CASCADE,
verbose_name=_("Announcement"))

    ordering_group = ("course",)

    class Meta(OrderableMixin.Meta):
        verbose_name = _("Course Announcement")
        verbose_name_plural = _("Course Announcements")
        constraints = [UniqueConstraint(fields=["course",
"announcement"], name="course_courseannouncement_co_an_uniq")]


@pghistory.track()
class CourseSurvey(OrderableMixin):
    class TimingChoices(TextChoices):
        PRE = "pre", _("Pre")
        POST = "post", _("Post")
        FREE = "free", _("Free")

    course = ForeignKey(Course, CASCADE, verbose_name=_("Course"))
    survey = ForeignKey(Survey, CASCADE, verbose_name=_("Survey"))
    timing = CharField(_("Timing"), max_length=10,
choices=TimingChoices.choices, default=TimingChoices.FREE)

    class Meta(OrderableMixin.Meta):
        verbose_name = _("Course Survey")
        verbose_name_plural = _("Course Surveys")
        constraints = [UniqueConstraint(fields=["course", "survey"],
name="course_coursesurvey_co_su_uniq")]


@pghistory.track()
class Lesson(OrderableMixin):
    course = ForeignKey(Course, CASCADE, verbose_name=_("Course"))
    title = CharField(_("Title"), max_length=255)
    description = TextField(_("Description"), blank=True,
default="")
    medias = ManyToManyField(Media, through="LessonMedia",
verbose_name=_("Medias"))
    start_offset = PositiveSmallIntegerField(_("Start Offset
(Days)"))
    end_offset = PositiveSmallIntegerField(_("End Offset (Days) from
Start Offset"), null=True, blank=True)

    ordering_group = ("course",)

    class Meta(OrderableMixin.Meta):
        verbose_name = _("Lesson")
        verbose_name_plural = _("Lessons")
        constraints = [UniqueConstraint(fields=["course", "title"],
name="course_lesson_co_ti_uniq")]

    if TYPE_CHECKING:
        start_date: datetime
        end_date: datetime

    def __str__(self):
        return self.title


@pghistory.track()
```

```
class LessonMedia(OrderableMixin):
    lesson = ForeignKey(Lesson, CASCADE, verbose_name=_("Lesson"))
    media = ForeignKey(Media, CASCADE, verbose_name=_("Media"))

    ordering_group = ("lesson",)

    class Meta(OrderableMixin.Meta):
        verbose_name = _("Lesson Media")
        verbose_name_plural = _("Lesson Medias")
        constraints = [UniqueConstraint(fields=["lesson", "media"],
name="course_lessonmedia_le_me_uniq")]

    if TYPE_CHECKING:
        media_id = str()


setattr(LessonMedia._meta, "triggers",
[lessonmedia_unifier(LessonMedia._meta.db_table,
Lesson._meta.db_table)])


@pghistory.track()
class Assessment(Model):
    course = ForeignKey(Course, CASCADE, verbose_name=_("Course"))
    weight = PositiveSmallIntegerField(_("Weight"))
    start_offset = PositiveSmallIntegerField(_("Start Offset
(Days)"))
    end_offset = PositiveSmallIntegerField(_("End Offset (Days) from
Start Offset"), null=True, blank=True)
    item_type = ForeignKey(
        ContentType,
        CASCADE,
        verbose_name=_("Item Type"),
        limit_choices_to={"model__in": [m.__name__.lower() for m in
ASSESSIBLE_MODELS]},
    )
    item_id = CharField(_("Item ID"), max_length=36)
    item = GenericForeignKey("item_type", "item_id")

    class Meta:
        verbose_name = _("Assessment")
        verbose_name_plural = _("Assessments")
        indexes = [Index(fields=["item_type", "item_id"])]
        constraints = [
            UniqueConstraint(fields=["course", "item_type",
"item_id"], name="course_assessment_co_itty_itid_uniq")
        ]


class GradingCriterionDict(TypedDict):
    title: str
    app_label: str
    model: str
    weight: int
    passing_point: int
    normalized_weight: float
    item_id: str
    start_date: datetime | None
    end_date: datetime | None


@pghistory.track()
class GradingPolicy(Model):
    course = OneToOneField(Course, CASCADE,
verbose_name=_("Course"))
    assessment_weight =
PositiveSmallIntegerField(verbose_name=_("Assessment Weight"),
default=100)
    completion_weight =
PositiveSmallIntegerField(verbose_name=_("Completion Weight"),
default=0)
    completion_passing_point =
PositiveSmallIntegerField(verbose_name=_("Completion Passing
Point"), default=80)

    class Meta:
        verbose_name = _("Grading Policy")
        verbose_name_plural = _("Grading Policies")

    if TYPE_CHECKING:
        course_id: str

    async def grading_criteria(self, access_date: AccessDate | None
= None) -> list[GradingCriterionDict]:
        start_date = access_date["start"] if access_date else None
        end_date = access_date["end"] if access_date else None
        criteria: list[GradingCriterionDict] = []

        total_weight = self.completion_weight +
self.assessment_weight

        if self.completion_weight or self.completion_passing_point:
            criteria.append(
                GradingCriterionDict(
                    title="Completion",
                    app_label="",
                    model="completion",
                    weight=self.completion_weight,
                    passing_point=self.completion_passing_point,
                    normalized_weight=float(self.completion_weight *
100 / total_weight) if total_weight else 0.0,
                    item_id=self.course_id,
                    start_date=start_date,
                    end_date=end_date,
                )
            )

        assessments = [
            assessment
            async for assessment in
self.course.assessment_set.select_related("item_type").order_by(
                "start_offset", "end_offset"
            )
        ]

        if not assessments:
            return criteria

        type_to_ids: dict[str, list[str]] = {}
        lookup: dict[str, Assessment] = {}

        for assessment in assessments:
            model = assessment.item_type.model
            type_to_ids.setdefault(model,
[]).append(assessment.item_id)
            lookup[assessment.item_id] = assessment

        items_qs = QuerySet().none()
        for model_name, ids in type_to_ids.items():
            app_label = lookup[ids[0]].item_type.app_label
            model_class = apps.get_model(app_label, model_name)
            qs = model_class.objects.filter(id__in=ids).values("id",
"title", "passing_point")
            items_qs = items_qs.union(qs)

        items = [item async for item in items_qs.all()]

        if not items:
            return criteria

        items_dict = {item["id"]: item for item in items}

        for assessment in assessments:
            item = items_dict.get(assessment.item_id)
            if not item:
                continue

            if not assessment.weight and not item["passing_point"]:
                continue

            start_offset = assessment.start_offset
            end_offset = assessment.end_offset

            criteria.append(
                GradingCriterionDict(
                    title=item["title"],
                    app_label=assessment.item_type.app_label,
                    model=assessment.item_type.model,
                    weight=assessment.weight,
                    passing_point=item["passing_point"],
                    normalized_weight=0.0,
                    item_id=assessment.item_id,
                    start_date=start_date +
timedelta(days=start_offset) if start_date else None,
                    end_date=None
                    if not start_date
                    else (start_date + timedelta(days=start_offset +
end_offset) if end_offset else end_date),
                )
            )

        if not criteria or all(p["weight"] == 0 for p in criteria):
            return criteria

        if len(criteria) == 1:
            criteria[0]["normalized_weight"] = 100.0
            return criteria

        assessment_criteria = [p for p in criteria if p["model"] !=
"completion"]
        if not assessment_criteria:
            return criteria

        total_assessment_weight = sum(p["weight"] for p in
assessment_criteria)
        if total_assessment_weight == 0:
            return criteria

        assessment_ratio = Decimal(str(self.assessment_weight)) /
total_weight * 100
        normalized_weights: list[tuple[int, Decimal]] = []

        for i, policy in enumerate(assessment_criteria):
            normalized_weight = Decimal(str(policy["weight"])) /
total_assessment_weight * assessment_ratio
            normalized_weight =
normalized_weight.quantize(Decimal("0.1"), rounding=ROUND_HALF_UP)
            normalized_weights.append((i, normalized_weight))
            policy["normalized_weight"] = float(normalized_weight)

        total_normalized = sum(weight for _, weight in
normalized_weights)
        difference = assessment_ratio - total_normalized

        if difference != 0:
            max_idx, _ = max(normalized_weights, key=lambda x: x[1])
            assessment_criteria[max_idx]["normalized_weight"] +=
float(difference)

        return criteria


setattr(Course._meta, "triggers",
[course_create_grading_policy(Course._meta.db_table,
GradingPolicy._meta.db_table)])
```

```
@pghistory.track()
class Engagement(TimeStampedMixin):
    course = ForeignKey(Course, CASCADE, verbose_name=_("Course"))
    learner = ForeignKey(User, CASCADE, verbose_name=_("Learner"))
    last_lesson = ForeignKey(Lesson, SET_NULL, verbose_name=_("Last
Lesson"), null=True, blank=True)
    active = BooleanField(_("Active"), default=True)

    class Meta(TimeStampedMixin.Meta):
        verbose_name = _("Engagement")
        verbose_name_plural = _("Engagements")
        indexes = [Index(fields=["learner_id", "active"])]
        constraints = [
            UniqueConstraint(
                fields=["course", "learner"],
condition=Q(active=True), name="course_engagement_co_le_uniq"
            )
        ]

    if TYPE_CHECKING:
        certificate_ids: list[int]  # annotated
        course_id: str

    def issue_context_key(self):
        return f"course::{self.course_id}::{self.pk}"

    @classmethod
    async def start(cls, *, course_id: str, learner_id: str):
        course = await Course.objects.aget(id=course_id)

        if course.verification_required:
            if not await
OtpLog.check_otp_verification(user_id=learner_id, consumer=course):
                raise
ValueError(ErrorCode.OTP_VERIFICATION_REQUIRED)

        try:
            engagement = await
Engagement.objects.acreate(course_id=course_id,
learner_id=learner_id, active=True)
        except IntegrityError:
            raise ValueError(ErrorCode.ALREADY_EXISTS)

        engagement._state.fields_cache["gradebook"] = None

        return engagement

    @classmethod
    async def request_certificate(cls, *, course_id: str, user_id:
str, certificate_id: int, verification_url: str):
        engagement = (
            await cls.objects
            .select_related("course", "learner", "gradebook")
            .annotate(
                certificate_ids=ArrayAgg(
                    "course__certificates__pk",
filter=Q(course__certificates__active=True), distinct=True
                )
            )
            .aget(course_id=course_id, learner_id=user_id,
active=True)
        )

        if certificate_id not in engagement.certificate_ids:
            raise ValueError(ErrorCode.CERTIFICATE_NOT_IN_COURSE)

        gradebook = getattr(engagement, "gradebook", None)
        if not gradebook or not (gradebook.confirmed and
gradebook.passed):
            raise
ValueError(ErrorCode.NOT_QUALIFIED_FOR_CERTIFICATE)

        data = CertificateAwardDataDict(
            document_title=_("Course Completion Certificate"),
            completion_title=engagement.course.title,
            completion_period=f"{engagement.created.strftime('%Y-%m-
%d')} ~ {gradebook.confirmed.strftime('%Y-%m-%d')}",
            completion_hours=_("%(hours)s hours") % {"hours":
engagement.course.effort_hours},
            recipient_name=engagement.learner.name,

recipient_birth_date=engagement.learner.birth_date.isoformat() if
engagement.learner.birth_date else "",
        )

        return await CertificateAward.issue(
            certificate_id=certificate_id,
            recipient=engagement.learner,
            context_key=engagement.issue_context_key(),
            data=data,
            verification_url=verification_url,
        )

    @classmethod
    async def grade(cls, *, course_id: str, learner_id: str, grader:
"User | None" = None):
        engagement = await
Engagement.objects.select_related("course__gradingpolicy").aget(
            course_id=course_id, learner_id=learner_id, active=True
        )
        criteria = await
engagement.course.gradingpolicy.grading_criteria()

        # completed lessons count rate
        completion_rate = 0.0

        qss = []
        for criterion in criteria:
            if criterion["model"] == "completion":
                completion = await (
                    Lesson.objects
                    .filter(course_id=criterion["item_id"])
                    .annotate(
                        media_count=Count("lessonmedia"),
                        passed_count=Count(
                            "lessonmedia",
                            filter=Q(

lessonmedia__media__watch__user_id=learner_id,

lessonmedia__media__watch__context_key=engagement.issue_context_key(
),

lessonmedia__media__watch__passed=True,
                            ),
                        ),
                    )
                    .aaggregate(
                        total_lessons=Count("id"),
                        passed_lessons=Count("id",
filter=Q(media_count__gt=0, passed_count=F("media_count"))),
                    )
                )
                total = completion["total_lessons"]
                passed = completion["passed_lessons"]
                completion_rate = (passed * 100.0 / total) if total
else 0.0

            else:
                M =
ASSESSIBLE_MODEL_MAP.get((criterion["app_label"],
criterion["model"]))
                G = ASSESSIBLE_GRADE_MODELS.get(M)
                if not (M and G):
                    raise ImproperlyConfigured(
                        f"Cannot find assessable model
{criterion['app_label']}.{criterion['model']}"
                    )

                pk_path =
f"attempt__{M._meta.model.__name__.lower()}_id"
                qss.append(
                    G.objects.filter(
                        **{pk_path: criterion["item_id"]},
                        attempt__learner_id=learner_id,

attempt__context_key=engagement.issue_context_key(),
                        attempt__active=True,
                        completed__isnull=False,
                        confirmed__isnull=False,
                    ).values_list(pk_path, "score", "passed")
                )

        if qss:
            qs = qss[0].union(*qss[1:]) if len(qss) > 1 else qss[0]
            assessment_results = {r[0]: {"score": r[1], "passed":
r[2]} async for r in qs}
        else:
            assessment_results = {}

        total_score = 0.0
        total_weight = 0.0
        failed_exist = False
        details = {}

        for criterion in criteria:
            weight = criterion["normalized_weight"]

            if criterion["model"] == "completion":
                passed = completion_rate >=
criterion["passing_point"]

                details["completion"] = {
                    "rate": completion_rate,
                    "passing_point": criterion["passing_point"],
                    "passed": passed,
                }

                if weight > 0:
                    total_score += completion_rate * weight / 100
                    total_weight += weight

                if not passed:
                    failed_exist = True

            else:
                result =
assessment_results.get(criterion["item_id"])
                if not result:
                    details[criterion["item_id"]] = None
                    failed_exist = True
                    continue

                score = result["score"]
                passed = result["passed"]

                details[criterion["item_id"]] = {
                    "score": score,
                    "passing_point": criterion["passing_point"],
                    "passed": passed,
                }

                if weight > 0:
                    total_score += score * weight / 100
                    total_weight += weight

                if not passed:
                    failed_exist = True

        final_score = total_score if total_weight > 0 else 0.0

        await Gradebook.objects.aupdate_or_create(
            engagement=engagement,
            defaults={
                "details": details,
                "score": final_score,
                "completion_rate": completion_rate,
                "passed": not failed_exist,
                "grader": grader,
            },
        )


@pghistory.track()
class Gradebook(TimeStampedMixin):
    engagement = OneToOneField(Engagement, CASCADE,
verbose_name=_("Engagement"))
    details = JSONField(verbose_name=_("Details"))
    score = FloatField(verbose_name=_("Score"))
    completion_rate = FloatField(_("Completion Rate"))
    passed = BooleanField(verbose_name=_("Passed"))
    confirmed = BooleanNowField(_("Confirmed"), null=True,
blank=True)
    note = TextField(_("Note"), blank=True, default="")
    grader = ForeignKey(User, CASCADE, null=True, blank=True,
verbose_name=_("Grader"), related_name="+")

    class Meta(TimeStampedMixin.Meta):
        verbose_name = _("Gradebook")
        verbose_name_plural = _("Gradebooks")
from typing import TYPE_CHECKING
from uuid import uuid4

import mimesis
from django.conf import settings
from django.db.models import QuerySet
from factory.declarations import Iterator, LazyFunction, SubFactory
from factory.django import DjangoModelFactory
from factory.helpers import post_generation
from mimesis.plugins.factory import FactoryField

from apps.assignment.models import Assignment
from apps.common.factory import LearningObjectFactory
from apps.competency.models import Certificate
from apps.content.models import Media
from apps.content.tests.factories import MediaFactory
from apps.course.models import (
    TEMPLATE_SCHEDULES,
    Assessment,
    Course,
    CourseAnnouncement,
    CourseInstructor,
    CourseSurvey,
    Lesson,
    LessonMedia,
    MessagePreset,
)
from apps.discussion.models import Discussion
from apps.exam.models import Exam
from apps.operation.models import Category
from apps.operation.tests.factories import AnnouncementFactory,
FAQFactory, HonorCodeFactory, InstructorFactory
from apps.survey.models import Survey

generic = mimesis.Generic(settings.DEFAULT_LANGUAGE)


class MessagePresetFactory(DjangoModelFactory[MessagePreset]):
    title = FactoryField("text.title")
    description = FactoryField("text")
    templates = LazyFunction(lambda:
list(TEMPLATE_SCHEDULES.keys()))

    class Meta:
        model = MessagePreset
        django_get_or_create = ("title",)
        skip_postgeneration_save = True


class CourseFactory(LearningObjectFactory[Course]):
    passing_point = FactoryField("choice", items=[60, 80])
    max_attempts = FactoryField("choice", items=[1, 2])
    verification_required = True

    owner = SubFactory("account.tests.factories.UserFactory")
    objective = FactoryField("text")
    preview_url = None
    effort_hours = FactoryField("choice", items=[8, 16, 32])
    level = Iterator(Course.LevelChoices)
    honor_code = SubFactory(HonorCodeFactory)
    faq = SubFactory(FAQFactory)
    message_preset = SubFactory(MessagePresetFactory)

    class Meta:
        model = Course
        django_get_or_create = ("title", "owner")
        skip_postgeneration_save = True

    if TYPE_CHECKING:
        categories: QuerySet[Category]
        related_courses: QuerySet[Course]
        certificates: QuerySet[Certificate]
        lesson_set: QuerySet[Lesson]
        pk: int

    @post_generation
    def post_generation(self, create, extracted, **kwargs):
        if not create:
            return

        # manytomany

self.categories.set(Category.objects.filter(depth=3).order_by("?")[:
generic.random.randint(1, 2)])

self.related_courses.set(Course.objects.exclude(id=self.pk).order_by
("?")[: generic.random.randint(1, 2)])
        self.certificates.set(Certificate.objects.order_by("?")[:
generic.random.randint(1, 2)])

        # instructor
        instructors =
InstructorFactory.create_batch(generic.random.randint(1, 3))
        if instructors:
            CourseInstructor.objects.bulk_create(
                [
                    CourseInstructor(course=self,
instructor=instructor, lead=True if i == 0 else False)
                    for i, instructor in enumerate(instructors)
                ],
                ignore_conflicts=True,
            )

        # announcement
        announcements =
AnnouncementFactory.create_batch(generic.random.randint(2, 4))
        if announcements:
            CourseAnnouncement.objects.bulk_create(
                [
                    CourseAnnouncement(course=self,
announcement=announcement, ordering=i)
                    for i, announcement in enumerate(announcements)
                ],
                ignore_conflicts=True,
            )

        # survey
        surveys = Survey.objects.order_by("?")[:
generic.random.randint(1, 2)]
        if surveys:
            CourseSurvey.objects.bulk_create(
                [
                    CourseSurvey(
                        course=self,
                        survey=survey,
timing=generic.random.choice(CourseSurvey.TimingChoices.choices)[0],
                        ordering=i,
                    )
                    for i, survey in enumerate(surveys)
                ],
                ignore_conflicts=True,
            )

        # lesson
        medias = Media.objects.order_by("?")[:
generic.random.choice([8, 16, 32])]

        for i, media in enumerate(medias):
            lesson, created = Lesson.objects.get_or_create(
                course=self,
                title=media.title,
                defaults={"description": media.description,
"start_offset": i * 7, "end_offset": 7, "ordering": i},
            )

            if created:
                LessonMedia(lesson=lesson, media=media,
ordering=0).save()

                if i in [2, 4]:
                    media = MediaFactory.create(owner=self.owner,
url=f"{generic.internet.url()}/{uuid4().hex}.mp4")
                    LessonMedia(lesson=lesson, media=media,
ordering=i).save()

        # assessment
        discussions = Discussion.objects.order_by("?")[:
generic.random.randint(1, 2)]
        exams = Exam.objects.order_by("?")[:
generic.random.randint(1, 2)]
        assignments = Assignment.objects.order_by("?")[:
generic.random.randint(1, 2)]

        last_lesson = self.lesson_set.last()
        course_days = last_lesson.start_offset + 7 if last_lesson
else 30

        weeks = course_days // 7
        assessments_to_create = []

        discussion_weeks = [1, 5][: len(discussions)] if weeks >= 5
else [1][: len(discussions)]
        assignment_weeks = [2, 6][: len(assignments)] if weeks >= 6
else [2][: len(assignments)]

        if weeks >= 8:
            exam_weeks = [weeks // 2, weeks][: len(exams)]
        elif weeks >= 4:
            exam_weeks = [weeks][: len(exams)]
        else:
            exam_weeks = []

        for i, discussion in enumerate(discussions):
            if i < len(discussion_weeks):
                week = discussion_weeks[i]
                if week <= weeks:
                    start_offset = (week - 1) * 7
                    assessments_to_create.append(
                        Assessment(course=self, weight=20,
start_offset=start_offset, end_offset=7, item=discussion)
                    )

        for i, assignment in enumerate(assignments):
            if i < len(assignment_weeks):
                week = assignment_weeks[i]
                if week <= weeks:
                    start_offset = (week - 1) * 7
                    assessments_to_create.append(
                        Assessment(course=self, weight=30,
start_offset=start_offset, end_offset=7, item=assignment)
                    )

        for i, exam in enumerate(exams):
            if i < len(exam_weeks):
                week = exam_weeks[i]
                start_day = (week - 1) * 7
                assessments_to_create.append(
                    Assessment(course=self, weight=50,
start_offset=start_day, end_offset=7, item=exam)
                )

        Assessment.objects.bulk_create(assessments_to_create,
ignore_conflicts=True)
import pytest
from django.conf import settings
from mimesis.plugins.factory import FactoryField
from pytest_django import DjangoDbBlocker

from apps.course.tests.factories import CourseFactory


@pytest.mark.order(-2)
@pytest.mark.django_db
def test_course():
    CourseFactory.create()


@pytest.mark.order(-2)
@pytest.mark.load_data
def test_load_course_data(db_no_rollback: DjangoDbBlocker):
    with FactoryField.override_locale(settings.DEFAULT_LANGUAGE):
        CourseFactory.create_batch(10)
from django.apps import AppConfig
from django.utils.translation import gettext_lazy as _


class CourseConfig(AppConfig):
    default_auto_field = "django.db.models.BigAutoField"
    name = "apps.course"
    verbose_name = _("Course")
from asgiref.sync import async_to_sync
from django.contrib import admin
from django.http import HttpRequest
from django.utils.translation import gettext_lazy as _
from django_jsonform.forms.fields import JSONFormField
from unfold.decorators import action

from apps.common.admin import HiddenModelAdmin, ModelAdmin,
TabularInline
from apps.course.models import (
    TEMPLATE_SCHEDULES,
    Assessment,
    Course,
    CourseAnnouncement,
    CourseInstructor,
    CourseSurvey,
    Engagement,
    Gradebook,
    GradingPolicy,
    Lesson,
    LessonMedia,
    MessagePreset,
)


@admin.register(Course)
class CourseAdmin(ModelAdmin[Course]):
    class CategoryInline(TabularInline[Course.categories.through]):
        model = Course.categories.through
        verbose_name = _("Category")
        verbose_name_plural = _("Categories")

    class
CertificateInline(TabularInline[Course.certificates.through]):
        model = Course.certificates.through
        verbose_name = _("Certificate")
        verbose_name_plural = _("Certificates")

    class CourseInstructorInline(TabularInline[CourseInstructor]):
        model = CourseInstructor
        # orderable
        ordering = ("ordering", "id")
        ordering_field = "ordering"

    class
CourseAnnouncementInline(TabularInline[CourseAnnouncement]):
        model = CourseAnnouncement
        # orderable
        ordering = ("ordering", "id")
        ordering_field = "ordering"

    class CourseSurveyInline(TabularInline[CourseSurvey]):
        model = CourseSurvey

    class AssessmentInline(TabularInline[Assessment]):
        model = Assessment

    class LessonInline(TabularInline[Lesson]):
        model = Lesson
        # orderable
        ordering = ("ordering", "id")
        ordering_field = "ordering"

    class GradingPolicyInline(TabularInline[GradingPolicy]):
        model = GradingPolicy

    class RelatedCourseInline(TabularInline[Course]):
        model = Course.related_courses.through
        fk_name = "from_course"
        verbose_name = _("Related Course")
        verbose_name_plural = _("Related Courses")

    inlines = (
        CategoryInline,
        CertificateInline,
        CourseAnnouncementInline,
        CourseInstructorInline,
        CourseSurveyInline,
        LessonInline,
        AssessmentInline,
        GradingPolicyInline,
        RelatedCourseInline,
    )

    def get_fields(self, request, obj=None):
        return [
            f
            for f in super().get_fields(request, obj=obj)
            if f not in ("categories", "certificates",
"related_courses")
        ]


@admin.register(Lesson)
class LessonAdmin(HiddenModelAdmin[Lesson]):
    class LessonMediaInline(TabularInline[LessonMedia]):
        model = LessonMedia
        # orderable
        ordering = ("ordering", "id")
        ordering_field = "ordering"
    inlines = (LessonMediaInline,)


@admin.register(GradingPolicy)
class GradingPolicyAdmin(HiddenModelAdmin[GradingPolicy]):
    pass


@admin.register(CourseInstructor)
class CourseInstructorAdmin(HiddenModelAdmin[CourseInstructor]):
    pass


@admin.register(CourseAnnouncement)
class CourseAnnouncementAdmin(HiddenModelAdmin[CourseAnnouncement]):
    pass


@admin.register(CourseSurvey)
class CourseSurveyAdmin(HiddenModelAdmin[CourseSurvey]):
    pass


@admin.register(Assessment)
class AssessmentAdmin(HiddenModelAdmin[Assessment]):
    pass


@admin.register(LessonMedia)
class LessonMediaAdmin(HiddenModelAdmin[LessonMedia]):
    pass


@admin.register(Engagement)
class EngagementAdmin(ModelAdmin[Engagement]):
    class GradebookInline(TabularInline[Gradebook]):
        model = Gradebook

    inlines = (GradebookInline,)

    actions_submit_line = ["grade"]

    @action(description=_("Grade"), permissions=["grade"])
    def grade(self, request: HttpRequest, obj: Engagement):
        async_to_sync(Engagement.grade)(course_id=obj.course.id,
learner_id=obj.learner.pk, grader=request.user)

    def has_grade_permission(self, request: HttpRequest, object_id:
str | int):
        return request.user.is_superuser


@admin.register(Gradebook)
class GradebookAdmin(ModelAdmin[Gradebook]):
    pass


@admin.register(MessagePreset)
class MessagePresetAdmin(HiddenModelAdmin[MessagePreset]):
    def formfield_for_dbfield(self, db_field, request, **kwargs):
        if db_field.name == "templates":
            return JSONFormField(
                schema={"type": "array", "items": {"type": "string",
"choices": list(TEMPLATE_SCHEDULES.keys())}}
            )
        return super().formfield_for_dbfield(db_field, request,
**kwargs)
from django.urls.base import reverse_lazy
from ninja.pagination import paginate
from ninja.router import Router

from apps.common.util import HttpRequest, Pagination
from apps.course.api.schema import (
    CourseAnnounceReadSchema,
    CourseAnnounceSchema,
    CourseCertificateRequestSchema,
    CourseDetailSchema,
    CourseEngagementSchema,
    CourseSessionSchema,
)
from apps.course.models import Course, Engagement
from apps.learning.api.access_control import access_date
from apps.operation.models import AnnouncementRead

router = Router(by_alias=True)


@router.get("/{id}/detail", response=CourseDetailSchema)
async def get_detail(request: HttpRequest, id: str):
    return await Course.get_detail(id)


@router.get("/{id}/session", response=CourseSessionSchema)
@access_date("course", "course")
async def get_session(request: HttpRequest, id: str):
    return await Course.get_session(course_id=id,
learner_id=request.auth, access_date=request.access_date)


@router.post("/{id}/engage", response=CourseEngagementSchema)
@access_date("course", "course")
async def start_engagement(request: HttpRequest, id: str):
    return await Engagement.start(course_id=id,
learner_id=request.auth)


@router.get("/{id}/announcement",
response=list[CourseAnnounceSchema])
```

```
@access_date("course", "course")
@paginate(Pagination)
async def get_announcements(request: HttpRequest, id: str):
    return Course(id=id).get_announcements(learner_id=request.auth)


@router.post("/{id}/announcement/read")
@access_date("course", "course")
async def read_announcement(request: HttpRequest, id: str, data:
CourseAnnounceReadSchema):
    await AnnouncementRead.objects.acreate(user_id=request.auth,
announcement_id=data.announcement_id)


@router.post("/{id}/certificate/request")
async def request_certificate(request: HttpRequest, id: str, data:
CourseCertificateRequestSchema):
    # cf competency/views.py
    verification_url =
request.build_absolute_uri(reverse_lazy("verify_certificate"))
    return await Engagement.request_certificate(
        course_id=id, user_id=request.auth,
certificate_id=data.certificate_id,
verification_url=verification_url
    )
from datetime import datetime
from typing import Annotated, Literal

from pydantic.fields import Field

from apps.account.api.schema import OwnerSchema
from apps.common.schema import AccessDateSchema,
LearningObjectMixinSchema, Schema, TimeStampedMixinSchema
from apps.course.models import Course
from apps.operation.api.schema import FAQItemSchema, HonorCodeSchema

LevelType = Literal["beginner", "intermediate", "advanced",
"common"]


class CourseDetailSchema(LearningObjectMixinSchema):
    class CourseCategorySchema(Schema):
        id: int
        name: str
        ancestors: list[str]

    class CourseCertificateSchema(Schema):
        class CourseCertificateIssuerSchema(Schema):
            name: str
            logo: str | None

        id: int
        name: str
        thumbnail: str
        description: str
        issuer: CourseCertificateIssuerSchema

    class CourseInstructorSchema(Schema):
        id: int
        name: str
        about: str
        bio: list[str]
        avatar: str | None
        lead: bool

    class RelatedCourseSchema(Schema):
        id: str
        title: str
        description: str
        thumbnail: str | None

    id: str
    owner: OwnerSchema
    objective: str
    preview_url: str | None
    effort_hours: int
    level: LevelType

    faq_items: list[FAQItemSchema]
    categories: list[CourseCategorySchema]
    certificates: list[CourseCertificateSchema]
    instructors: list[CourseInstructorSchema]
    related_courses: list[RelatedCourseSchema]

    @staticmethod
    def resolve_faq_items(obj: Course):
        return obj.faq.faqitem_set.all() if obj.faq else []

    @staticmethod
    def resolve_lessons(obj: Course):
        return obj.lesson_set.all()


class CourseEngagementSchema(TimeStampedMixinSchema):
    class CourseGradebookSchema(TimeStampedMixinSchema):
        id: int
        details: dict[str, float]
        score: float
        completion_rate: float
        passed: bool

    id: int
    gradebook: Annotated[CourseGradebookSchema, Field(None)]
    active: bool


class CourseSchema(LearningObjectMixinSchema):
    class LessonSchema(Schema):
        class LessonMediaSchema(Schema):
            id: str
            title: str
            thumbnail: str | None
            format: str
            ordering: int

        id: int
        medias: list[LessonMediaSchema]
        start_date: datetime
        end_date: datetime
        ordering: int
        title: str
        description: str

    class GradingCriterionSchema(Schema):
        title: str
        app_label: str
        model: str
        passing_point: int
        weight: float
        normalized_weight: float
        item_id: str
        start_date: datetime | None
        end_date: datetime | None

    id: str
    honor_code: HonorCodeSchema
    grading_criteria: list[GradingCriterionSchema]
    lessons: list[LessonSchema]
    objective: str
    preview_url: str | None
    effort_hours: int
    level: LevelType

    @staticmethod
    def resolve_lessons(obj: Course):
        return obj.lesson_set.all()


class CourseSessionSchema(Schema):
    access_date: AccessDateSchema
    course: CourseSchema
    engagement: Annotated[CourseEngagementSchema, Field(None)]
    otp_token: Annotated[str, Field(None)]


class CourseAnnounceSchema(TimeStampedMixinSchema):
    id: int
    read: datetime | None
    title: str
    body: str
    public: bool
    pinned: bool
```

```
class CourseAnnounceReadSchema(Schema):
    announcement_id: int


class CourseCertificateRequestSchema(Schema):
    certificate_id: int
import re
from pathlib import Path

from django.conf import settings
from django.core.management.base import BaseCommand
from django.template import Context
from django.template import Template as DjangoTemplate
from django.utils.translation import gettext as _
from mjml import mjml2html


class Command(BaseCommand):
    help = _("Convert MJML templates to HTML")

    def handle(self, *args: object, **options: dict[str, object]):
        app_root = Path(__file__).resolve().parent.parent.parent
        mjml_dir = app_root / "mjml"
        mail_dir = app_root / "templates" / "course" / "mail"

        mail_dir.mkdir(parents=True, exist_ok=True)
        mjml_files = list(mjml_dir.glob("*.mjml"))

        if not mjml_files:
            self.stdout.write(self.style.WARNING(_("No MJML files
found")))
            return

        static_context = Context({
            "platform_name": settings.PLATFORM_NAME,
            "platform_address": settings.PLATFORM_ADDRESS,
            "privacy_policy_url": settings.PRIVACY_POLICY_URL,
            "terms_url": settings.TERMS_URL,
            "support_email": settings.DEFAULT_FROM_EMAIL,
        })

        for mjml_file in mjml_files:
            html_file = mail_dir / f"{mjml_file.stem}.html"

            with open(mjml_file, "r", encoding="utf-8") as file:
                mjml_str = file.read()

            root_start = mjml_str.find("<mjml>")
            root_end = mjml_str.find("</mjml>") + len("</mjml>")

            if root_start == -1 or root_end == -1:
                self.stdout.write(self.style.ERROR(_("Invalid MJML
file: %(name)s") % {"name": mjml_file.name}))
                continue

            before_root = mjml_str[:root_start]
            after_root = mjml_str[root_end:]

            def partial_loader(path: str):
                with open(mjml_dir / path, "r", encoding="utf-8") as
file:
                    partial = file.read()
                return partial

            body = mjml_str[root_start:root_end]
            html_content = before_root + mjml2html(body,
include_loader=partial_loader) + after_root

            html_content =
DjangoTemplate(html_content).render(static_context)
            html_content =
self.restore_dynamic_placeholder(html_content)

            with open(html_file, "w", encoding="utf-8") as file:
                file.write(html_content)

            self.stdout.write(self.style.SUCCESS(_("Successfully
converted %(name)s") % {"name": mjml_file.stem}))

    @staticmethod
    def restore_dynamic_placeholder(template_str: str):
        return re.sub(r"\{\s*(\w+)\s*\}", r"{{ \1 }}", template_str)
import pgtrigger


def lessonmedia_unifier(lessonmedia_table: str, lesson_table: str):
    return pgtrigger.Trigger(
        name=f"{lessonmedia_table}_unifier",
        operation=pgtrigger.Insert | pgtrigger.Update,
        when=pgtrigger.Before,
        func=f"""
            IF EXISTS (
                SELECT 1
                FROM {lessonmedia_table} lm
                JOIN {lesson_table} l ON l.id = lm.lesson_id
                WHERE lm.media_id = NEW.media_id
                  AND l.course_id = (
                      SELECT course_id
                      FROM {lesson_table}
                      WHERE id = NEW.lesson_id
                      LIMIT 1
                  )
                  AND lm.id IS DISTINCT FROM NEW.id
            ) THEN
                RAISE EXCEPTION 'Media already exists in this
course';
```

```
            END IF;
            RETURN NEW;
        """,
    )


def course_create_grading_policy(course_table: str,
gradingpolicy_table: str):
    return pgtrigger.Trigger(
        name=f"{course_table}_create_grading_policy",
        operation=pgtrigger.Insert,
        when=pgtrigger.After,
        func=f"""
            INSERT INTO {gradingpolicy_table} (
                course_id, assessment_weight, completion_weight,
completion_passing_point
            ) VALUES (
                NEW.id, 100, 0, 80
            );
            RETURN NEW;
        """,
    )
```