TIMETABLE SERVICE:

```
# backend/app/services/timetable_generation_service.py
"""
Timetable Generation Service for SchoolOS Academics Module (ASYNC).

This service implements AI-powered timetable scheduling with constraint
satisfaction.
Inspired by TimetableMaster reference software for real-world school
scheduling.

ARCHITECTURE PATTERN (matches cart_service.py):
- Service layer contains PURE BUSINESS LOGIC only
- No JWT/auth handling - all user context comes from endpoint layer
- All operations are school-scoped (school_id passed from current_profile)
- Uses AsyncSession for all database operations

Security:
- Endpoint layer validates school_id matches current_profile.school_id
- Service assumes all inputs are pre-validated by endpoint
- RLS policies provide defense-in-depth at database level

Performance:
- Single-pass algorithm with conflict tracking
- Optimized constraint validation (early exit on hard constraints)
- Minimal database queries (bulk operations where possible)
"""
import random
from collections import defaultdict
from datetime import time
from typing import Optional
from uuid import UUID

from sqlalchemy import select
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import selectinload

from app.models.period import Period
from app.models.subject import Subject
from app.models.timetable import Timetable
from app.schemas.timetable_schema import (
    ConflictDetail,
    ConstraintRule,
    SubjectRequirement,
    TimetableConstraint,
    TimetableEntryOut,
    TimetableGenerateRequest,
    TimetableGenerateResponse,
    UnassignedSubjectInfo,
)

#
==================================================================
=================
# PART 1: DATA STRUCTURES
#
==================================================================
=================


class Slot:
    """
    Represents a single schedulable time slot in the timetable grid.

    Attributes:
        day: Integer representing day of week (1=Monday, 6=Saturday)
        period_id: Database ID of the period
        period_number: Sequential number for ordering (1, 2, 3...)
        start_time: Time when period starts
        is_occupied: Whether this slot is already assigned
        teacher_id: ID of teacher assigned to this slot (if any)
        subject_id: ID of subject scheduled in this slot (if any)
    """

    def __init__(self, day: int, period_id: int, period_number: int, start_time:
time):
        self.day = day
        self.period_id = period_id
        self.period_number = period_number
        self.start_time = start_time
        self.is_occupied = False
        self.teacher_id: Optional[int] = None
        self.subject_id: Optional[int] = None

    def __repr__(self):
        return f"Slot(Day{self.day}, P{self.period_number}, {'Occupied' if
self.is_occupied else 'Free'})"


class ScheduleState:
    """
    Tracks the current state of the timetable being generated.
    This is the central data structure that maintains:
    - The schedule grid (days x periods)
    - Conflict tracking for teachers and classes
    - Subject placement history for constraint validation
    - Teacher workload tracking (daily and weekly)
    - Generated timetable entries
    - Warnings and conflicts encountered
    """

    def __init__(self, class_id: int, school_id: int, working_days: list[int]):
        self.class_id = class_id
        self.school_id = school_id
        self.working_days = working_days

        # Grid structure: day -> period_id -> Slot
        self.grid: dict[int, dict[int, Slot]] = defaultdict(dict)

        # Conflict tracking: teacher_id -> day -> set(period_ids)
        self.teacher_schedule: dict[int, dict[int, set[int]]] = defaultdict(lambda:
defaultdict(set))

        # Class schedule tracking: day -> set(period_ids)
        self.class_schedule: dict[int, set[int]] = defaultdict(set)

        # Subject placement history: subject_id -> [(day, period_id)]
        self.subject_placements: dict[int, list[tuple[int, int]]] = defaultdict(list)

        # Teacher workload tracking (NEW)
        # teacher_id -> day -> count of classes
        self.teacher_daily_load: dict[int, dict[int, int]] = defaultdict(lambda:
defaultdict(int))
        # teacher_id -> total classes in week
        self.teacher_weekly_load: dict[int, int] = defaultdict(int)

        # Output collections
        self.entries: list[dict] = []  # Raw dict entries to be converted to
TimetableEntryOut
        self.warnings: list[str] = []
        self.conflicts: list[ConflictDetail] = []


#
==============================================================
================
# PART 2: CONSTRAINT VALIDATORS
#
```

```
============================================================================

class ConstraintValidator:
    """
    Validates hard and soft constraints for timetable scheduling.

    Hard Constraints (MUST satisfy):
    - Teacher availability (can't be in two places at once)
    - Class conflicts (class can't have overlapping periods)

    Soft Constraints (SHOULD satisfy, but can be violated with warning):
    - Subject timing restrictions (e.g., PE only in afternoon)
    - Minimum gap between subject occurrences
    - Core subjects in morning slots
    """

    @staticmethod
    def validate_teacher_availability(teacher_id: int, day: int, period_id: int, state:
ScheduleState, constraints: list[ConstraintRule]) -> tuple[bool, Optional[str]]:
        """
        Check if teacher is available at this slot.

        Returns:
            (is_valid, error_message)
            - True with None: Teacher is available
            - False with message: Conflict detected
        """
        # Hard constraint: Teacher double-booking
        if period_id in state.teacher_schedule[teacher_id][day]:
            return False, f"Teacher {teacher_id} already teaching Day {day}, Period
{period_id}"

        # Soft constraint: Custom availability rules (e.g., part-time teachers)
        for rule in constraints:
            if rule.rule_type == "teacher_availability" and rule.target_id ==
teacher_id:
                blocked_days = rule.parameters.get("blocked_days", [])
                if day in blocked_days:
                    return False, f"Teacher {teacher_id} not available on Day {day}"

        return True, None

    @staticmethod
    def validate_subject_timing(subject_id: int, period_number: int,
period_start_time: time, constraints: list[ConstraintRule]) -> tuple[bool,
Optional[str]]:
        """
        Check if subject can be scheduled at this time.

        Example: Physical Education only in last 2 periods

        Returns:
            (is_valid, error_message)
        """
        for rule in constraints:
            if rule.rule_type == "subject_time_restriction" and rule.target_id ==
subject_id:
                # Check allowed periods (e.g., [7, 8] for PE)
                allowed_periods = rule.parameters.get("allowed_periods", [])
                if allowed_periods and period_number not in allowed_periods:
                    return False, f"Subject {subject_id} restricted to periods
{allowed_periods}"

                # Check time-based restriction (e.g., "only after 14:00")
                min_time = rule×parameters×get("min_start_time")
                if min_time:
                    min_time_obj = time.fromisoformat(min_time)
                    if period_start_time < min_time_obj:
                        return False, f"Subject {subject_id} must start after {min_time}"

        return True, None

    @staticmethod
    def validate_min_gap_days(subject_id: int, day: int, state: ScheduleState,
min_gap: int) -> tuple[bool, Optional[str]]:
        """
        Ensure minimum gap between subject occurrences.

        Example: Math shouldn't be on consecutive days (min_gap=1)

        Returns:
            (is_valid, error_message)
        """
        if min_gap == 0:
            return True, None

        placements = state.subject_placements[subject_id]
        for placed_day, _ in placements:
            if abs(placed_day - day) < min_gap:
                return False, f"Subject {subject_id} needs {min_gap} day gap
(currently on day {placed_day})"
```

```
        return True, None

    @staticmethod
    def find_consecutive_slots(state: ScheduleState, day: int, required_count:
int, available_periods: list[Slot]) -> Optional[list[Slot]]:
        """
        Find N consecutive free periods on a given day.

        Used for lab subjects that require 2-3 consecutive periods.

        Args:
            state: Current schedule state
            day: Day of week to search
            required_count: Number of consecutive periods needed (e.g., 2 for lab)
            available_periods: List of free slots to search

        Returns:
            List of consecutive slots if found, None otherwise
        """
        # Sort by period number for sequential search
        sorted_periods = sorted(available_periods, key=lambda s:
s.period_number)

        for i in range(len(sorted_periods) - required_count + 1):
            candidate = sorted_periods[i : i + required_count]

            # Verify periods are truly consecutive (no gaps)
            period_numbers = [s.period_number for s in candidate]
            expected_range = list(range(period_numbers[0], period_numbers[0] +
required_count))

            if period_numbers == expected_range:
                # Verify all slots are free
                if all(not s.is_occupied for s in candidate):
                    return candidate

        return None


#
==================================================================
=================
# PART 2B: TEACHER WORKLOAD CONSTRAINT VALIDATORS (NEW)
#
==================================================================
=================


class TeacherWorkloadValidator:
    """
    Validates teacher workload constraints (daily and weekly limits).

    PRIORITY: Highest - these are HARD constraints that MUST be enforced.
    Teachers cannot exceed max_classes_per_day or max_classes_per_week.

    Min constraints are SOFT - generate warnings but don't block scheduling.
    """

    @staticmethod
    def can_assign_teacher(teacher_id: int, day: int, state: ScheduleState,
teacher_constraints: Optional["TimetableConstraint"]) -> tuple[bool,
Optional[str]]:
        """
        Check if teacher can be assigned to a slot based on workload limits.

        Priority Order:
        1. Check daily maximum (HARD constraint)
        2. Check weekly maximum (HARD constraint)
        3. Return True if both pass

        Args:
            teacher_id: Teacher to check
            day: Day of week (1-7)
            state: Current schedule state
            teacher_constraints: Teacher workload limits

        Returns:
            (can_assign, rejection_reason)
        """
        if not teacher_constraints:
            return True, None

        # Check daily maximum (HARD constraint - MUST enforce)
        if teacher_constraints.max_classes_per_day:
            current_daily = state.teacher_daily_load[teacher_id][day]
            if current_daily >= teacher_constraints.max_classes_per_day:
                return False, (f"Teacher {teacher_id} has reached daily limit "
f"({current_daily}/{teacher_constraints.max_classes_per_day}) on day {day}")

        # Check weekly maximum (HARD constraint - MUST enforce)
        if teacher_constraints.max_classes_per_week:
            current_weekly = state.teacher_weekly_load[teacher_id]
```

```
          if current_weekly >= teacher_constraints.max_classes_per_week:
              return False, (f"Teacher {teacher_id} has reached weekly limit "
f"({current_weekly}/{teacher_constraints.max_classes_per_week})")

      return True, None

   @staticmethod
   def check_minimum_thresholds(state: ScheduleState, teacher_constraints:
Optional["TimetableConstraint"]) -> list[str]:
      """
      Check if teachers meet minimum workload thresholds.

      This generates WARNINGS only (soft constraint).
      Used at the end of generation to notify admins.

      Args:
         state: Final schedule state
         teacher_constraints: Teacher workload limits

      Returns:
         List of warning messages for teachers below minimum
      """
      warnings = []

      if not teacher_constraints:
         return warnings

      # Check daily minimums
      if teacher_constraints.min_classes_per_day:
         for teacher_id, daily_schedule in state.teacher_daily_load.items():
            for day, count in daily_schedule.items():
               if count < teacher_constraints.min_classes_per_day:
                  warnings.append(f"Teacher {teacher_id} has only {count} classes
on day {day} " f"(minimum: {teacher_constraints.min_classes_per_day})")

      # Check weekly minimums
      if teacher_constraints.min_classes_per_week:
         for teacher_id, weekly_count in state.teacher_weekly_load.items():
            if weekly_count < teacher_constraints.min_classes_per_week:
               warnings.append(f"Teacher {teacher_id} has only {weekly_count}
classes this week " f"(minimum:
{teacher_constraints.min_classes_per_week})")

      return warnings


#
```

```
=================================================================
=================
# PART 3: CORE SCHEDULING ENGINE
#
=================================================================
=================


class TimetableScheduler:
    """
    Main scheduling algorithm implementing constraint satisfaction problem
(CSP).

    Algorithm Phases:
    1. Grid Initialization - Build empty schedule matrix
    2. Priority Sorting - Order subjects by scheduling difficulty
    3. Greedy Placement - Assign subjects to slots with conflict checking
    4. Optimization - Calculate quality score
    """

    def __init__(self, db: AsyncSession):
        """
        Initialize scheduler with database session.

        Args:
            db: SQLAlchemy async session for database operations
        """
        self.db = db
        self.validator = ConstraintValidator()

    async def initialize_grid(self, school_id: int, working_days: list[int], state:
ScheduleState) -> None:
        """
        Build the empty schedule grid based on school's period structure.

        Handles TWO period models:
        - Day-agnostic: Same periods every day (school_id=1)
        - Day-specific: Different periods per day (school_id=2)

        Args:
            school_id: School ID for fetching periods
            working_days: List of working days (1=Mon, 6=Sat)
            state: ScheduleState to populate

        Raises:
            ValueError: If no active periods found for school
        """
        # Detect period model by checking sample period
        sample_query = select(Period).where(Period.school_id == school_id,
Period.is_active).limit(1)
        sample_result = await self.db.execute(sample_query)
        sample = sample_result.scalar_one_or_none()

        if not sample:
            raise ValueError(f"No active periods found for school {school_id}")

        # Day-specific periods (School 2 model: periods have day_of_week set)
        if sample.day_of_week is not None:
            periods_query = select(Period).where(Period.school_id == school_id,
Period.is_recess.is_(False), Period.is_active).order_by(Period.start_time)

            periods_result = await self.db.execute(periods_query)
            all_periods = periods_result.scalars().all()

            for period in all_periods:
                day = self._day_name_to_number(period.day_of_week)
                if day in working_days:
                    state.grid[day][period.id] = Slot(day=day, period_id=period.id,
period_number=period.period_number, start_time=period.start_time)

        # Day-agnostic periods (School 1 model: periods apply to all days)
        else:
            periods_query = select(Period).where(Period.school_id == school_id,
Period.is_recess.is_(False), Period.is_active,
Period.day_of_week.is_(None)).order_by(Period.period_number)

            periods_result = await self.db.execute(periods_query)
            all_periods = periods_result.scalars().all()

            # Replicate periods across all working days
            for day in working_days:
                for period in all_periods:
                    state.grid[day][period.id] = Slot(day=day, period_id=period.id,
period_number=period.period_number, start_time=period.start_time)

    def _day_name_to_number(self, day_name: str) -> int:
        """
        Convert day name to ISO day number.

        Args:
            day_name: Full day name (e.g., "Monday")

        Returns:
            Integer 1-7 (Monday=1, Sunday=7)
        """
        mapping = {"Monday": 1, "Tuesday": 2, "Wednesday": 3, "Thursday": 4,
"Friday": 5, "Saturday": 6, "Sunday": 7}
        return mapping.get(day_name, 1)

    def _sort_subjects_by_priority(self, requirements: list[SubjectRequirement])
-> list[SubjectRequirement]:
        """
        Sort subjects for optimal scheduling order.

        Priority (hardest to easiest):
        1. Consecutive period requirements (labs need 2-3 slots together)
        2. Core subjects (must be in morning, harder to place)
        3. High frequency subjects (5-6 periods/week need more slots)

        Args:
            requirements: List of subject requirements

        Returns:
            Sorted list with hardest-to-schedule subjects first
        """

        def priority_score(req: SubjectRequirement) -> tuple[int, int, int]:
            return (1 if req.requires_consecutive else 0, 1 if req.is_core else 0,
req.periods_per_week)  # Labs first  # Core second  # High frequency third

        return sorted(requirements, key=priority_score, reverse=True)

    async def schedule_subject(self, requirement: SubjectRequirement, state:
ScheduleState, constraints: list[ConstraintRule], academic_year_id: int,
teacher_constraints: Optional[TimetableConstraint] = None, subject_name: str
= "") -> int:
        """
        Schedule all periods for a single subject using greedy algorithm with
teacher workload enforcement.

        Algorithm (with constraint priorities):
        1. HIGHEST PRIORITY: Enforce teacher workload limits (daily/weekly max)
        2. MEDIUM PRIORITY: Prioritize core subjects in early periods
        3. LOWEST PRIORITY: Even distribution across days

        Args:
            requirement: Subject requirement specification
            state: Current schedule state
            constraints: List of constraint rules (legacy)
            academic_year_id: Academic year for timetable entries
            teacher_constraints: Teacher workload limits (NEW)
            subject_name: Name of subject for core detection (NEW)

        Returns:
            Number of successfully placed periods
        """
        placed_count = 0
        consecutive_count = 2 if requirement.requires_consecutive else 1

        # Determine if this is a core subject
        is_core_subject = False
        if teacher_constraints and teacher_constraints.prioritize_core_subjects:
            core_names = teacher_constraints.core_subject_names or []
            is_core_subject = any(core in subject_name for core in core_names)

        # Try to place all requested periods
        for attempt_num in range(requirement.periods_per_week):
            placed = False

            # Prioritize days with fewer placements of this subject (even
distribution)
            day_candidates = sorted(
                state.working_days,
                key=lambda d: (
                    # Penalty if subject already on this day
                    len([p for p in state.subject_placements[requirement.subject_id] if
p[0] == d]),
                    # Add randomness to avoid clustering
                    random.random(),
                ),
            )

            for day in day_candidates:
                # HIGHEST PRIORITY: Check teacher workload constraints
                can_assign, workload_reason =
TeacherWorkloadValidator.can_assign_teacher(requirement.teacher_id, day,
state, teacher_constraints)
                if not can_assign:
                    # Skip this day - teacher has reached limit
                    continue

                # Validate minimum gap constraint
                is_valid, error =
self.validator.validate_min_gap_days(requirement.subject_id, day, state,
requirement.min_gap_days)
                if not is_valid:
                    continue
                # Get available slots for this day
                available = [slot for slot in state.grid[day].values() if not
slot.is_occupied]

                if not available:
                    continue

                # Find appropriate slots based on requirements
                if consecutive_count > 1:
                    # Lab subject: need consecutive periods
                    slots = self.validator.find_consecutive_slots(state, day,
consecutive_count, available)
                else:
                    # Single period subject
                    # MEDIUM PRIORITY: Core subjects prefer morning slots (periods
1-3)
                    if is_core_subject:
                        morning_slots = [s for s in available if s.period_number <= 3]
                        candidate_slots = morning_slots if morning_slots else available
                    elif requirement.is_core:
                        # Fallback to requirement flag
                        morning_slots = [s for s in available if s.period_number <= 4]
                        candidate_slots = morning_slots if morning_slots else available
                    else:
                        candidate_slots = available

                    slots = [random.choice(candidate_slots)] if candidate_slots else
None

                if not slots:
                    continue

                # Validate all constraints for selected slots
                all_valid = True
                validation_errors = []

                for slot in slots:
                    # Teacher availability check
                    is_valid, error =
self.validator.validate_teacher_availability(requirement.teacher_id, day,
slot.period_id, state, constraints)
                    if not is_valid:
                        all_valid = False
                        validation_errors.append(error)
                        break
                    # Subject timing restriction check
                    is_valid, error =
self.validator.validate_subject_timing(requirement.subject_id,
slot.period_number, slot.start_time, constraints)
                    if not is_valid:
                        all_valid = False
                        validation_errors.append(error)
                        break

                if all_valid:
                    # Successfully validated - place the subject in all slots
                    for slot in slots:
                        slot.is_occupied = True
                        slot.teacher_id = requirement.teacher_id
                        slot.subject_id = requirement.subject_id

                        # Update tracking structures
                        state.teacher_schedule[requirement.teacher_id]
[day].add(slot.period_id)
                        state.class_schedule[day].add(slot.period_id)
                        state.subject_placements[requirement.subject_id].append((day,
slot.period_id))

                        # NEW: Update teacher workload counters
                        state.teacher_daily_load[requirement.teacher_id][day] += 1
                        state.teacher_weekly_load[requirement.teacher_id] += 1

                        # Create timetable entry (as dict, converted to Pydantic later)
                        state.entries.append(
                            {"class_id": state.class_id, "subject_id":
requirement.subject_id, "teacher_id": requirement.teacher_id, "period_id":
slot.period_id, "day_of_week": day, "academic_year_id": academic_year_id,
"school_id": state.school_id}
                        )

                    placed_count += 1
                    placed = True
                    break
                else:
                    # Log validation failures as warnings
                    for err in validation_errors:
                        if err not in state.warnings:  # Avoid duplicate warnings
                            state.warnings.append(err)

            if not placed:
                # Could not place this period
                reason = "teacher workload limit reached" if teacher_constraints else
```

```
"no available slots"
                state.warnings.append(f"Could not place period {placed_count + 1}/
{requirement.periods_per_week} " f"for Subject {requirement.subject_id}
({reason})")
                break  # Stop trying if we fail once (avoid infinite loop)

        return placed_count

    def calculate_optimization_score(self, state: ScheduleState) -> float:
        """
        Calculate timetable quality score (0-100).

        Score Components:
        - 60%: Placement success rate (occupied slots / total requested)
        - 20%: Distribution evenness (periods spread across days)
        - 20%: Core subjects in morning (placeholder, needs subject metadata)

        Args:
            state: Current schedule state

        Returns:
            Float between 0.0 and 100.0
        """
        # Component 1: Placement success (60 points)
        total_slots = sum(len(periods) for periods in state.grid.values())
        occupied_slots = sum(1 for day_slots in state.grid.values() for slot in
day_slots.values() if slot.is_occupied)
        placement_score = (occupied_slots / total_slots * 60) if total_slots > 0
else 0

        # Component 2: Distribution evenness (20 points)
        days_with_classes = len([d for d in state.working_days if
state.class_schedule[d]])
        distribution_score = (days_with_classes / len(state.working_days) * 20) if
state.working_days else 0

        # Component 3: Morning core subjects (20 points) - currently placeholder
        # TODO: Implement based on subject.category metadata
        morning_core_score = 20.0

        return round(placement_score + distribution_score + morning_core_score,
2)


#
============================================================
=================
# PART 4: SERVICE CLASS (Main Public API)
#
=================================================================
==================


class TimetableGenerationService:
    """
    Service class for timetable generation operations (ASYNC).

    ARCHITECTURE NOTE:
    Follows cart_service.py pattern - all methods are instance methods
    that receive db session in constructor.
    """

    def __init__(self, db: AsyncSession):
        """
        Initialize service with database session.

        Args:
            db: SQLAlchemy async session
        """
        self.db = db
        self.scheduler = TimetableScheduler(db)

    async def generate_timetable(self, request: TimetableGenerateRequest,
school_id: int) -> TimetableGenerateResponse:
        """
        Generate a complete timetable for a class.

        Business Logic:
        1. Initialize empty schedule grid
        2. Sort subjects by priority (hardest first)
        3. Greedily assign subjects to slots
        4. Calculate quality metrics
        5. Persist to database (if not dry_run)

        Args:
            request: Generation parameters (class, subjects, constraints)
            school_id: School ID from current_profile (security context)

        Returns:
            TimetableGenerateResponse with entries, warnings, and metrics
        """
        import time as time_module

        start_time = time_module.time()
        # Initialize state
        state = ScheduleState(request×class_id, school_id, request.working_days)

        try:
            # Phase 0: Fetch subject names for better error messages
            subject_ids = [req.subject_id for req in request.subject_requirements]
            subject_query =
select(Subject).where(Subject.subject_id.in_(subject_ids))
            subject_result = await self.db.execute(subject_query)
            subjects_map = {s.subject_id: s.name for s in
subject_result.scalars().all()}

            # Phase 1: Build schedule grid
            await self.scheduler.initialize_grid(school_id, request.working_days,
state)

            # Phase 2: Sort subjects by priority
            sorted_requirements =
self.scheduler._sort_subjects_by_priority(request.subject_requirements)

            # Phase 3: Schedule each subject with teacher workload enforcement
            unassigned_subjects = []
            for req in sorted_requirements:
                subject_name = subjects_map.get(req.subject_id, f"Subject
{req.subject_id}")
                placed_count = await
self.scheduler.schedule_subject(requirement=req, state=state,
constraints=request.constraints, academic_year_id=request.academic_year_id,
teacher_constraints=request.teacher_constraints,
subject_name=subject_name)

                # Track partially/fully unassigned subjects
                if placed_count < req.periods_per_week:
                    unassigned_subjects.append(
                        UnassignedSubjectInfo(subject_id=req×subject_id,
subject_name=subject_name, requested_periods=req.periods_per_week,
assigned_periods=placed_count, reason="Insufficient slots, teacher workload
limit, or constraint conflicts")
                    )

            # Phase 3.5: Check minimum teacher workload thresholds (soft
constraint warnings)
            if request.teacher_constraints:
                workload_warnings =
TeacherWorkloadValidator.check_minimum_thresholds(state,
request.teacher_constraints)
            state.warnings.extend(workload_warnings)

         # Phase 4: Persist to database (if not dry run)
         generated_entries = []

         if not request.dry_run:
            # Bulk insert timetable entries and keep references
            db_entries = []
            for entry_data in state.entries:
               db_entry = Timetable(**entry_data)
               self.db.add(db_entry)
               db_entries.append(db_entry)

            # Flush to assign IDs but don't commit yet
            await self.db.flush()

            # Extract IDs for re-querying with eager loading
            entry_ids = [db_entry.id for db_entry in db_entries]

            # Commit the transaction
            await self.db.commit()

            # Re-query with eager loading to populate relationships
            # CRITICAL: Must eagerly load nested relationships (e.g.,
subject.streams, teacher.profile)
            # to avoid MissingGreenlet errors during Pydantic serialization
            from app.models.teacher import Teacher

            reloaded_query = (
               select(Timetable)
               .where(Timetable.id.in_(entry_ids))
               .options(
                  selectinload(Timetable.subject).selectinload(Subject.streams),
                  selectinload(Timetable.teacher).selectinload(Teacher.profile),  #
FIX: Load nested profile
                  selectinload(Timetable.period),
               )
            )
            reloaded_result = await self.db.execute(reloaded_query)
            reloaded_entries = reloaded_result.scalars().all()

            # Serialize to Pydantic models with populated relationships
            for db_entry in reloaded_entries:

generated_entries.append(TimetableEntryOut.model_validate(db_entry))
         else:
                # Dry run: return entries without IDs
                generated_entries = [TimetableEntryOut(id=0, **entry_data,
subject=None, teacher=None, period=None, is_active=True) for entry_data in
state.entries]  # Placeholder for dry run

            # Phase 5: Calculate metrics
            optimization_score = self.scheduler.calculate_optimization_score(state)
            elapsed_time = time_module.time() - start_time

            return TimetableGenerateResponse(
                success=len(state×conflicts) == 0,
                generated_entries=generated_entries,
                unassigned_subjects=unassigned_subjects,
                warnings=state.warnings,
                conflicts=state.conflicts,
                optimization_score=optimization_score,
                generation_metadata={
                    "total_periods_placed": len(state.entries),
                    "time_taken_seconds": round(elapsed_time, 2),
                    "working_days": request.working_days,
                    "total_subjects": len(request.subject_requirements),
                    "algorithm_version": "v1.0-greedy-csp",
                },
            )

        except Exception as e:
            # Handle unexpected errors gracefully
            return TimetableGenerateResponse(
                success=False,
                generated_entries=[],
                conflicts=[ConflictDetail(conflict_type="generation_error", day=0,
period_id=0, details=f"Unexpected error: {str(e)}")],
                optimization_score=0.0,
                generation_metadata={"error": str(e), "error_type":
type(e).__name__},
            )

    async def check_teacher_conflict(self, teacher_id: int, day_of_week: int,
period_id: int, exclude_entry_id: Optional[int] = None) -> tuple[bool,
Optional[str]]:
        """
        Check if placing teacher at this slot creates a conflict.

        Used by frontend for manual drag-and-drop validation.

        Args:
            teacher_id: Teacher to check
            day_of_week: Day number (1=Monday)
            period_id: Period ID
            exclude_entry_id: Optional entry ID to exclude (for updates)

        Returns:
            (has_conflict, conflict_details)
        """
        query = select(Timetable)×where(Timetable×teacher_id == teacher_id,
Timetable.day_of_week == day_of_week, Timetable.period_id == period_id,
Timetable.is_active)

        if exclude_entry_id:
            query = query×where(Timetable×id != exclude_entry_id)

        result = await self.db.execute(query)
        conflict = result.scalar_one_or_none()

        if conflict:
            return True, f"Teacher {teacher_id} already teaching Class
{conflict.class_id} at this time"

        return False, None

    async def swap_timetable_entries(self, entry_1_id: int, entry_2_id: int,
performed_by_user_id: UUID, school_id: int) -> tuple[bool, str,
Optional[list[Timetable]]]:  # Now automatically extracted from JWT in endpoint
        """
        Swap two timetable entries after validating no teacher conflicts occur.

        This method allows principals to manually adjust generated timetables.
        The swap will be rejected if it creates teacher double-booking.

        Args:
            entry_1_id: ID of first timetable entry
            entry_2_id: ID of second timetable entry
            performed_by_user_id: User UUID performing the swap (automatically
extracted from JWT token, references profiles.user_id)
            school_id: School ID for security validation

        Returns:
            (success, message, swapped_entries)
        """
        from datetime import datetime

        # Fetch both entries with eager loading (CRITICAL: Load ALL relationships
for Pydantic validation)
        from app.models.teacher import Teacher
        query1 = (
            select(Timetable)
            .where(Timetable.id == entry_1_id, Timetable.is_active,
Timetable.school_id == school_id)
            .options(
                selectinload(Timetable.subject).selectinload(Subject.streams),
                selectinload(Timetable.teacher).selectinload(Teacher.profile),  # FIX:
Load nested profile to avoid MissingGreenlet
                selectinload(Timetable.period),
            )
        )
        query2 = (
            select(Timetable)
            .where(Timetable.id == entry_2_id, Timetable.is_active,
Timetable.school_id == school_id)
            .options(
                selectinload(Timetable.subject).selectinload(Subject.streams),
                selectinload(Timetable.teacher).selectinload(Teacher.profile),  # FIX:
Load nested profile to avoid MissingGreenlet
                selectinload(Timetable.period),
            )
        )

        result1 = await self.db.execute(query1)
        result2 = await self.db.execute(query2)

        entry1 = result1.scalar_one_or_none()
        entry2 = result2.scalar_one_or_none()

        # Validation 1: Both entries must exist and belong to the school
        if not entry1:
            return False, f"Timetable entry {entry_1_id} not found or inactive", None
        if not entry2:
            return False, f"Timetable entry {entry_2_id} not found or inactive", None

        # Validation 2: Both entries must be editable
        if not entry1.is_editable:
            return False, f"Entry {entry_1_id} is locked and cannot be swapped",
None
        if not entry2.is_editable:
            return False, f"Entry {entry_2_id} is locked and cannot be swapped",
None

        # Validation 3: Check if swapping would create teacher conflicts
        can_swap, conflict_details = await self._can_swap(entry1, entry2)
        if not can_swap:
            return False, f"Cannot swap: {conflict_details}", None

        # Perform the swap - exchange teachers (the actual swap operation)
        entry1_old_teacher = entry1.teacher_id

        entry1.teacher_id = entry2.teacher_id
        entry1.last_modified_by = performed_by_user_id
        entry1.last_modified_at = datetime.utcnow()

        entry2.teacher_id = entry1_old_teacher
        entry2.last_modified_by = performed_by_user_id
        entry2.last_modified_at = datetime.utcnow()

        # Commit the transaction
        try:
            await self.db.commit()

            # Re-fetch entries with eager loading to populate relationships for
Pydantic
            query1_reload = (
                select(Timetable)
                ×where(Timetable×id == entry_1_id)
                .options(
                    selectinload(Timetable.subject).selectinload(Subject.streams),
                    selectinload(Timetable.teacher).selectinload(Teacher.profile),  #
FIX: Load nested profile to avoid MissingGreenlet
                    selectinload(Timetable.period),
                )
            )
            query2_reload = (
                select(Timetable)
                ×where(Timetable×id == entry_2_id)
                .options(
                    selectinload(Timetable.subject).selectinload(Subject.streams),
                    selectinload(Timetable.teacher).selectinload(Teacher.profile),  #
FIX: Load nested profile to avoid MissingGreenlet
                    selectinload(Timetable.period),
                )
            )

            result1_reload = await self.db.execute(query1_reload)
            result2_reload = await self.db.execute(query2_reload)

            entry1_reloaded = result1_reload.scalar_one()
            entry2_reloaded = result2_reload.scalar_one()
            return True, "Timetable entries swapped successfully",
[entry1_reloaded, entry2_reloaded]
        except Exception as e:
            await self.db.rollback()
            return False, f"Database error during swap: {str(e)}", None

    async def _can_swap(self, entry1: Timetable, entry2: Timetable) ->
tuple[bool, Optional[str]]:
        """
        Private helper to validate if two entries can be swapped without conflicts.

        Checks:
        1. Teacher 1 is not already teaching another class at entry2's time
        2. Teacher 2 is not already teaching another class at entry1's time

        Args:
            entry1: First timetable entry
            entry2: Second timetable entry

        Returns:
            (can_swap, conflict_reason)
        """
        # Check if entry1's teacher has a conflict at entry2's time
        conflict1_exists, conflict1_details = await
self.check_teacher_conflict(teacher_id=entry1.teacher_id,
day_of_week=entry2.day_of_week, period_id=entry2.period_id,
exclude_entry_id=entry1.id)  # Exclude entry1 itself

        if conflict1_exists:
            return False, f"Teacher {entry1.teacher_id} {conflict1_details}"

        # Check if entry2's teacher has a conflict at entry1's time
        conflict2_exists, conflict2_details = await
self.check_teacher_conflict(teacher_id=entry2.teacher_id,
day_of_week=entry1.day_of_week, period_id=entry1.period_id,
exclude_entry_id=entry2.id)  # Exclude entry2 itself

        if conflict2_exists:
            return False, f"Teacher {entry2.teacher_id} {conflict2_details}"

        # No conflicts - swap is safe
        return True, None
```

TIMETABLE ENDPOINTS:

```python
# backend/app/api/v1/endpoints/timetable_generation.py
"""
Timetable Generation API Endpoints.

These endpoints allow school admins to generate and validate timetables using AI.

Security:
- All endpoints require Admin role
- User can only generate timetables for their school
- Class/teacher/subject IDs are validated against school ownership

Architecture Pattern (matches carts.py):
- Endpoint validates auth and extracts user context (school_id)
- Service class is instantiated with db session
- All business logic delegated to service layer
"""
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy import update
from sqlalchemy.ext.asyncio import AsyncSession

from app.core.security import get_current_user_profile, require_role
from app.db.session import get_db
from app.models.class_model import Class
from app.models.profile import Profile
from app.models.subject import Subject
from app.models.teacher import Teacher
from app.models.timetable import Timetable
from app.schemas.timetable_schema import (
    ConflictCheckResponse,
    TeacherAvailabilityCheck,
    TimetableEntryOut,
    TimetableGenerateRequest,
    TimetableGenerateResponse,
    TimetableSwapRequest,
    TimetableSwapResponse,
)
from app.services.timetable_generation_service import TimetableGenerationService

router = APIRouter()


@router.post(
```

~~~
    "/generate",
    response_model=TimetableGenerateResponse,
    status_code=status.HTTP_200_OK,
    dependencies=[Depends(require_role("Admin"))],
    summary="Generate Timetable with AI",
    description="""
    **Generate a complete timetable for a class using AI scheduling.**

    Features:
    - Validates all constraints before scheduling
    - Handles consecutive periods (labs)
    - Respects teacher availability
    - Enforces subject timing restrictions (e.g., PE in last 2 periods)
    - Ensures minimum gap between subject occurrences
    - Prioritizes core subjects in morning slots
    - Supports both dry-run (preview) and actual save

    Returns detailed metrics including:
    - Successfully placed entries
    - Unassigned subjects with reasons
    - Warnings (soft constraint violations)
    - Conflicts (hard constraint violations)
    - Optimization score (0-100)
    """,
)
async def generate_class_timetable(
    request: TimetableGenerateRequest,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Generate a timetable for a class with intelligent constraint handling.

    **Request Body Example:**
    ```json
    {
      "class_id": 19,
      "academic_year_id": 2,
      "working_days": [1, 2, 3, 4, 5, 6],
      "subject_requirements": [
        {
          "subject_id": 2,
          "teacher_id": 13,
          "periods_per_week": 5,
          "is_core": true,
          "requires_consecutive": false,
          "min_gap_days": 1
~~~

```
      }
     ],
     "constraints": [],
     "dry_run": true
   }
   ```
   """
   # Security Validation 1: Verify class belongs to user's school
   target_class = await db.get(Class, request.class_id)
   if not target_class or target_class.school_id != current_profile.school_id:
      raise HTTPException(
         status_code=status.HTTP_403_FORBIDDEN,
         detail="The specified class does not belong to your school.",
)

   # Security Validation 2: Verify all teachers belong to user's school
   for req in request.subject_requirements:
      teacher = await db.get(Teacher, req.teacher_id)
      if not teacher or teacher.school_id != current_profile.school_id:
         raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail=f"Teacher {req.teacher_id} does not belong to your school.",
)

   # Security Validation 3: Verify all subjects belong to user's school
   for req in request.subject_requirements:
      subject = await db.get(Subject, req.subject_id)
      if not subject or subject.school_id != current_profile.school_id:
         raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail=f"Subject {req.subject_id} does not belong to your school.",
)

   # Business Logic: Delegate to service
   try:
      service = TimetableGenerationService(db)
      response = await service.generate_timetable(request=request,
school_id=current_profile.school_id) # Security context from JWT
      return response
   except Exception as e:
      # If any exception occurs, the get_db() dependency will handle rollback
      # We just need to return a proper HTTP error
      raise
HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
detail=f"Timetable generation failed: {str(e)}")
```
```

~~~
@router.post(
    "/check-conflict",
    response_model=ConflictCheckResponse,
    dependencies=[Depends(require_role("Admin"))],
    summary="Check for Scheduling Conflicts",
    description="""
    **Validate if a teacher placement would create a conflict.**

    Use this before drag-and-drop operations or manual edits to prevent:
    - Teacher double-booking (same teacher, same time)
    - Invalid period assignments

    This is used by the frontend for real-time validation during manual edits.
    """,
)
async def check_scheduling_conflict(
    check: TeacherAvailabilityCheck,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Check if assigning a teacher to a slot creates a conflict.

    **Example Request:**
    ```json
    {
      "teacher_id": 13,
      "class_id": 19,
      "day_of_week": 1,
      "period_id": 2
    }
    ```
    """
    # Security: Verify teacher belongs to user's school
    teacher = await db.get(Teacher, check.teacher_id)
    if not teacher or teacher.school_id != current_profile.school_id:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Teacher does not belong to your school.",
        )

    # Security: Verify class belongs to user's school
    target_class = await db.get(Class, check.class_id)
    if not target_class or target_class.school_id != current_profile.school_id:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
~~~

```
            detail="Class does not belong to your school.",
        )

    # Business Logic: Delegate to service
    service = TimetableGenerationService(db)
    has_conflict, details = await
service.check_teacher_conflict(teacher_id=check.teacher_id,
day_of_week=check.day_of_week, period_id=check.period_id)

    return ConflictCheckResponse(has_conflict=has_conflict,
conflict_type="teacher_double_booking" if has_conflict else None,
details=details)


@router.delete(
    "/clear/{class_id}",
    status_code=status.HTTP_204_NO_CONTENT,
    dependencies=[Depends(require_role("Admin"))],
    summary="Clear Timetable for Class",
    description="Soft-delete all timetable entries for a class (sets
is_active=False)",
)
async def clear_class_timetable(
    class_id: int,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Clear all timetable entries for a class to start fresh.

    This is useful when:
    - Regenerating a timetable with different constraints
    - Starting a new academic year
    - Fixing major scheduling issues

    **Note:** This is a soft-delete (is_active=False), so data is preserved.
    """
    # Security check
    target_class = await db.get(Class, class_id)
    if not target_class or target_class.school_id != current_profile.school_id:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="The specified class does not belong to your school.",
        )

    # Business Logic: Soft-delete timetable entries
    stmt = update(Timetable).where(Timetable.class_id == class_id,
Timetable.is_active).values(is_active=False)

    await db.execute(stmt)
    await db.commit()

    return None


@router.post(
    "/swap",
    response_model=TimetableSwapResponse,
    status_code=status.HTTP_200_OK,
    dependencies=[Depends(require_role("Admin"))],
    summary="Manually Swap Two Timetable Entries",
    description="""
    **Manually swap two timetable entries for principal-level adjustments.**

    This endpoint allows school administrators to fine-tune generated timetables
    without regenerating the entire schedule. The swap will be rejected if it
    creates teacher conflicts (double-booking).

    Use Cases:
    - Adjust generated timetable to match teacher preferences
    - Fix specific scheduling issues without full regeneration
    - Accommodate last-minute teacher availability changes

    Validation:
    - Both entries must belong to the same school
    - Both entries must be editable (is_editable=True)
    - Swap must not create teacher double-booking
    - All changes are logged with user ID and timestamp

    Returns:
    - Success response with updated entries
    - Failure response with conflict details
    """,
)
async def swap_timetable_entries(
    swap_request: TimetableSwapRequest,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Swap two timetable entries after validating no conflicts occur.

    **Request Body Example:**
```

~~~
    ```json
    {
      "class_id": 19,
      "entry_1_id": 145,
      "entry_2_id": 203
    }
    ```

    **Success Response:**
    ```json
    {
      "success": true,
      "message": "Timetable entries swapped successfully",
      "swapped_entries": [
        {"id": 145, "day_of_week": 2, "period_id": 3, ...},
        {"id": 203, "day_of_week": 4, "period_id": 5, ...}
      ],
      "conflict_details": null
    }
    ```

    **Failure Response:**
    ```json
    {
      "success": false,
      "message": "Cannot swap: Teacher 13 already teaching Class 20 at this
time",
      "swapped_entries": null,
      "conflict_details": "Teacher double-booking detected"
    }
    ```
    """
    # Security Validation: Verify class belongs to user's school
    target_class = await db.get(Class, swap_request.class_id)
    if not target_class or target_class.school_id != current_profile.school_id:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="The specified class does not belong to your school.",
        )

    # Business Logic: Delegate to service
    # Extract user_id from authenticated profile (JWT token)
    service = TimetableGenerationService(db)
    success, message, swapped_entries = await service.swap_timetable_entries(
        entry_1_id=swap_request.entry_1_id, entry_2_id=swap_request.entry_2_id,
performed_by_user_id=current_profile.user_id,
~~~

```
school_id=current_profile.school_id  # Extract from JWT
    )

    # Convert Timetable models to TimetableEntryOut schemas
    # The schema uses from_attributes=True so it will auto-map the
relationships
    swapped_entries_out = None
    if swapped_entries:
        swapped_entries_out = [TimetableEntryOut.model_validate(entry) for
entry in swapped_entries]

    return TimetableSwapResponse(success=success, message=message,
swapped_entries=swapped_entries_out, conflict_details=None if success else
message)



# backend/app/api/v1/endpoints/timetable.py

from datetime import date
from enum import Enum

from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlalchemy.ext.asyncio import AsyncSession

from app.core.security import get_current_user_profile, require_role
from app.db.session import get_db
from app.models.class_model import Class
from app.models.period import Period
from app.models.profile import Profile
from app.models.subject import Subject
from app.models.teacher import Teacher
from app.schemas.timetable_schema import (
    TimetableEntryCreate,
    TimetableEntryOut,
    TimetableEntryUpdate,
)
from app.schemas.timetable_schema import TimetableEntryOut as
TimetableOut
from app.services import timetable_service

router = APIRouter()


class ScheduleTargetType(str, Enum):
    CLASS = "class"
    TEACHER = "teacher"
    STUDENT = "student"


# Admin only: Create a new timetable entry
@router.post(
    "/",
    response_model=TimetableEntryOut,
    status_code=status.HTTP_201_CREATED,
    dependencies=[Depends(require_role("Admin"))],
)
async def create_new_timetable_entry(
    timetable_in: TimetableEntryCreate,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Create a new timetable entry. Admin only.
    """
    timetable_in.school_id = current_profile.school_id

    # CRITICAL SECURITY FIX: Verify that all foreign keys belong to the user's
school.
    # We fetch each parent object to confirm its existence and school_id.

    # Verify Class
    target_class = await db.get(Class, timetable_in.class_id)
    if not target_class or target_class.school_id != current_profile.school_id:
        raise HTTPException(
            status_code=403,
            detail="The specified class does not belong to your school.",
        )

    # Verify Subject
    target_subject = await db.get(Subject, timetable_in.subject_id)
    if not target_subject or target_subject.school_id !=
current_profile.school_id:
        raise HTTPException(
            status_code=403,
            detail="The specified subject does not belong to your school.",
        )

    # Verify Teacher
    target_teacher = await db.get(Teacher, timetable_in.teacher_id)
    if not target_teacher or target_teacher.school_id !=
current_profile.school_id:
        raise HTTPException(
            status_code=403,
            detail="The specified teacher does not belong to your school.",
        )

    # Verify Period
    target_period = await db.get(Period, timetable_in.period_id)
    if not target_period or target_period.school_id != current_profile.school_id:
        raise HTTPException(
            status_code=403,
            detail="The specified period does not belong to your school.",
        )

    return await timetable_service.create_timetable_entry(db=db,
timetable_in=timetable_in)


# Student/Parent only: Get timetable for a specific class
@router.get(
    "/classes/{class_id}",
    response_model=list[TimetableEntryOut],
    dependencies=[Depends(require_role("Admin", "Teacher", "Student",
"Parent"))],
)
async def get_timetable_for_class(
    class_id: int,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Get the timetable for a specific class.
    """
    target_class = await db.get(Class, class_id)
    if target_class and target_class.school_id != current_profile.school_id:
        return []
    if target_class is None:
        return []

    timetable = await timetable_service.get_class_timetable(db=db,
class_id=class_id)
    if not timetable:
        raise HTTPException(status_code=404, detail="Timetable not found for
this class.")
    return timetable


# Teacher only: Get personalized timetable
@router.get(
    "/teachers/{teacher_id}",
    response_model=list[TimetableEntryOut],
    dependencies=[Depends(require_role("Teacher"))],
)
async def get_timetable_for_teacher(
    teacher_id: int,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Get the personalized timetable for a specific teacher.
    """
    target_teacher = await db.get(Teacher, teacher_id)
    if not target_teacher or target_teacher.school_id !=
current_profile.school_id:
        raise HTTPException(status_code=404, detail="Teacher not found.")

    timetable = await timetable_service.get_teacher_timetable(db=db,
teacher_id=teacher_id)
    if not timetable:
        raise HTTPException(status_code=404, detail="Timetable not found for
this teacher.")
    return timetable


@router.get(
    "/teacher/{teacher_id}/schedule",
    response_model=list[TimetableOut],
    dependencies=[Depends(require_role("Teacher"))],
)
async def get_teacher_schedule(
    teacher_id: int,
    schedule_date: date
    | None = Query(
        None,
        description="Optional date filter (YYYY-MM-DD) to retrieve a specific
day's schedule.",
    ),
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """Return either the full timetable or a specific day's schedule for a
teacher."""

    target_teacher = await db.get(Teacher, teacher_id)
    if not target_teacher or target_teacher.school_id !=
current_profile.school_id:
        raise HTTPException(status_code=404, detail="Teacher not found.")

    if schedule_date:
        return await timetable_service.get_schedule_for_day(
            db=db,
            school_id=current_profile.school_id,
            target_type="teacher",
            target_id=teacher_id,
            schedule_date=schedule_date,
        )

    return await timetable_service.get_teacher_timetable(db=db,
teacher_id=teacher_id)


# Admin only: Update an existing timetable entry
@router.put(
    "/{entry_id}",
    response_model=TimetableEntryOut,
    dependencies=[Depends(require_role("Admin"))],
)
async def update_timetable_entry(
    entry_id: int,
    timetable_in: TimetableEntryUpdate,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    db_obj = await timetable_service.get_timetable_entry_by_id(db, entry_id)
    if not db_obj or db_obj.school_id != current_profile.school_id:
        raise HTTPException(status_code=404, detail="Timetable entry not
found.")
    return await timetable_service.update_timetable_entry(db, db_obj=db_obj,
timetable_in=timetable_in)


# Admin only: Soft-delete an existing timetable entry
@router.delete(
    "/{entry_id}",
    status_code=status.HTTP_204_NO_CONTENT,
    dependencies=[Depends(require_role("Admin"))],
)
async def delete_timetable_entry(
    entry_id: int,
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Soft-deletes a timetable entry by setting its is_active flag to false.
    """
    db_obj = await timetable_service.get_entry_with_details(db, entry_id)
    if not db_obj or db_obj.school_id != current_profile.school_id:
        raise HTTPException(status_code=404, detail="Timetable entry not
found.")

    deleted_entry = await timetable_service.soft_delete_timetable_entry(db,
entry_id=entry_id)
    if not deleted_entry:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Active timetable entry with id {entry_id} not found",
        )
    return None  # Return 204 No Content on success


@router.get("/schedule-for-day", response_model=list[TimetableOut])
async def get_schedule(
    target_type: ScheduleTargetType,
    target_id: int,
    schedule_date: date = Query(..., description="The date for the schedule in
YYYY-MM-DD format"),
    db: AsyncSession = Depends(get_db),
    current_profile: Profile = Depends(get_current_user_profile),
):
    """
    Get the daily schedule for a specific class, teacher, or student.
    """

    return await timetable_service.get_schedule_for_day(
        db=db,
        school_id=current_profile.school_id,
        target_type=target_type.value,
        target_id=target_id,
        schedule_date=schedule_date,
    )
```