Below is what is what I want to achieve.

Using .Net core. Can you please provide implementation for a school management platform that can handle many schools. The platform should allow teachers to upload school results, students, their grade/class, course/subjects, assign students to courses/subjects. We must have attendance register, notifications to parents, social media interaction including WhatsApp. Parents should be able to print term results, school must be able to do the same.. Please provide the service layer implementations, models/entities, ef core relationships

can we accomodate the grade to be dynamic for both symbols and unit. in other countries like zimbabwe they might use units to score/grade they must be able to set what each grade/score is what unit(1,2,3,4..etc) and what sysmbol it is (A,B C.D,etc).... can we add timetable genrattion for a class or per student of subject specific which can then be corrected according to the school rules.... Can we have all these changes and the previous suggestions be put into one detailed response

Can the grading scheme be defined by one body and then all school entities can use those scales.

If capturing the answers as handwritten and determine what they were trying to say can be done on the device level instead of going to the cloud that would be ideal and the cloud will just mark to answer against teacher saved answers but it is not possible to determine the handwritting on the device then it can compare with the stored model... If possible can you define all the steps need to store this model training data and how to use it in the app. Please be very detailed because this can be the grand feature of the app. Can we then have an implementation in detail of the system/Ai generating alphabets, sentences, numbers, etc then the student captures it using handwritten responses so to personalize the models once all has been captured if there were any not clear or model failed to recognize the capture handwritting it can the assign to respective class teacher to confirm. Once all is confirmed it can the be trained on the clouded and the stored locally and on the cloud. Which will allow for the checking for the answers to happen locally before being sent to the cloud to be compared to the teacher's answers. The implementation must be comprehensive and detailed. the platform must have a feature for online schooling for school that would like to offer online schooling. a feature for assign work(homework, classwork, assignment). work that can be loaded on the platform should be done on the platform and the platform can grade the work and assign the mark to the student contionous evaluation marks. That platform should have services for online tutors to offer their services to student, they can offer the service as one on one or teacher to students(one to many), all sessions must have video integration. tutors can list their availability, subjects/courses they tutor and potential student can register for a slot and pay fee, this can be once off or recurring session paid weekly or monthly. Make sure all the exam stuff are also accommodated and implemented

The following is the versions of code you provide. Some do not have implementations, some have placeholders. Please complete the code and compare the two versions and merge the code and return the final comprehensive solution implementation. Please no placeholders everything must have detailed implementations. If the two versions have an implementation for something please user better version. Also improve the code where you see it needs to be improved. When providing code and continuing please continue where you would have left on the previous code instead of cutting and thenredoing the same thing. You should be structured.

Here are the implementations you provided

First Version

// =================== MODELS/ENTITIES ===================

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

// Core System Entities

public class GradingBody

{

public int Id { get; set; }

public string Name { get; set; } // e.g., "Zimbabwe Education Board", "Cambridge"

public string Country { get; set; }

public string Description { get; set; }

public DateTime CreatedAt { get; set; }

public List<GradingScheme> GradingSchemes { get; set; } = new();

public List<School> Schools { get; set; } = new();

}

public class GradingScheme

{

public int Id { get; set; }

public string Name { get; set; } // e.g., "Primary Grading", "Secondary Grading"

public int GradingBodyId { get; set; }

public GradingBody GradingBody { get; set; }

public List<GradeScale> GradeScales { get; set; } = new();

public List<Subject> Subjects { get; set; } = new();

}

public class GradeScale

{

public int Id { get; set; }

public int GradingSchemeId { get; set; }

public GradingScheme GradingScheme { get; set; }

public string Symbol { get; set; } // A, B, C, D, F

public int? Unit { get; set; } // 1, 2, 3, 4, 5 (nullable for systems that don't use units)

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public string Description { get; set; } // "Excellent", "Good", "Average"

public decimal GradePoint { get; set; } // For GPA calculations

}

public class School

{

public int Id { get; set; }

public string Name { get; set; }

public string Address { get; set; }

public string Phone { get; set; }

public string Email { get; set; }

public string Logo { get; set; }

public int GradingBodyId { get; set; }

public GradingBody GradingBody { get; set; }

public bool IsOnlineSchoolingEnabled { get; set; }

public DateTime CreatedAt { get; set; }

// Navigation properties

public List<Class> Classes { get; set; } = new();

public List<Teacher> Teachers { get; set; } = new();

public List<Student> Students { get; set; } = new();

public List<Subject> Subjects { get; set; } = new();

public List<SchoolYear> SchoolYears { get; set; } = new();

public List<Exam> Exams { get; set; } = new();

public List<OnlineTutor> OnlineTutors { get; set; } = new();

}

public class SchoolYear

{

public int Id { get; set; }

public string Name { get; set; } // "2024/2025"

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<Term> Terms { get; set; } = new();

}

public class Term

{

public int Id { get; set; }

public string Name { get; set; } // "Term 1", "First Semester"

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public int SchoolYearId { get; set; }

public SchoolYear SchoolYear { get; set; }

public List<Result> Results { get; set; } = new();

public List<Exam> Exams { get; set; } = new();

}

public class Class

{

public int Id { get; set; }

public string Name { get; set; } // "Grade 1A", "Form 4B"

public string Level { get; set; } // "Primary", "Secondary"

public int Capacity { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int? ClassTeacherId { get; set; } // Head teacher

public Teacher ClassTeacher { get; set; }

public List<Student> Students { get; set; } = new();

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Timetable> Timetables { get; set; } = new();

public List<Attendance> Attendances { get; set; } = new();

}

public class Subject

{

public int Id { get; set; }

public string Name { get; set; }

public string Code { get; set; } // "MATH101", "ENG101"

public string Description { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int GradingSchemeId { get; set; }

public GradingScheme GradingScheme { get; set; }

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Result> Results { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<OnlineTutorSubject> OnlineTutorSubjects { get; set; } = new();

}

public class ClassSubject

{

public int Id { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public List<TimetableSlot> TimetableSlots { get; set; } = new();

}

// User Management

public abstract class User

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public string Address { get; set; }

public string ProfilePicture { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? LastLogin { get; set; }

public bool IsActive { get; set; }

public string UserType { get; set; } // Discriminator for inheritance

}

public class Teacher : User

{

public string EmployeeId { get; set; }

public string Qualification { get; set; }

public DateTime DateOfHire { get; set; }

public decimal Salary { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Class> ManagedClasses { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<HandwritingValidation> HandwritingValidations { get; set; } = new();

}

public class Student : User

{

public string StudentNumber { get; set; }

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; }

public DateTime EnrollmentDate { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<Parent> Parents { get; set; } = new();

public List<Result> Results { get; set; } = new();

public List<Attendance> Attendances { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<HandwritingSample> HandwritingSamples { get; set; } = new();

public List<OnlineTutoringSession> TutoringSessionsAsStudent { get; set; } = new();

public List<StudentTimetable> StudentTimetables { get; set; } = new();

}

public class Parent : User

{

public string Relationship { get; set; } // Father, Mother, Guardian

public string Occupation { get; set; }

public string WhatsAppNumber { get; set; }

public bool ReceiveNotifications { get; set; }

public bool ReceiveWhatsAppNotifications { get; set; }

public bool ReceiveEmailNotifications { get; set; }

public bool ReceiveSMSNotifications { get; set; }

public List<Student> Children { get; set; } = new();

public List<Notification> Notifications { get; set; } = new();

}

// Assessment & Results

public class Result

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int TermId { get; set; }

public Term Term { get; set; }

public decimal Score { get; set; }

public string Grade { get; set; } // Calculated from GradeScale

public int? Unit { get; set; } // Calculated from GradeScale

public string Comments { get; set; }

public DateTime DateRecorded { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public string AssessmentType { get; set; } // "Continuous", "Exam", "Assignment"

}

public class Exam

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int TermId { get; set; }

public Term Term { get; set; }

public List<ExamSubject> ExamSubjects { get; set; } = new();

}

public class ExamSubject

{

public int Id { get; set; }

public int ExamId { get; set; }

public Exam Exam { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public DateTime ExamDate { get; set; }

public TimeSpan Duration { get; set; }

public decimal TotalMarks { get; set; }

public string Instructions { get; set; }

}

// Attendance

public class Attendance

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

public string Remarks { get; set; }

public int RecordedByTeacherId { get; set; }

public Teacher RecordedByTeacher { get; set; }

}

public enum AttendanceStatus

{

Present,

Absent,

Late,

Excused

}

// Assignment & Homework System

public class Assignment

{

public int Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public AssignmentType Type { get; set; }

public DateTime DueDate { get; set; }

public decimal TotalMarks { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public bool IsOnlinePlatformWork { get; set; }

public string Instructions { get; set; }

public DateTime CreatedAt { get; set; }

public List<AssignmentSubmission> Submissions { get; set; } = new();

public List<AssignmentQuestion> Questions { get; set; } = new();

}

public enum AssignmentType

{

Homework,

Classwork,

Assignment,

Project,

Quiz

}

public class AssignmentQuestion

{

public int Id { get; set; }

public int AssignmentId { get; set; }

public Assignment Assignment { get; set; }

public string Question { get; set; }

public QuestionType Type { get; set; }

public string CorrectAnswer { get; set; }

public decimal Marks { get; set; }

public int OrderIndex { get; set; }

public List<QuestionOption> Options { get; set; } = new(); // For multiple choice

}

public enum QuestionType

{

MultipleChoice,

TrueFalse,

ShortAnswer,

Essay,

Handwritten

}

public class QuestionOption

{

public int Id { get; set; }

public int QuestionId { get; set; }

public AssignmentQuestion Question { get; set; }

public string OptionText { get; set; }

public bool IsCorrect { get; set; }

public char OptionLetter { get; set; } // A, B, C, D

}

public class AssignmentSubmission

{

public int Id { get; set; }

public int AssignmentId { get; set; }

public Assignment Assignment { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public DateTime SubmittedAt { get; set; }

public decimal? Score { get; set; }

public string Feedback { get; set; }

public SubmissionStatus Status { get; set; }

public bool IsAutoGraded { get; set; }

public List<SubmissionAnswer> Answers { get; set; } = new();

}

public enum SubmissionStatus

{

Submitted,

Graded,

Late,

Missing

}

public class SubmissionAnswer

{

public int Id { get; set; }

public int SubmissionId { get; set; }

public AssignmentSubmission Submission { get; set; }

public int QuestionId { get; set; }

public AssignmentQuestion Question { get; set; }

public string Answer { get; set; }

public string HandwrittenImagePath { get; set; } // For handwritten answers

public decimal? Score { get; set; }

public bool IsCorrect { get; set; }

public string ProcessedText { get; set; } // AI-processed handwritten text

public decimal Confidence { get; set; } // AI confidence level

}

// Timetable System

public class Timetable

{

public int Id { get; set; }

public string Name { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SchoolYearId { get; set; }

public SchoolYear SchoolYear { get; set; }

public bool IsActive { get; set; }

public DateTime CreatedAt { get; set; }

public List<TimetableSlot> Slots { get; set; } = new();

}

public class TimetableSlot

{

public int Id { get; set; }

public int TimetableId { get; set; }

public Timetable Timetable { get; set; }

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public int ClassSubjectId { get; set; }

public ClassSubject ClassSubject { get; set; }

public string Room { get; set; }

public SlotType Type { get; set; }

}

public enum SlotType

{

Regular,

Break,

Lunch,

Assembly,

Sports

}

public class StudentTimetable

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int TimetableSlotId { get; set; }

public TimetableSlot TimetableSlot { get; set; }

public bool IsOptional { get; set; } // For elective subjects

}

// Handwriting Recognition System

public class HandwritingSample

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public string ImagePath { get; set; }

public string ExpectedText { get; set; }

public string RecognizedText { get; set; }

public decimal Confidence { get; set; }

public HandwritingType Type { get; set; }

public bool IsValidated { get; set; }

public int? ValidatedByTeacherId { get; set; }

public Teacher ValidatedByTeacher { get; set; }

public DateTime CreatedAt { get; set; }

public bool IsTrainingData { get; set; }

}

public enum HandwritingType

{

Alphabet,

Number,

Word,

Sentence,

Answer

}

public class HandwritingValidation

{

public int Id { get; set; }

public int HandwritingSampleId { get; set; }

public HandwritingSample HandwritingSample { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public string CorrectedText { get; set; }

public DateTime ValidatedAt { get; set; }

public ValidationStatus Status { get; set; }

}

public enum ValidationStatus

{

Pending,

Approved,

Corrected,

Rejected

}

public class PersonalizedModel

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public string ModelPath { get; set; } // Local device path

public string CloudModelPath { get; set; } // Cloud backup path

public DateTime LastTrainingDate { get; set; }

public int SampleCount { get; set; }

public decimal Accuracy { get; set; }

public bool IsDeployedLocally { get; set; }

public bool IsDeployedOnCloud { get; set; }

}

// Online Tutoring System

public class OnlineTutor

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public string Bio { get; set; }

public string Qualifications { get; set; }

public decimal HourlyRate { get; set; }

public bool IsVerified { get; set; }

public decimal Rating { get; set; }

public int TotalSessions { get; set; }

public DateTime CreatedAt { get; set; }

public int? SchoolId { get; set; } // Optional - tutor might be independent

public School School { get; set; }

public List<OnlineTutorSubject> TutorSubjects { get; set; } = new();

public List<TutorAvailability> Availability { get; set; } = new();

public List<OnlineTutoringSession> Sessions { get; set; } = new();

}

public class OnlineTutorSubject

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public string GradeLevel { get; set; } // "Primary", "Secondary", "A-Level"

}

public class TutorAvailability

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public bool IsAvailable { get; set; }

}

public class OnlineTutoringSession

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int? SubjectId { get; set; }

public Subject Subject { get; set; }

public DateTime ScheduledDateTime { get; set; }

public TimeSpan Duration { get; set; }

public SessionType Type { get; set; }

public SessionStatus Status { get; set; }

public decimal Cost { get; set; }

public PaymentFrequency PaymentFrequency { get; set; }

public string MeetingUrl { get; set; }

public string SessionNotes { get; set; }

public int? Rating { get; set; }

public string Review { get; set; }

public DateTime? ActualStartTime { get; set; }

public DateTime? ActualEndTime { get; set; }

}

public enum SessionType

{

OneOnOne,

GroupSession

}

public enum SessionStatus

{

Scheduled,

InProgress,

Completed,

Cancelled,

NoShow

}

public enum PaymentFrequency

{

OneTime,

Weekly,

Monthly

}

// Notifications & Communication

public class Notification

{

public int Id { get; set; }

public string Title { get; set; }

public string Message { get; set; }

public NotificationType Type { get; set; }

public int? ParentId { get; set; }

public Parent Parent { get; set; }

public int? StudentId { get; set; }

public Student Student { get; set; }

public int? TeacherId { get; set; }

public Teacher Teacher { get; set; }

public bool IsRead { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? ReadAt { get; set; }

public NotificationChannel Channel { get; set; }

public string ExternalId { get; set; } // For WhatsApp/SMS tracking

}

public enum NotificationType

{

Attendance,

Results,

Assignment,

Exam,

GeneralInfo,

Payment,

Disciplinary,

TutoringSession

}

public enum NotificationChannel

{

InApp,

Email,

SMS,

WhatsApp,

Push

}

public class SocialMediaPost

{

public int Id { get; set; }

public string Content { get; set; }

public string ImagePath { get; set; }

public SocialMediaPlatform Platform { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public DateTime PostedAt { get; set; }

public string ExternalPostId { get; set; }

public int Likes { get; set; }

public int Shares { get; set; }

public int Comments { get; set; }

}

public enum SocialMediaPlatform

{

Facebook,

Twitter,

Instagram,

WhatsApp,

LinkedIn

}

// =================== DBCONTEXT ===================

using Microsoft.EntityFrameworkCore;

public class SchoolManagementContext : DbContext

{

public SchoolManagementContext(DbContextOptions<SchoolManagementContext> options) : base(options) { }

// Core Entities

public DbSet<GradingBody> GradingBodies { get; set; }

public DbSet<GradingScheme> GradingSchemes { get; set; }

public DbSet<GradeScale> GradeScales { get; set; }

public DbSet<School> Schools { get; set; }

public DbSet<SchoolYear> SchoolYears { get; set; }

public DbSet<Term> Terms { get; set; }

public DbSet<Class> Classes { get; set; }

public DbSet<Subject> Subjects { get; set; }

public DbSet<ClassSubject> ClassSubjects { get; set; }

// Users

public DbSet<User> Users { get; set; }

public DbSet<Teacher> Teachers { get; set; }

public DbSet<Student> Students { get; set; }

public DbSet<Parent> Parents { get; set; }

// Assessment

public DbSet<Result> Results { get; set; }

public DbSet<Exam> Exams { get; set; }

public DbSet<ExamSubject> ExamSubjects { get; set; }

public DbSet<Assignment> Assignments { get; set; }

public DbSet<AssignmentQuestion> AssignmentQuestions { get; set; }

public DbSet<QuestionOption> QuestionOptions { get; set; }

public DbSet<AssignmentSubmission> AssignmentSubmissions { get; set; }

public DbSet<SubmissionAnswer> SubmissionAnswers { get; set; }

// Attendance & Timetable

public DbSet<Attendance> Attendances { get; set; }

public DbSet<Timetable> Timetables { get; set; }

public DbSet<TimetableSlot> TimetableSlots { get; set; }

public DbSet<StudentTimetable> StudentTimetables { get; set; }

// Handwriting Recognition

public DbSet<HandwritingSample> HandwritingSamples { get; set; }

public DbSet<HandwritingValidation> HandwritingValidations { get; set; }

public DbSet<PersonalizedModel> PersonalizedModels { get; set; }

// Online Tutoring

public DbSet<OnlineTutor> OnlineTutors { get; set; }

public DbSet<OnlineTutorSubject> OnlineTutorSubjects { get; set; }

public DbSet<TutorAvailability> TutorAvailabilities { get; set; }

public DbSet<OnlineTutoringSession> OnlineTutoringSessions { get; set; }

// Communication

public DbSet<Notification> Notifications { get; set; }

public DbSet<SocialMediaPost> SocialMediaPosts { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// User inheritance configuration

modelBuilder.Entity<User>()

.HasDiscriminator<string>("UserType")

.HasValue<Teacher>("Teacher")

.HasValue<Student>("Student")

.HasValue<Parent>("Parent");

// Configure relationships

ConfigureGradingSystem(modelBuilder);

ConfigureSchoolStructure(modelBuilder);

ConfigureUserRelationships(modelBuilder);

ConfigureAssessmentSystem(modelBuilder);

ConfigureTimetableSystem(modelBuilder);

ConfigureHandwritingSystem(modelBuilder);

ConfigureTutoringSystem(modelBuilder);

ConfigureCommunicationSystem(modelBuilder);

ConfigureIndexes(modelBuilder);

ConfigureConstraints(modelBuilder);

}

private void ConfigureGradingSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeScale>()

.HasOne(gs => gs.GradingScheme)

.WithMany(gs => gs.GradeScales)

.HasForeignKey(gs => gs.GradingSchemeId);

modelBuilder.Entity<GradingScheme>()

.HasOne(gs => gs.GradingBody)

.WithMany(gb => gb.GradingSchemes)

.HasForeignKey(gs => gs.GradingBodyId);

}

private void ConfigureSchoolStructure(ModelBuilder modelBuilder)

{

modelBuilder.Entity<School>()

.HasOne(s => s.GradingBody)

.WithMany(gb => gb.Schools)

.HasForeignKey(s => s.GradingBodyId);

modelBuilder.Entity<Class>()

.HasOne(c => c.ClassTeacher)

.WithMany(t => t.ManagedClasses)

.HasForeignKey(c => c.ClassTeacherId)

.OnDelete(DeleteBehavior.SetNull);

modelBuilder.Entity<ClassSubject>()

.HasKey(cs => cs.Id);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Class)

.WithMany(c => c.ClassSubjects)

.HasForeignKey(cs => cs.ClassId);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Subject)

.WithMany(s => s.ClassSubjects)

.HasForeignKey(cs => cs.SubjectId);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Teacher)

.WithMany(t => t.ClassSubjects)

.HasForeignKey(cs => cs.TeacherId);

}

private void ConfigureUserRelationships(ModelBuilder modelBuilder)

{

// Student-Parent many-to-many

modelBuilder.Entity<Student>()

.HasMany(s => s.Parents)

.WithMany(p => p.Children)

.UsingEntity<Dictionary<string, object>>(

"StudentParent",

j => j.HasOne<Parent>().WithMany().HasForeignKey("ParentId"),

j => j.HasOne<Student>().WithMany().HasForeignKey("StudentId"));

}

private void ConfigureAssessmentSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Result>()

.Property(r => r.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<AssignmentQuestion>()

.Property(aq => aq.Marks)

.HasPrecision(5, 2);

modelBuilder.Entity<SubmissionAnswer>()

.Property(sa => sa.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<SubmissionAnswer>()

.Property(sa => sa.Confidence)

.HasPrecision(5, 4);

}

private void ConfigureTimetableSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<StudentTimetable>()

.HasKey(st => new { st.StudentId, st.TimetableSlotId });

}

private void ConfigureHandwritingSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<HandwritingSample>()

.Property(hs => hs.Confidence)

.HasPrecision(5, 4);

modelBuilder.Entity<PersonalizedModel>()

.Property(pm => pm.Accuracy)

.HasPrecision(5, 4);

}

private void ConfigureTutoringSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<OnlineTutorSubject>()

.HasKey(ots => new { ots.TutorId, ots.SubjectId });

modelBuilder.Entity<OnlineTutoringSession>()

.Property(ots => ots.Cost)

.HasPrecision(10, 2);

modelBuilder.Entity<OnlineTutor>()

.Property(ot => ot.HourlyRate)

.HasPrecision(10, 2);

modelBuilder.Entity<OnlineTutor>()

.Property(ot => ot.Rating)

.HasPrecision(3, 2);

}

private void ConfigureCommunicationSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Notification>()

.HasOne(n => n.Parent)

.WithMany(p => p.Notifications)

.HasForeignKey(n => n.ParentId)

.OnDelete(DeleteBehavior.Cascade);

}

private void ConfigureIndexes(ModelBuilder modelBuilder)

{

// Performance indexes

modelBuilder.Entity<Student>()

.HasIndex(s => s.StudentNumber)

.IsUnique();

modelBuilder.Entity<Teacher>()

.HasIndex(t => t.EmployeeId)

.IsUnique();

modelBuilder.Entity<User>()

.HasIndex(u => u.Email)

.IsUnique();

modelBuilder.Entity<Attendance>()

.HasIndex(a => new { a.StudentId, a.Date });

modelBuilder.Entity<r>()

.HasIndex(r => new { r.StudentId, r.SubjectId, r.TermId });

}

private void ConfigureConstraints(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeScale>()

.HasCheckConstraint("CK\_GradeScale\_Percentage",

"MinPercentage >= 0 AND MaxPercentage <= 100 AND MinPercentage <= MaxPercentage");

modelBuilder.Entity<TimetableSlot>()

.HasCheckConstraint("CK\_TimetableSlot\_Time", "StartTime < EndTime");

}

}

// =================== DTOS ===================

public class StudentResultDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public string ClassName { get; set; }

public string TermName { get; set; }

public List<SubjectResultDto> SubjectResults { get; set; } = new();

public decimal OverallAverage { get; set; }

public string OverallGrade { get; set; }

public int? OverallUnit { get; set; }

public int Position { get; set; }

public int TotalStudents { get; set; }

}

public class SubjectResultDto

{

public string SubjectName { get; set; }

public decimal Score { get; set; }

public string Grade { get; set; }

public int? Unit { get; set; }

public string Comments { get; set; }

public string TeacherName { get; set; }

}

public class TimetableDto

{

public int Id { get; set; }

public string ClassName { get; set; }

public List<TimetableSlotDto> Slots { get; set; } = new();

}

public class TimetableSlotDto

{

public DayOfWeek DayOfWeek { get; set; }

public string StartTime { get; set; }

public string EndTime { get; set; }

public string SubjectName { get; set; }

public string TeacherName { get; set; }

public string Room { get; set; }

public SlotType Type { get; set; }

}

public class AttendanceReportDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public int TotalDays { get; set; }

public int PresentDays { get; set; }

public int AbsentDays { get; set; }

public int LateDays { get; set; }

public decimal AttendancePercentage { get; set; }

}

public class HandwritingRecognitionDto

{

public string ImageBase64 { get; set; }

public int StudentId { get; set; }

public HandwritingType Type { get; set; }

public string ExpectedText { get; set; }

}

public class TutoringSessionDto

{

public int TutorId { get; set; }

public int StudentId { get; set; }

public int? SubjectId { get; set; }

public DateTime ScheduledDateTime { get; set; }

public TimeSpan Duration { get; set; }

public SessionType Type { get; set; }

public PaymentFrequency PaymentFrequency { get; set; }

}

// =================== SERVICE INTERFACES ===================

public interface IGradingService

{

Task<GradingBody> CreateGradingBodyAsync(GradingBody gradingBody);

Task<GradingScheme> CreateGradingSchemeAsync(GradingScheme gradingScheme);

Task<GradeScale> AddGradeScaleAsync(GradeScale gradeScale);

Task<(string grade, int? unit)> CalculateGradeAsync(decimal score, int gradingSchemeId);

Task<List<GradingBody>> GetAllGradingBodiesAsync();

Task<List<GradingScheme>> GetGradingSchemesByBodyAsync(int gradingBodyId);

}

public interface IStudentService

{

Task<Student> CreateStudentAsync(Student student);

Task<Student> UpdateStudentAsync(Student student);

Task<List<Student>> GetStudentsByClassAsync(int classId);

Task<List<Student>> GetStudentsBySchoolAsync(int schoolId);

Task<Student> GetStudentByNumberAsync(string studentNumber, int schoolId);

Task<bool> AssignStudentToClassAsync(int studentId, int classId);

Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId);

Task<byte[]> GenerateStudentReportCardAsync(int studentId, int termId);

}

public interface ITeacherService

{

Task<Teacher> CreateTeacherAsync(Teacher teacher);

Task<List<Teacher>> GetTeachersBySchoolAsync(int schoolId);

Task<bool> AssignTeacherToSubjectAsync(int teacherId, int classId, int subjectId);

Task<List<ClassSubject>> GetTeacherAssignmentsAsync(int teacherId);

}

public interface IResultService

{

Task<r> RecordResultAsync(r result);

Task<List<r>> BulkRecordResultsAsync(List<r> results);

Task<List<StudentResultDto>> GetClassResultsAsync(int classId, int termId);

Task<byte[]> GenerateClassReportAsync(int classId, int termId);

Task<StudentResultDto> CalculateStudentPositionAsync(int studentId, int termId);

}

public interface IAttendanceService

{

Task<Attendance> RecordAttendanceAsync(Attendance attendance);

Task<List<Attendance>> BulkRecordAttendanceAsync(List<Attendance> attendances);

Task<List<AttendanceReportDto>> GetClassAttendanceReportAsync(int classId, DateTime startDate, DateTime endDate);

Task<AttendanceReportDto> GetStudentAttendanceReportAsync(int studentId, DateTime startDate, DateTime endDate);

}

public interface ITimetableService

{

Task<Timetable> GenerateTimetableAsync(int classId, int schoolYearId);

Task<TimetableDto> GetClassTimetableAsync(int classId);

Task<TimetableDto> GetStudentTimetableAsync(int studentId);

Task<TimetableSlot> UpdateTimetableSlotAsync(TimetableSlot slot);

Task<bool> ValidateTimetableRulesAsync(int timetableId);

Task<List<TimetableSlot>> GetTeacherScheduleAsync(int teacherId, DateTime date);

}

public interface INotificationService

{

Task<Notification> CreateNotificationAsync(Notification notification);

Task SendNotificationAsync(int notificationId);

Task SendBulkNotificationsAsync(List<int> notificationIds);

Task<List<Notification>> GetParentNotificationsAsync(int parentId);

Task<bool> MarkNotificationAsReadAsync(int notificationId);

Task SendWhatsAppNotificationAsync(string phoneNumber, string message);

Task SendEmailNotificationAsync(string email, string subject, string message);

}

public interface IAssignmentService

{

Task<Assignment> CreateAssignmentAsync(Assignment assignment);

Task<AssignmentSubmission> SubmitAssignmentAsync(AssignmentSubmission submission);

Task<AssignmentSubmission> AutoGradeAssignmentAsync(int submissionId);

Task<List<Assignment>> GetClassAssignmentsAsync(int classId);

Task<List<Assignment>> GetStudentAssignmentsAsync(int studentId);

Task<decimal> CalculateContinuousAssessmentMarkAsync(int studentId, int subjectId, int termId);

}

public interface IHandwritingRecognitionService

{

Task<string> RecognizeHandwritingAsync(string imagePath, int studentId);

Task<HandwritingSample> CreateHandwritingSampleAsync(HandwritingRecognitionDto dto);

Task<bool> ValidateHandwritingSampleAsync(int sampleId, string correctedText, int teacherId);

Task TrainPersonalizedModelAsync(int studentId);

Task<PersonalizedModel> DeployModelLocallyAsync(int studentId);

Task<string> GenerateTrainingContentAsync(HandwritingType type, string difficulty = "beginner");

Task<bool> ProcessHandwritingOfflineAsync(string imagePath, int studentId);

}

public interface IOnlineTutoringService

{

Task<OnlineTutor> RegisterTutorAsync(OnlineTutor tutor);

Task<OnlineTutoringSession> BookSessionAsync(TutoringSessionDto sessionDto);

Task<List<OnlineTutor>> SearchTutorsAsync(int? subjectId, string gradeLevel, decimal? maxRate);

Task<List<TutorAvailability>> GetTutorAvailabilityAsync(int tutorId, DateTime date);

Task<OnlineTutoringSession> StartSessionAsync(int sessionId);

Task<OnlineTutoringSession> EndSessionAsync(int sessionId, string notes, int? rating, string review);

Task<decimal> CalculateTutorEarningsAsync(int tutorId, DateTime startDate, DateTime endDate);

}

public interface IExamService

{

Task<Exam> CreateExamAsync(Exam exam);

Task<ExamSubject> AddExamSubjectAsync(ExamSubject examSubject);

Task<List<Exam>> GetSchoolExamsAsync(int schoolId, int termId);

Task<byte[]> GenerateExamTimetableAsync(int examId);

}

public interface ISocialMediaService

{

Task<SocialMediaPost> CreatePostAsync(SocialMediaPost post);

Task<bool> PublishToFacebookAsync(int postId);

Task<bool> PublishToWhatsAppAsync(int postId, List<string> phoneNumbers);

Task<bool> PublishToInstagramAsync(int postId);

Task<List<SocialMediaPost>> GetSchoolPostsAsync(int schoolId);

}

// =================== SERVICE IMPLEMENTATIONS ===================

public class GradingService : IGradingService

{

private readonly SchoolManagementContext \_context;

public GradingService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<GradingBody> CreateGradingBodyAsync(GradingBody gradingBody)

{

gradingBody.CreatedAt = DateTime.UtcNow;

\_context.GradingBodies.Add(gradingBody);

await \_context.SaveChangesAsync();

return gradingBody;

}

public async Task<GradingScheme> CreateGradingSchemeAsync(GradingScheme gradingScheme)

{

\_context.GradingSchemes.Add(gradingScheme);

await \_context.SaveChangesAsync();

return gradingScheme;

}

public async Task<GradeScale> AddGradeScaleAsync(GradeScale gradeScale)

{

\_context.GradeScales.Add(gradeScale);

await \_context.SaveChangesAsync();

return gradeScale;

}

public async Task<(string grade, int? unit)> CalculateGradeAsync(decimal score, int gradingSchemeId)

{

var gradeScale = await \_context.GradeScales

.Where(gs => gs.GradingSchemeId == gradingSchemeId

&& score >= gs.MinPercentage

&& score <= gs.MaxPercentage)

.FirstOrDefaultAsync();

return gradeScale != null ? (gradeScale.Symbol, gradeScale.Unit) : ("F", null);

}

public async Task<List<GradingBody>> GetAllGradingBodiesAsync()

{

return await \_context.GradingBodies

.Include(gb => gb.GradingSchemes)

.ThenInclude(gs => gs.GradeScales)

.ToListAsync();

}

public async Task<List<GradingScheme>> GetGradingSchemesByBodyAsync(int gradingBodyId)

{

return await \_context.GradingSchemes

.Where(gs => gs.GradingBodyId == gradingBodyId)

.Include(gs => gs.GradeScales)

.ToListAsync();

}

}

public class StudentService : IStudentService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

public StudentService(SchoolManagementContext context, IGradingService gradingService)

{

\_context = context;

\_gradingService = gradingService;

}

public async Task<Student> CreateStudentAsync(Student student)

{

student.CreatedAt = DateTime.UtcNow;

student.IsActive = true;

student.UserType = "Student";

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

return student;

}

public async Task<Student> UpdateStudentAsync(Student student)

{

\_context.Students.Update(student);

await \_context.SaveChangesAsync();

return student;

}

public async Task<List<Student>> GetStudentsByClassAsync(int classId)

{

return await \_context.Students

.Where(s => s.ClassId == classId && s.IsActive)

.Include(s => s.Parents)

.OrderBy(s => s.LastName)

.ThenBy(s => s.FirstName)

.ToListAsync();

}

public async Task<List<Student>> GetStudentsBySchoolAsync(int schoolId)

{

return await \_context.Students

.Where(s => s.SchoolId == schoolId && s.IsActive)

.Include(s => s.Class)

.Include(s => s.Parents)

.ToListAsync();

}

public async Task<Student> GetStudentByNumberAsync(string studentNumber, int schoolId)

{

return await \_context.Students

.Where(s => s.StudentNumber == studentNumber && s.SchoolId == schoolId)

.Include(s => s.Class)

.Include(s => s.Parents)

.FirstOrDefaultAsync();

}

public async Task<bool> AssignStudentToClassAsync(int studentId, int classId)

{

var student = await \_context.Students.FindAsync(studentId);

if (student == null) return false;

student.ClassId = classId;

await \_context.SaveChangesAsync();

return true;

}

public async Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId)

{

var student = await \_context.Students

.Include(s => s.Class)

.FirstOrDefaultAsync(s => s.Id == studentId);

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Subject)

.Include(r => r.Subject.GradingScheme)

.ThenInclude(gs => gs.GradeScales)

.Include(r => r.Teacher)

.ToListAsync();

var term = await \_context.Terms.FindAsync(termId);

var subjectResults = results.Select(r => new SubjectResultDto

{

SubjectName = r.Subject.Name,

Score = r.Score,

Grade = r.Grade,

Unit = r.Unit,

Comments = r.Comments,

TeacherName = $"{r.Teacher.FirstName} {r.Teacher.LastName}"

}).ToList();

var overallAverage = results.Any() ? results.Average(r => r.Score) : 0;

var gradingScheme = results.FirstOrDefault()?.Subject.GradingScheme;

var overallGrade = "";

int? overallUnit = null;

if (gradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(overallAverage, gradingScheme.Id);

overallGrade = gradeInfo.grade;

overallUnit = gradeInfo.unit;

}

// Calculate position in class

var classAverages = await \_context.Results

.Where(r => r.TermId == termId && r.Student.ClassId == student.ClassId)

.GroupBy(r => r.StudentId)

.Select(g => new { StudentId = g.Key, Average = g.Average(r => r.Score) })

.OrderByDescending(x => x.Average)

.ToListAsync();

var position = classAverages.FindIndex(x => x.StudentId == studentId) + 1;

return new StudentResultDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

ClassName = student.Class.Name,

TermName = term.Name,

SubjectResults = subjectResults,

OverallAverage = overallAverage,

OverallGrade = overallGrade,

OverallUnit = overallUnit,

Position = position,

TotalStudents = classAverages.Count

};

}

public async Task<byte[]> GenerateStudentReportCardAsync(int studentId, int termId)

{

var results = await GetStudentTermResultsAsync(studentId, termId);

// Here you would use a PDF library like iTextSharp or similar

// For now, returning placeholder

var reportContent = $"""

STUDENT REPORT CARD

Student: {results.StudentName}

Student Number: {results.StudentNumber}

Class: {results.ClassName}

Term: {results.TermName}

SUBJECT RESULTS:

{string.Join("\n", results.SubjectResults.Select(sr =>

$"{sr.SubjectName}: {sr.Score}% ({sr.Grade}{(sr.Unit.HasValue ? $" - Unit {sr.Unit}" : "")}) - {sr.Comments}"))}

OVERALL PERFORMANCE:

Average: {results.OverallAverage:F2}%

Grade: {results.OverallGrade}{(results.OverallUnit.HasValue ? $" - Unit {results.OverallUnit}" : "")}

Position: {results.Position} out of {results.TotalStudents}

""";

return System.Text.Encoding.UTF8.GetBytes(reportContent);

}

}

public class ResultService : IResultService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

private readonly INotificationService \_notificationService;

public ResultService(SchoolManagementContext context, IGradingService gradingService, INotificationService notificationService)

{

\_context = context;

\_gradingService = gradingService;

\_notificationService = notificationService;

}

public async Task<r> RecordResultAsync(r result)

{

// Calculate grade and unit based on score

var subject = await \_context.Subjects

.Include(s => s.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == result.SubjectId);

if (subject?.GradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(result.Score, subject.GradingScheme.Id);

result.Grade = gradeInfo.grade;

result.Unit = gradeInfo.unit;

}

result.DateRecorded = DateTime.UtcNow;

\_context.Results.Add(result);

await \_context.SaveChangesAsync();

// Send notification to parents

await SendResultNotificationToParentsAsync(result);

return result;

}

public async Task<List<r>> BulkRecordResultsAsync(List<r> results)

{

foreach (var result in results)

{

var subject = await \_context.Subjects

.Include(s => s.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == result.SubjectId);

if (subject?.GradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(result.Score, subject.GradingScheme.Id);

result.Grade = gradeInfo.grade;

result.Unit = gradeInfo.unit;

}

result.DateRecorded = DateTime.UtcNow;

}

\_context.Results.AddRange(results);

await \_context.SaveChangesAsync();

// Send notifications

foreach (var result in results)

{

await SendResultNotificationToParentsAsync(result);

}

return results;

}

public async Task<List<StudentResultDto>> GetClassResultsAsync(int classId, int termId)

{

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.Include(s => s.Results.Where(r => r.TermId == termId))

.ThenInclude(r => r.Subject)

.ThenInclude(s => s.GradingScheme)

.ToListAsync();

var studentResults = new List<StudentResultDto>();

foreach (var student in students)

{

var subjectResults = student.Results.Select(r => new SubjectResultDto

{

SubjectName = r.Subject.Name,

Score = r.Score,

Grade = r.Grade,

Unit = r.Unit,

Comments = r.Comments

}).ToList();

var overallAverage = student.Results.Any() ? student.Results.Average(r => r.Score) : 0;

studentResults.Add(new StudentResultDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

SubjectResults = subjectResults,

OverallAverage = overallAverage

});

}

// Calculate positions

var sortedResults = studentResults.OrderByDescending(sr => sr.OverallAverage).ToList();

for (int i = 0; i < sortedResults.Count; i++)

{

sortedResults[i].Position = i + 1;

sortedResults[i].TotalStudents = sortedResults.Count;

}

return sortedResults;

}

public async Task<byte[]> GenerateClassReportAsync(int classId, int termId)

{

var classResults = await GetClassResultsAsync(classId, termId);

var classInfo = await \_context.Classes

.Include(c => c.School)

.FirstOrDefaultAsync(c => c.Id == classId);

var term = await \_context.Terms.FindAsync(termId);

var reportContent = $"""

CLASS PERFORMANCE REPORT

School: {classInfo.School.Name}

Class: {classInfo.Name}

Term: {term.Name}

STUDENT RESULTS:

{string.Join("\n", classResults.Select(sr =>

$"{sr.Position}. {sr.StudentName} ({sr.StudentNumber}) - Average: {sr.OverallAverage:F2}%"))}

CLASS STATISTICS:

Total Students: {classResults.Count}

Class Average: {(classResults.Any() ? classResults.Average(sr => sr.OverallAverage) : 0):F2}%

Highest Score: {(classResults.Any() ? classResults.Max(sr => sr.OverallAverage) : 0):F2}%

Lowest Score: {(classResults.Any() ? classResults.Min(sr => sr.OverallAverage) : 0):F2}%

""";

return System.Text.Encoding.UTF8.GetBytes(reportContent);

}

public async Task<StudentResultDto> CalculateStudentPositionAsync(int studentId, int termId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetStudentTermResultsAsync(studentId, termId);

}

private async Task SendResultNotificationToParentsAsync(r result)

{

var student = await \_context.Students

.Include(s => s.Parents)

.Include(s => s.Class)

.FirstOrDefaultAsync(s => s.Id == result.StudentId);

var subject = await \_context.Subjects.FindAsync(result.SubjectId);

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "New Result Posted",

Message = $"New {subject.Name} result for {student.FirstName}: {result.Score}% ({result.Grade})",

Type = NotificationType.Results,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

}

private async Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId)

{

// Implementation moved to StudentService for better organization

var studentService = new StudentService(\_context, \_gradingService);

return await studentService.GetStudentTermResultsAsync(studentId, termId);

}

}

public class AttendanceService : IAttendanceService

{

private readonly SchoolManagementContext \_context;

private readonly INotificationService \_notificationService;

public AttendanceService(SchoolManagementContext context, INotificationService notificationService)

{

\_context = context;

\_notificationService = notificationService;

}

public async Task<Attendance> RecordAttendanceAsync(Attendance attendance)

{

// Check if attendance already exists for this student and date

var existingAttendance = await \_context.Attendances

.FirstOrDefaultAsync(a => a.StudentId == attendance.StudentId

&& a.Date.Date == attendance.Date.Date);

if (existingAttendance != null)

{

existingAttendance.Status = attendance.Status;

existingAttendance.Remarks = attendance.Remarks;

existingAttendance.RecordedByTeacherId = attendance.RecordedByTeacherId;

}

else

{

\_context.Attendances.Add(attendance);

}

await \_context.SaveChangesAsync();

// Send notification if absent

if (attendance.Status == AttendanceStatus.Absent)

{

await SendAbsenteeNotificationAsync(attendance);

}

return existingAttendance ?? attendance;

}

public async Task<List<Attendance>> BulkRecordAttendanceAsync(List<Attendance> attendances)

{

var results = new List<Attendance>();

foreach (var attendance in attendances)

{

var result = await RecordAttendanceAsync(attendance);

results.Add(result);

}

return results;

}

public async Task<List<AttendanceReportDto>> GetClassAttendanceReportAsync(int classId, DateTime startDate, DateTime endDate)

{

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.ToListAsync();

var attendanceData = await \_context.Attendances

.Where(a => a.ClassId == classId

&& a.Date >= startDate

&& a.Date <= endDate)

.GroupBy(a => a.StudentId)

.Select(g => new

{

StudentId = g.Key,

TotalDays = g.Count(),

PresentDays = g.Count(a => a.Status == AttendanceStatus.Present),

AbsentDays = g.Count(a => a.Status == AttendanceStatus.Absent),

LateDays = g.Count(a => a.Status == AttendanceStatus.Late)

})

.ToListAsync();

var totalSchoolDays = await CalculateSchoolDaysAsync(startDate, endDate);

return students.Select(s =>

{

var attendance = attendanceData.FirstOrDefault(a => a.StudentId == s.Id);

var presentDays = attendance?.PresentDays ?? 0;

var totalDays = Math.Max(attendance?.TotalDays ?? 0, totalSchoolDays);

return new AttendanceReportDto

{

StudentName = $"{s.FirstName} {s.LastName}",

StudentNumber = s.StudentNumber,

TotalDays = totalDays,

PresentDays = presentDays,

AbsentDays = attendance?.AbsentDays ?? 0,

LateDays = attendance?.LateDays ?? 0,

AttendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0

};

}).ToList();

}

public async Task<AttendanceReportDto> GetStudentAttendanceReportAsync(int studentId, DateTime startDate, DateTime endDate)

{

var student = await \_context.Students.FindAsync(studentId);

var attendances = await \_context.Attendances

.Where(a => a.StudentId == studentId

&& a.Date >= startDate

&& a.Date <= endDate)

.ToListAsync();

var totalDays = await CalculateSchoolDaysAsync(startDate, endDate);

var presentDays = attendances.Count(a => a.Status == AttendanceStatus.Present);

var absentDays = attendances.Count(a => a.Status == AttendanceStatus.Absent);

var lateDays = attendances.Count(a => a.Status == AttendanceStatus.Late);

return new AttendanceReportDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

TotalDays = Math.Max(attendances.Count, totalDays),

PresentDays = presentDays,

AbsentDays = absentDays,

LateDays = lateDays,

AttendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0

};

}

private async Task<int> CalculateSchoolDaysAsync(DateTime startDate, DateTime endDate)

{

// Calculate weekdays between dates (excluding weekends)

var days = 0;

for (var date = startDate; date <= endDate; date = date.AddDays(1))

{

if (date.DayOfWeek != DayOfWeek.Saturday && date.DayOfWeek != DayOfWeek.Sunday)

days++;

}

return days;

}

private async Task SendAbsenteeNotificationAsync(Attendance attendance)

{

var student = await \_context.Students

.Include(s => s.Parents)

.FirstOrDefaultAsync(s => s.Id == attendance.StudentId);

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "Student Absent",

Message = $"{student.FirstName} was marked absent on {attendance.Date:yyyy-MM-dd}",

Type = NotificationType.Attendance,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

}

}

public class TimetableService : ITimetableService

{

private readonly SchoolManagementContext \_context;

public TimetableService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<Timetable> GenerateTimetableAsync(int classId, int schoolYearId)

{

var classEntity = await \_context.Classes

.Include(c => c.ClassSubjects)

.ThenInclude(cs => cs.Subject)

.Include(c => c.ClassSubjects)

.ThenInclude(cs => cs.Teacher)

.FirstOrDefaultAsync(c => c.Id == classId);

var timetable = new Timetable

{

Name = $"{classEntity.Name} Timetable {DateTime.Now.Year}",

ClassId = classId,

SchoolYearId = schoolYearId,

IsActive = true,

CreatedAt = DateTime.UtcNow

};

\_context.Timetables.Add(timetable);

await \_context.SaveChangesAsync();

// Generate basic timetable structure

await GenerateBasicTimetableStructureAsync(timetable, classEntity.ClassSubjects.ToList());

return timetable;

}

private async Task GenerateBasicTimetableStructureAsync(Timetable timetable, List<ClassSubject> classSubjects)

{

var timeSlots = new[]

{

(new TimeSpan(8, 0, 0), new TimeSpan(8, 45, 0)),

(new TimeSpan(8, 45, 0), new TimeSpan(9, 30, 0)),

(new TimeSpan(9, 30, 0), new TimeSpan(9, 45, 0)), // Break

(new TimeSpan(9, 45, 0), new TimeSpan(10, 30, 0)),

(new TimeSpan(10, 30, 0), new TimeSpan(11, 15, 0)),

(new TimeSpan(11, 15, 0), new TimeSpan(12, 0, 0)),

(new TimeSpan(12, 0, 0), new TimeSpan(13, 0, 0)), // Lunch

(new TimeSpan(13, 0, 0), new TimeSpan(13, 45, 0)),

(new TimeSpan(13, 45, 0), new TimeSpan(14, 30, 0))

};

var workingDays = new[] { DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday, DayOfWeek.Thursday, DayOfWeek.Friday };

var subjectRotation = 0;

foreach (var day in workingDays)

{

for (int i = 0; i < timeSlots.Length; i++)

{

var (startTime, endTime) = timeSlots[i];

TimetableSlot slot;

if (i == 2) // Break time

{

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Type = SlotType.Break,

Room = "Playground"

};

}

else if (i == 6) // Lunch time

{

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Type = SlotType.Lunch,

Room = "Cafeteria"

};

}

else if (classSubjects.Any())

{

var classSubject = classSubjects[subjectRotation % classSubjects.Count];

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

ClassSubjectId = classSubject.Id,

Type = SlotType.Regular,

Room = $"Room {subjectRotation + 1}"

};

subjectRotation++;

}

else continue;

\_context.TimetableSlots.Add(slot);

}

}

await \_context.SaveChangesAsync();

}

public async Task<TimetableDto> GetClassTimetableAsync(int classId)

{

var timetable = await \_context.Timetables

.Where(t => t.ClassId == classId && t.IsActive)

.Include(t => t.Slots)

.ThenInclude(s => s.ClassSubject)

.ThenInclude(cs => cs.Subject)

.Include(t => t.Slots)

.ThenInclude(s => s.ClassSubject)

.ThenInclude(cs => cs.Teacher)

.Include(t => t.Class)

.FirstOrDefaultAsync();

if (timetable == null) return null;

var slots = timetable.Slots.Select(s => new TimetableSlotDto

{

DayOfWeek = s.DayOfWeek,

StartTime = s.StartTime.ToString(@"hh\:mm"),

EndTime = s.EndTime.ToString(@"hh\:mm"),

SubjectName = s.ClassSubject?.Subject?.Name ?? s.Type.ToString(),

TeacherName = s.ClassSubject?.Teacher != null

? $"{s.ClassSubject.Teacher.FirstName} {s.ClassSubject.Teacher.LastName}"

: "",

Room = s.Room,

Type = s.Type

}).ToList();

return new TimetableDto

{

Id = timetable.Id,

ClassName = timetable.Class.Name,

Slots = slots

};

}

public async Task<TimetableDto> GetStudentTimetableAsync(int studentId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetClassTimetableAsync(student.ClassId);

}

public async Task<TimetableSlot> UpdateTimetableSlotAsync(TimetableSlot slot)

{

\_context.TimetableSlots.Update(slot);

await \_context.SaveChangesAsync();

return slot;

}

public async Task<bool> ValidateTimetableRulesAsync(int timetableId)

{

var slots = await \_context.TimetableSlots

.Where(s => s.TimetableId == timetableId)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Teacher)

.ToListAsync();

// Check for teacher conflicts

var teacherConflicts = slots

.Where(s => s.ClassSubject?.Teacher != null)

.GroupBy(s => new { s.DayOfWeek, s.ClassSubject.TeacherId })

.Where(g => g.Any(s1 => g.Any(s2 => s1.Id != s2.Id

&& s1.StartTime < s2.EndTime

&& s2.StartTime < s1.EndTime)))

.Any();

// Check for room conflicts

var roomConflicts = slots

.Where(s => !string.IsNullOrEmpty(s.Room))

.GroupBy(s => new { s.DayOfWeek, s.Room })

.Where(g => g.Any(s1 => g.Any(s2 => s1.Id != s2.Id

&& s1.StartTime < s2.EndTime

&& s2.StartTime < s1.EndTime)))

.Any();

return !teacherConflicts && !roomConflicts;

}

public async Task<List<TimetableSlot>> GetTeacherScheduleAsync(int teacherId, DateTime date)

{

return await \_context.TimetableSlots

.Where(s => s.ClassSubject.TeacherId == teacherId)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Subject)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Class)

.ToListAsync();

}

}

public class AssignmentService : IAssignmentService

{

private readonly SchoolManagementContext \_context;

private readonly INotificationService \_notificationService;

private readonly IHandwritingRecognitionService \_handwritingService;

public AssignmentService(SchoolManagementContext context, INotificationService notificationService, IHandwritingRecognitionService handwritingService)

{

\_context = context;

\_notificationService = notificationService;

\_handwritingService = handwritingService;

}

public async Task<Assignment> CreateAssignmentAsync(Assignment assignment)

{

assignment.CreatedAt = DateTime.UtcNow;

\_context.Assignments.Add(assignment);

await \_context.SaveChangesAsync();

// Notify students/parents about new assignment

await SendAssignmentNotificationAsync(assignment);

return assignment;

}

public async Task<AssignmentSubmission> SubmitAssignmentAsync(AssignmentSubmission submission)

{

submission.SubmittedAt = DateTime.UtcNow;

submission.Status = submission.SubmittedAt <= await GetAssignmentDueDateAsync(submission.AssignmentId)

? SubmissionStatus.Submitted

: SubmissionStatus.Late;

\_context.AssignmentSubmissions.Add(submission);

await \_context.SaveChangesAsync();

// Process handwritten answers

foreach (var answer in submission.Answers.Where(a => !string.IsNullOrEmpty(a.HandwrittenImagePath)))

{

var recognizedText = await \_handwritingService.RecognizeHandwritingAsync(answer.HandwrittenImagePath, submission.StudentId);

answer.ProcessedText = recognizedText;

}

// Auto-grade if it's an online platform assignment

var assignment = await \_context.Assignments.FindAsync(submission.AssignmentId);

if (assignment.IsOnlinePlatformWork)

{

await AutoGradeAssignmentAsync(submission.Id);

}

return submission;

}

public async Task<AssignmentSubmission> AutoGradeAssignmentAsync(int submissionId)

{

var submission = await \_context.AssignmentSubmissions

.Include(s => s.Assignment)

.ThenInclude(a => a.Questions)

.ThenInclude(q => q.Options)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == submissionId);

decimal totalScore = 0;

decimal maxScore = submission.Assignment.Questions.Sum(q => q.Marks);

foreach (var answer in submission.Answers)

{

var question = submission.Assignment.Questions.First(q => q.Id == answer.QuestionId);

switch (question.Type)

{

case QuestionType.MultipleChoice:

var correctOption = question.Options.FirstOrDefault(o => o.IsCorrect);

if (correctOption != null && answer.Answer == correctOption.OptionLetter.ToString())

{

answer.IsCorrect = true;

answer.Score = question.Marks;

totalScore += question.Marks;

}

break;

case QuestionType.TrueFalse:

if (string.Equals(answer.Answer, question.CorrectAnswer, StringComparison.OrdinalIgnoreCase))

{

answer.IsCorrect = true;

answer.Score = question.Marks;

totalScore += question.Marks;

}

break;

case QuestionType.ShortAnswer:

// Simple string matching - could be enhanced with fuzzy matching

var similarity = CalculateStringSimilarity(answer.Answer, question.CorrectAnswer);

if (similarity > 0.8m)

{

answer.IsCorrect = true;

answer.Score = question.Marks \* similarity;

totalScore += answer.Score.Value;

}

break;

case QuestionType.Handwritten:

// Use processed handwritten text for comparison

var handwritingSimilarity = CalculateStringSimilarity(answer.ProcessedText, question.CorrectAnswer);

if (handwritingSimilarity > 0.7m) // Lower threshold for handwriting

{

answer.IsCorrect = true;

answer.Score = question.Marks \* handwritingSimilarity;

totalScore += answer.Score.Value;

}

break;

}

}

submission.Score = maxScore > 0 ? (totalScore / maxScore) \* 100 : 0;

submission.Status = SubmissionStatus.Graded;

submission.IsAutoGraded = true;

await \_context.SaveChangesAsync();

return submission;

}

public async Task<List<Assignment>> GetClassAssignmentsAsync(int classId)

{

return await \_context.Assignments

.Where(a => a.ClassId == classId)

.Include(a => a.Subject)

.Include(a => a.Teacher)

.Include(a => a.Questions)

.OrderByDescending(a => a.CreatedAt)

.ToListAsync();

}

public async Task<List<Assignment>> GetStudentAssignmentsAsync(int studentId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetClassAssignmentsAsync(student.ClassId);

}

public async Task<decimal> CalculateContinuousAssessmentMarkAsync(int studentId, int subjectId, int termId)

{

var submissions = await \_context.AssignmentSubmissions

.Where(s => s.StudentId == studentId

&& s.Assignment.SubjectId == subjectId

&& s.Status == SubmissionStatus.Graded)

.Include(s => s.Assignment)

.ToListAsync();

if (!submissions.Any()) return 0;

// Weight different assignment types

var weightedScores = submissions.Select(s => new

{

Score = s.Score ?? 0,

Weight = s.Assignment.Type switch

{

AssignmentType.Homework => 0.2m,

AssignmentType.Classwork => 0.3m,

AssignmentType.Assignment => 0.3m,

AssignmentType.Project => 0.4m,

AssignmentType.Quiz => 0.25m,

\_ => 0.25m

}

});

var totalWeightedScore = weightedScores.Sum(ws => ws.Score \* ws.Weight);

var totalWeight = weightedScores.Sum(ws => ws.Weight);

return totalWeight > 0 ? totalWeightedScore / totalWeight : 0;

}

private decimal CalculateStringSimilarity(string str1, string str2)

{

if (string.IsNullOrEmpty(str1) || string.IsNullOrEmpty(str2)) return 0;

// Simple Levenshtein distance-based similarity

var distance = LevenshteinDistance(str1.ToLower(), str2.ToLower());

var maxLength = Math.Max(str1.Length, str2.Length);

return maxLength > 0 ? 1m - (decimal)distance / maxLength : 0;

}

private int LevenshteinDistance(string s1, string s2)

{

if (s1.Length == 0) return s2.Length;

if (s2.Length == 0) return s1.Length;

int[,] d = new int[s1.Length + 1, s2.Length + 1];

for (int i = 0; i <= s1.Length; i++) d[i, 0] = i;

for (int j = 0; j <= s2.Length; j++) d[0, j] = j;

for (int i = 1; i <= s1.Length; i++)

{

for (int j = 1; j <= s2.Length; j++)

{

int cost = s1[i - 1] == s2[j - 1] ? 0 : 1;

d[i, j] = Math.Min(Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1), d[i - 1, j - 1] + cost);

}

}

return d[s1.Length, s2.Length];

}

private async Task<DateTime> GetAssignmentDueDateAsync(int assignmentId)

{

var assignment = await \_context.Assignments.FindAsync(assignmentId);

return assignment.DueDate;

}

private async Task SendAssignmentNotificationAsync(Assignment assignment)

{

var students = await \_context.Students

.Where(s => s.ClassId == assignment.ClassId)

.Include(s => s.Parents)

.ToListAsync();

foreach (var student in students)

{

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "New Assignment",

Message = $"New {assignment.Type} assigned: {assignment.Title}. Due: {assignment.DueDate:yyyy-MM-dd}",

Type = NotificationType.Assignment,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

}

}

}

}

public class HandwritingRecognitionService : IHandwritingRecognitionService

{

private readonly SchoolManagementContext \_context;

private readonly IConfiguration \_configuration;

private readonly HttpClient \_httpClient;

public HandwritingRecognitionService(SchoolManagementContext context, IConfiguration configuration, HttpClient httpClient)

{

\_context = context;

\_configuration = configuration;

\_httpClient = httpClient;

}

public async Task<string> RecognizeHandwritingAsync(string imagePath, int studentId)

{

// First try local model if available

var localResult = await ProcessHandwritingOfflineAsync(imagePath, studentId);

if (localResult) return await GetLocalRecognitionResultAsync(imagePath);

// Fallback to cloud-based recognition

return await ProcessHandwritingCloudAsync(imagePath, studentId);

}

public async Task<HandwritingSample> CreateHandwritingSampleAsync(HandwritingRecognitionDto dto)

{

var imageBytes = Convert.FromBase64String(dto.ImageBase64);

var imagePath = await SaveImageAsync(imageBytes, dto.StudentId);

var recognizedText = await RecognizeHandwritingAsync(imagePath, dto.StudentId);

var sample = new HandwritingSample

{

StudentId = dto.StudentId,

ImagePath = imagePath,

ExpectedText = dto.ExpectedText,

RecognizedText = recognizedText,

Type = dto.Type,

CreatedAt = DateTime.UtcNow,

IsTrainingData = true,

Confidence = CalculateConfidence(dto.ExpectedText, recognizedText)

};

\_context.HandwritingSamples.Add(sample);

await \_context.SaveChangesAsync();

return sample;

}

public async Task<bool> ValidateHandwritingSampleAsync(int sampleId, string correctedText, int teacherId)

{

var sample = await \_context.HandwritingSamples.FindAsync(sampleId);

if (sample == null) return false;

var validation = new HandwritingValidation

{

HandwritingSampleId = sampleId,

TeacherId = teacherId,

CorrectedText = correctedText,

ValidatedAt = DateTime.UtcNow,

Status = ValidationStatus.Approved

};

\_context.HandwritingValidations.Add(validation);

sample.IsValidated = true;

sample.RecognizedText = correctedText;

await \_context.SaveChangesAsync();

// Trigger model retraining if enough samples

await CheckAndTriggerModelRetrainingAsync(sample.StudentId);

return true;

}

public async Task TrainPersonalizedModelAsync(int studentId)

{

var trainingSamples = await \_context.HandwritingSamples

.Where(s => s.StudentId == studentId && s.IsValidated)

.ToListAsync();

if (trainingSamples.Count < 50) // Minimum samples for training

{

throw new InvalidOperationException("Insufficient training samples. Minimum 50 validated samples required.");

}

// Prepare training data

var trainingData = trainingSamples.Select(s => new

{

ImagePath = s.ImagePath,

GroundTruth = s.RecognizedText, // Use validated text

Type = s.Type

}).ToList();

// Call ML training service (this would be implemented with ML.NET or similar)

var modelPath = await TrainModelAsync(studentId, trainingData);

var existingModel = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId);

if (existingModel != null)

{

existingModel.ModelPath = modelPath;

existingModel.LastTrainingDate = DateTime.UtcNow;

existingModel.SampleCount = trainingSamples.Count;

existingModel.Accuracy = await CalculateModelAccuracyAsync(modelPath, studentId);

}

else

{

var newModel = new PersonalizedModel

{

StudentId = studentId,

ModelPath = modelPath,

LastTrainingDate = DateTime.UtcNow,

SampleCount = trainingSamples.Count,

Accuracy = await CalculateModelAccuracyAsync(modelPath, studentId),

IsDeployedLocally = false,

IsDeployedOnCloud = true

};

\_context.PersonalizedModels.Add(newModel);

}

await \_context.SaveChangesAsync();

}

public async Task<PersonalizedModel> DeployModelLocallyAsync(int studentId)

{

var model = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId);

if (model == null) return null;

// Deploy model to local device (implementation depends on your mobile/desktop app architecture)

var localPath = await DeployToLocalDeviceAsync(model.CloudModelPath, studentId);

model.ModelPath = localPath;

model.IsDeployedLocally = true;

await \_context.SaveChangesAsync();

return model;

}

public async Task<string> GenerateTrainingContentAsync(HandwritingType type, string difficulty = "beginner")

{

return type switch

{

HandwritingType.Alphabet => GenerateAlphabetContent(difficulty),

HandwritingType.Number => GenerateNumberContent(difficulty),

HandwritingType.Word => GenerateWordContent(difficulty),

HandwritingType.Sentence => GenerateSentenceContent(difficulty),

\_ => "Practice writing: Hello World"

};

}

public async Task<bool> ProcessHandwritingOfflineAsync(string imagePath, int studentId)

{

var model = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId && m.IsDeployedLocally);

if (model == null) return false;

// Process using local model (implementation depends on your ML framework)

try

{

await ProcessWithLocalModelAsync(imagePath, model.ModelPath);

return true;

}

catch

{

return false;

}

}

private async Task<string> ProcessHandwritingCloudAsync(string imagePath, int studentId)

{

// Call cloud-based handwriting recognition API

var cloudApiUrl = \_configuration["HandwritingRecognition:CloudApiUrl"];

using var content = new MultipartFormDataContent();

var imageBytes = await File.ReadAllBytesAsync(imagePath);

content.Add(new ByteArrayContent(imageBytes), "image", "handwriting.jpg");

content.Add(new StringContent(studentId.ToString()), "studentId");

var response = await \_httpClient.PostAsync(cloudApiUrl, content);

var result = await response.Content.ReadAsStringAsync();

return result; // Assume API returns recognized text

}

private decimal CalculateConfidence(string expected, string recognized)

{

if (string.IsNullOrEmpty(expected) || string.IsNullOrEmpty(recognized)) return 0;

var similarity = 1m - (decimal)LevenshteinDistance(expected.ToLower(), recognized.ToLower()) / Math.Max(expected.Length, recognized.Length);

return Math.Max(0, Math.Min(1, similarity));

}

private int LevenshteinDistance(string s1, string s2)

{

// Same implementation as in AssignmentService

if (s1.Length == 0) return s2.Length;

if (s2.Length == 0) return s1.Length;

int[,] d = new int[s1.Length + 1, s2.Length + 1];

for (int i = 0; i <= s1.Length; i++) d[i, 0] = i;

for (int j = 0; j <= s2.Length; j++) d[0, j] = j;

for (int i = 1; i <= s1.Length; i++)

{

for (int j = 1; j <= s2.Length; j++)

{

int cost = s1[i - 1] == s2[j - 1] ? 0 : 1;

d[i, j] = Math.Min(Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1), d[i - 1, j - 1] + cost);

}

}

return d[s1.Length, s2.Length];

}

private string GenerateAlphabetContent(string difficulty)

{

return difficulty switch

{

"beginner" => "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z",

"intermediate" => "a b c d e f g h i j k l m n o p q r s t u v w x y z",

"advanced" => "Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz",

\_ => "A B C D E"

};

}

private string GenerateNumberContent(string difficulty)

{

return difficulty switch

{

"beginner" => "1 2 3 4 5 6 7 8 9 0",

"intermediate" => "12 34 56 78 90 123 456 789",

"advanced" => "1,234 5,678 9,012 3,456 7,890",

\_ => "1 2 3 4 5"

};

}

private string GenerateWordContent(string difficulty)

{

return difficulty switch

{

"beginner" => "cat dog sun fun run",

"intermediate" => "school book pencil teacher student",

"advanced" => "education mathematics science literature",

\_ => "cat dog"

};

}

private string GenerateSentenceContent(string difficulty)

{

return difficulty switch

{

"beginner" => "The cat sat on the mat.",

"intermediate" => "I love going to school every day.",

"advanced" => "Education is the most powerful weapon which you can use to change the world.",

\_ => "Hello world."

};

}

private async Task<string> SaveImageAsync(byte[] imageBytes, int studentId)

{

var uploadsPath = Path.Combine("uploads", "handwriting", studentId.ToString());

Directory.CreateDirectory(uploadsPath);

var fileName = $"{Guid.NewGuid()}.jpg";

var filePath = Path.Combine(uploadsPath, fileName);

await File.WriteAllBytesAsync(filePath, imageBytes);

return filePath;

}

private async Task CheckAndTriggerModelRetrainingAsync(int studentId)

{

var validatedSamples = await \_context.HandwritingSamples

.CountAsync(s => s.StudentId == studentId && s.IsValidated);

if (validatedSamples >= 50 && validatedSamples % 25 == 0) // Retrain every 25 new samples

{

await TrainPersonalizedModelAsync(studentId);

}

}

private async Task<string> TrainModelAsync(int studentId, object trainingData)

{

// Implementation would use ML.NET or similar framework

var modelPath = $"models/student\_{studentId}\_{DateTime.UtcNow:yyyyMMdd}.model";

// Training logic here...

return modelPath;

}

private async Task<decimal> CalculateModelAccuracyAsync(string modelPath, int studentId)

{

// Test model accuracy on validation set

return 0.85m; // Placeholder

}

private async Task<string> DeployToLocalDeviceAsync(string cloudPath, int studentId)

{

// Deploy to local device storage

return $"local/models/student\_{studentId}.model";

}

private async Task<string> GetLocalRecognitionResultAsync(string imagePath)

{

// Get result from local processing

return "Sample recognized text";

}

private async Task ProcessWithLocalModelAsync(string imagePath, string modelPath)

{

// Process image with local model

await Task.Delay(100); // Placeholder

}

}

// =================== AI ASSESSMENT & VERIFICATION SYSTEM ===================

public interface IAIAssessmentService

{

Task<AIAssessmentResult> AssessSubmissionAsync(int submissionId);

Task<List<AIAssessmentResult>> BatchAssessSubmissionsAsync(List<int> submissionIds);

Task<AIAssessmentResult> ProcessHandwrittenAnswerAsync(int submissionAnswerId);

Task<bool> FlagForTeacherReviewAsync(int assessmentResultId, string reason);

Task<List<AIAssessmentResult>> GetPendingReviewsAsync(int teacherId);

Task<AIAssessmentResult> TeacherVerifyAssessmentAsync(int assessmentResultId, int teacherId, TeacherVerificationDto verification);

}

public class AIAssessmentResult

{

public int Id { get; set; }

public int? SubmissionId { get; set; }

public AssignmentSubmission Submission { get; set; }

public int? SubmissionAnswerId { get; set; }

public SubmissionAnswer SubmissionAnswer { get; set; }

public decimal AIScore { get; set; }

public decimal ConfidenceLevel { get; set; }

public string AIFeedback { get; set; }

public AIAssessmentStatus Status { get; set; }

public bool RequiresTeacherReview { get; set; }

public string ReviewReason { get; set; }

public DateTime ProcessedAt { get; set; }

// Teacher verification

public int? VerifiedByTeacherId { get; set; }

public Teacher VerifiedByTeacher { get; set; }

public decimal? TeacherScore { get; set; }

public string TeacherFeedback { get; set; }

public DateTime? VerifiedAt { get; set; }

public VerificationStatus? VerificationStatus { get; set; }

// Detailed AI analysis

public string HandwritingRecognitionText { get; set; }

public decimal HandwritingConfidence { get; set; }

public List<AIScoreBreakdown> ScoreBreakdowns { get; set; } = new();

}

public enum AIAssessmentStatus

{

Processing,

Completed,

Failed,

PendingReview,

Verified,

Disputed

}

public enum VerificationStatus

{

Approved,

Modified,

Rejected,

NeedsReprocessing

}

public class AIScoreBreakdown

{

public int Id { get; set; }

public int AIAssessmentResultId { get; set; }

public AIAssessmentResult AIAssessmentResult { get; set; }

public string Criterion { get; set; } // "Accuracy", "Completeness", "Clarity", "Grammar"

public decimal Score { get; set; }

public decimal MaxScore { get; set; }

public string Explanation { get; set; }

}

public class TeacherVerificationDto

{

public decimal? OverrideScore { get; set; }

public string TeacherFeedback { get; set; }

public VerificationStatus VerificationStatus { get; set; }

public List<CriterionVerification> CriterionVerifications { get; set; } = new();

}

public class CriterionVerification

{

public string Criterion { get; set; }

public decimal TeacherScore { get; set; }

public string TeacherComment { get; set; }

}

// Add to DbContext

public class SchoolManagementContext : DbContext

{

// ... existing DbSets ...

public DbSet<AIAssessmentResult> AIAssessmentResults { get; set; }

public DbSet<AIScoreBreakdown> AIScoreBreakdowns { get; set; }

// ... existing configuration methods ...

private void ConfigureAIAssessmentSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.AIScore)

.HasPrecision(5, 2);

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.ConfidenceLevel)

.HasPrecision(5, 4);

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.HandwritingConfidence)

.HasPrecision(5, 4);

modelBuilder.Entity<AIScoreBreakdown>()

.Property(sb => sb.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<AIScoreBreakdown>()

.Property(sb => sb.MaxScore)

.HasPrecision(5, 2);

}

}

public class AIAssessmentService : IAIAssessmentService

{

private readonly SchoolManagementContext \_context;

private readonly IHandwritingRecognitionService \_handwritingService;

private readonly INotificationService \_notificationService;

private readonly HttpClient \_httpClient;

private readonly IConfiguration \_configuration;

public AIAssessmentService(

SchoolManagementContext context,

IHandwritingRecognitionService handwritingService,

INotificationService notificationService,

HttpClient httpClient,

IConfiguration configuration)

{

\_context = context;

\_handwritingService = handwritingService;

\_notificationService = notificationService;

\_httpClient = httpClient;

\_configuration = configuration;

}

public async Task<AIAssessmentResult> AssessSubmissionAsync(int submissionId)

{

var submission = await \_context.AssignmentSubmissions

.Include(s => s.Assignment)

.ThenInclude(a => a.Questions)

.ThenInclude(q => q.Options)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == submissionId);

if (submission == null) return null;

var assessmentResult = new AIAssessmentResult

{

SubmissionId = submissionId,

Status = AIAssessmentStatus.Processing,

ProcessedAt = DateTime.UtcNow

};

\_context.AIAssessmentResults.Add(assessmentResult);

await \_context.SaveChangesAsync();

try

{

decimal totalScore = 0;

decimal maxPossibleScore = submission.Assignment.Questions.Sum(q => q.Marks);

var allBreakdowns = new List<AIScoreBreakdown>();

foreach (var answer in submission.Answers)

{

var question = submission.Assignment.Questions.First(q => q.Id == answer.QuestionId);

var answerAssessment = await AssessIndividualAnswerAsync(answer, question);

totalScore += answerAssessment.Score;

allBreakdowns.AddRange(answerAssessment.Breakdowns);

// Update the answer with AI results

answer.Score = answerAssessment.Score;

answer.IsCorrect = answerAssessment.Score >= (question.Marks \* 0.7m); // 70% threshold

answer.ProcessedText = answerAssessment.ProcessedText;

answer.Confidence = answerAssessment.Confidence;

}

// Calculate final score and confidence

assessmentResult.AIScore = maxPossibleScore > 0 ? (totalScore / maxPossibleScore) \* 100 : 0;

assessmentResult.ConfidenceLevel = allBreakdowns.Any() ? allBreakdowns.Average(b => b.Score / b.MaxScore) : 0;

assessmentResult.AIFeedback = GenerateOverallFeedback(allBreakdowns, assessmentResult.AIScore);

assessmentResult.Status = AIAssessmentStatus.Completed;

assessmentResult.ScoreBreakdowns = allBreakdowns;

// Determine if teacher review is needed

var needsReview = DetermineIfTeacherReviewNeeded(assessmentResult, allBreakdowns);

if (needsReview.needed)

{

await FlagForTeacherReviewAsync(assessmentResult.Id, needsReview.reason);

}

// Update submission

submission.Score = assessmentResult.AIScore;

submission.Status = needsReview.needed ? SubmissionStatus.Submitted : SubmissionStatus.Graded;

submission.IsAutoGraded = true;

await \_context.SaveChangesAsync();

// Notify if teacher review is needed

if (needsReview.needed)

{

await NotifyTeacherForReviewAsync(submission.Assignment.TeacherId, assessmentResult.Id);

}

return assessmentResult;

}

catch (Exception ex)

{

assessmentResult.Status = AIAssessmentStatus.Failed;

assessmentResult.AIFeedback = $"Assessment failed: {ex.Message}";

await \_context.SaveChangesAsync();

return assessmentResult;

}

}

public async Task<List<AIAssessmentResult>> BatchAssessSubmissionsAsync(List<int> submissionIds)

{

var results = new List<AIAssessmentResult>();

// Process in batches to avoid overwhelming the system

const int batchSize = 10;

for (int i = 0; i < submissionIds.Count; i += batchSize)

{

var batch = submissionIds.Skip(i).Take(batchSize);

var batchTasks = batch.Select(AssessSubmissionAsync);

var batchResults = await Task.WhenAll(batchTasks);

results.AddRange(batchResults.Where(r => r != null));

}

return results;

}

public async Task<AIAssessmentResult> ProcessHandwrittenAnswerAsync(int submissionAnswerId)

{

var answer = await \_context.SubmissionAnswers

.Include(a => a.Question)

.Include(a => a.Submission)

.ThenInclude(s => s.Student)

.FirstOrDefaultAsync(a => a.Id == submissionAnswerId);

if (answer == null || string.IsNullOrEmpty(answer.HandwrittenImagePath))

return null;

var assessmentResult = new AIAssessmentResult

{

SubmissionAnswerId = submissionAnswerId,

Status = AIAssessmentStatus.Processing,

ProcessedAt = DateTime.UtcNow

};

\_context.AIAssessmentResults.Add(assessmentResult);

await \_context.SaveChangesAsync();

try

{

// Step 1: Handwriting Recognition

var recognizedText = await \_handwritingService.RecognizeHandwritingAsync(

answer.HandwrittenImagePath,

answer.Submission.StudentId);

assessmentResult.HandwritingRecognitionText = recognizedText;

// Calculate handwriting confidence

var handwritingConfidence = await CalculateHandwritingConfidenceAsync(

answer.HandwrittenImagePath, recognizedText);

assessmentResult.HandwritingConfidence = handwritingConfidence;

// Step 2: Content Assessment

var contentAssessment = await AssessAnswerContentAsync(

recognizedText,

answer.Question.CorrectAnswer,

answer.Question.Type,

answer.Question.Marks);

assessmentResult.AIScore = contentAssessment.Score;

assessmentResult.ConfidenceLevel = Math.Min(handwritingConfidence, contentAssessment.Confidence);

assessmentResult.AIFeedback = contentAssessment.Feedback;

assessmentResult.ScoreBreakdowns = contentAssessment.Breakdowns;

// Update the original answer

answer.ProcessedText = recognizedText;

answer.Score = contentAssessment.Score;

answer.Confidence = assessmentResult.ConfidenceLevel;

answer.IsCorrect = contentAssessment.Score >= (answer.Question.Marks \* 0.7m);

// Determine if manual review is needed

var needsReview = handwritingConfidence < 0.8m || contentAssessment.Confidence < 0.8m;

if (needsReview)

{

await FlagForTeacherReviewAsync(assessmentResult.Id,

$"Low confidence: Handwriting={handwritingConfidence:P}, Content={contentAssessment.Confidence:P}");

}

assessmentResult.Status = AIAssessmentStatus.Completed;

await \_context.SaveChangesAsync();

return assessmentResult;

}

catch (Exception ex)

{

assessmentResult.Status = AIAssessmentStatus.Failed;

assessmentResult.AIFeedback = $"Processing failed: {ex.Message}";

await \_context.SaveChangesAsync();

return assessmentResult;

}

}

public async Task<bool> FlagForTeacherReviewAsync(int assessmentResultId, string reason)

{

var assessmentResult = await \_context.AIAssessmentResults.FindAsync(assessmentResultId);

if (assessmentResult == null) return false;

assessmentResult.RequiresTeacherReview = true;

assessmentResult.ReviewReason = reason;

assessmentResult.Status = AIAssessmentStatus.PendingReview;

await \_context.SaveChangesAsync();

return true;

}

public async Task<List<AIAssessmentResult>> GetPendingReviewsAsync(int teacherId)

{

return await \_context.AIAssessmentResults

.Where(ar => ar.RequiresTeacherReview

&& ar.Status == AIAssessmentStatus.PendingReview

&& (ar.Submission.Assignment.TeacherId == teacherId ||

ar.SubmissionAnswer.Submission.Assignment.TeacherId == teacherId))

.Include(ar => ar.Submission)

.ThenInclude(s => s.Student)

.Include(ar => ar.Submission)

.ThenInclude(s => s.Assignment)

.Include(ar => ar.SubmissionAnswer)

.ThenInclude(sa => sa.Question)

.Include(ar => ar.ScoreBreakdowns)

.OrderByDescending(ar => ar.ProcessedAt)

.ToListAsync();

}

public async Task<AIAssessmentResult> TeacherVerifyAssessmentAsync(int assessmentResultId, int teacherId, TeacherVerificationDto verification)

{

var assessmentResult = await \_context.AIAssessmentResults

.Include(ar => ar.ScoreBreakdowns)

.Include(ar => ar.Submission)

.ThenInclude(s => s.Assignment)

.FirstOrDefaultAsync(ar => ar.Id == assessmentResultId);

if (assessmentResult == null) return null;

// Record teacher verification

assessmentResult.VerifiedByTeacherId = teacherId;

assessmentResult.TeacherScore = verification.OverrideScore ?? assessmentResult.AIScore;

assessmentResult.TeacherFeedback = verification.TeacherFeedback;

assessmentResult.VerifiedAt = DateTime.UtcNow;

assessmentResult.VerificationStatus = verification.VerificationStatus;

assessmentResult.Status = AIAssessmentStatus.Verified;

assessmentResult.RequiresTeacherReview = false;

// Update criterion scores if provided

foreach (var criterionVerification in verification.CriterionVerifications)

{

var breakdown = assessmentResult.ScoreBreakdowns

.FirstOrDefault(b => b.Criterion == criterionVerification.Criterion);

if (breakdown != null)

{

breakdown.Score = criterionVerification.TeacherScore;

breakdown.Explanation = criterionVerification.TeacherComment;

}

}

// Update the associated submission/answer

if (assessmentResult.SubmissionId.HasValue)

{

var submission = assessmentResult.Submission;

submission.Score = assessmentResult.TeacherScore;

submission.Status = SubmissionStatus.Graded;

submission.Feedback = verification.TeacherFeedback;

}

else if (assessmentResult.SubmissionAnswerId.HasValue)

{

var answer = await \_context.SubmissionAnswers.FindAsync(assessmentResult.SubmissionAnswerId);

if (answer != null)

{

answer.Score = assessmentResult.TeacherScore;

answer.IsCorrect = assessmentResult.TeacherScore >= (answer.Question.Marks \* 0.7m);

}

}

await \_context.SaveChangesAsync();

// Learn from teacher corrections for future AI improvements

await RecordTeacherCorrectionForLearningAsync(assessmentResult);

return assessmentResult;

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessIndividualAnswerAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

switch (question.Type)

{

case QuestionType.MultipleChoice:

return await AssessMultipleChoiceAsync(answer, question);

case QuestionType.TrueFalse:

return await AssessTrueFalseAsync(answer, question);

case QuestionType.ShortAnswer:

return await AssessShortAnswerAsync(answer, question);

case QuestionType.Essay:

return await AssessEssayAsync(answer, question);

case QuestionType.Handwritten:

if (!string.IsNullOrEmpty(answer.HandwrittenImagePath))

{

var handwrittenResult = await ProcessHandwrittenAnswerAsync(answer.Id);

return (handwrittenResult?.AIScore ?? 0,

handwrittenResult?.ConfidenceLevel ?? 0,

handwrittenResult?.AIFeedback ?? "Processing failed",

handwrittenResult?.ScoreBreakdowns ?? new List<AIScoreBreakdown>());

}

return await AssessShortAnswerAsync(answer, question);

default:

return (0, 0, "Unknown question type", breakdowns);

}

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessMultipleChoiceAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var correctOption = question.Options.FirstOrDefault(o => o.IsCorrect);

var isCorrect = correctOption != null &&

string.Equals(answer.Answer, correctOption.OptionLetter.ToString(), StringComparison.OrdinalIgnoreCase);

var score = isCorrect ? question.Marks : 0;

var feedback = isCorrect ? "Correct answer" : $"Incorrect. The correct answer is {correctOption?.OptionLetter}";

var breakdown = new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = score,

MaxScore = question.Marks,

Explanation = feedback

};

return (score, 1.0m, feedback, new List<AIScoreBreakdown> { breakdown });

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessTrueFalseAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var isCorrect = string.Equals(answer.Answer, question.CorrectAnswer, StringComparison.OrdinalIgnoreCase);

var score = isCorrect ? question.Marks : 0;

var feedback = isCorrect ? "Correct answer" : $"Incorrect. The correct answer is {question.CorrectAnswer}";

var breakdown = new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = score,

MaxScore = question.Marks,

Explanation = feedback

};

return (score, 1.0m, feedback, new List<AIScoreBreakdown> { breakdown });

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessShortAnswerAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

// Assess accuracy

var accuracy = CalculateTextSimilarity(answer.Answer, question.CorrectAnswer);

var accuracyScore = question.Marks \* accuracy;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = accuracyScore,

MaxScore = question.Marks,

Explanation = $"Answer similarity to expected response: {accuracy:P}"

});

var totalScore = accuracyScore;

var confidence = accuracy > 0.6m ? 0.9m : 0.7m; // Lower confidence for low similarity

var feedback = accuracy switch

{

>= 0.9m => "Excellent answer, very close to expected response",

>= 0.7m => "Good answer, mostly correct",

>= 0.5m => "Partially correct, but missing some key points",

\_ => "Answer needs improvement, significantly different from expected response"

};

return (totalScore, confidence, feedback, breakdowns);

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessEssayAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

// This would ideally use advanced NLP/AI services like OpenAI GPT or Azure Cognitive Services

// For now, implementing basic assessment criteria

// Content relevance (40% of marks)

var contentScore = await AssessContentRelevanceAsync(answer.Answer, question.CorrectAnswer);

var contentMarks = question.Marks \* 0.4m \* contentScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Content Relevance",

Score = contentMarks,

MaxScore = question.Marks \* 0.4m,

Explanation = $"Content relevance score: {contentScore:P}"

});

// Grammar and language (30% of marks)

var grammarScore = await AssessGrammarAsync(answer.Answer);

var grammarMarks = question.Marks \* 0.3m \* grammarScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Grammar & Language",

Score = grammarMarks,

MaxScore = question.Marks \* 0.3m,

Explanation = $"Grammar and language quality: {grammarScore:P}"

});

// Structure and organization (30% of marks)

var structureScore = await AssessStructureAsync(answer.Answer);

var structureMarks = question.Marks \* 0.3m \* structureScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Structure & Organization",

Score = structureMarks,

MaxScore = question.Marks \* 0.3m,

Explanation = $"Structure and organization: {structureScore:P}"

});

var totalScore = contentMarks + grammarMarks + structureMarks;

var averageScore = (contentScore + grammarScore + structureScore) / 3;

// Lower confidence for essays as they're more subjective

var confidence = averageScore > 0.7m ? 0.75m : 0.6m;

var feedback = GenerateEssayFeedback(contentScore, grammarScore, structureScore);

return (totalScore, confidence, feedback, breakdowns);

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessAnswerContentAsync(

string recognizedText, string correctAnswer, QuestionType questionType, decimal maxMarks)

{

// Use the appropriate assessment method based on question type

var dummyAnswer = new SubmissionAnswer { Answer = recognizedText };

var dummyQuestion = new AssignmentQuestion

{

CorrectAnswer = correctAnswer,

Type = questionType,

Marks = maxMarks

};

return questionType switch

{

QuestionType.ShortAnswer => await AssessShortAnswerAsync(dummyAnswer, dummyQuestion),

QuestionType.Essay => await AssessEssayAsync(dummyAnswer, dummyQuestion),

\_ => await AssessShortAnswerAsync(dummyAnswer, dummyQuestion)

};

}

private decimal CalculateTextSimilarity(string text1, string text2)

{

if (string.IsNullOrEmpty(text1) || string.IsNullOrEmpty(text2)) return 0;

// Normalize texts

text1 = text1.ToLower().Trim();

text2 = text2.ToLower().Trim();

// Simple word-based similarity

var words1 = text1.Split(' ', StringSplitOptions.RemoveEmptyEntries);

var words2 = text2.Split(' ', StringSplitOptions.RemoveEmptyEntries);

var commonWords = words1.Intersect(words2).Count();

var totalWords = Math.Max(words1.Length, words2.Length);

return totalWords > 0 ? (decimal)commonWords / totalWords : 0;

}

private async Task<decimal> CalculateHandwritingConfidenceAsync(string imagePath, string recognizedText)

{

// This would use image quality metrics and OCR confidence scores

// For now, return a simulated confidence based on text length and clarity

if (string.IsNullOrEmpty(recognizedText)) return 0;

// Simulate confidence calculation

var baseConfidence = 0.8m;

var lengthFactor = Math.Min(recognizedText.Length / 50m, 1m); // Longer text = higher confidence

var clarityFactor = recognizedText.Count(char.IsLetter) / (decimal)recognizedText.Length;

return Math.Min(baseConfidence \* lengthFactor \* clarityFactor, 1m);

}

private async Task<decimal> AssessContentRelevanceAsync(string studentAnswer, string expectedAnswer)

{

return CalculateTextSimilarity(studentAnswer, expectedAnswer);

}

private async Task<decimal> AssessGrammarAsync(string text)

{

if (string.IsNullOrEmpty(text)) return 0;

// Basic grammar assessment

var sentences = text.Split('.', '!', '?').Where(s => !string.IsNullOrWhiteSpace(s)).ToArray();

if (sentences.Length == 0) return 0;

var grammarScore = 0.8m; // Base score

// Simple checks

var hasCapitalizedSentences = sentences.Count(s => char.IsUpper(s.Trim().FirstOrDefault())) / (decimal)sentences.Length;

var hasProperPunctuation = (text.Count(c => ".!?".Contains(c)) >= sentences.Length) ? 1m : 0.7m;

return (grammarScore + hasCapitalizedSentences + hasProperPunctuation) / 3;

}

private async Task<decimal> AssessStructureAsync(string text)

{

if (string.IsNullOrEmpty(text)) return 0;

var sentences = text.Split('.', '!', '?').Where(s => !string.IsNullOrWhiteSpace(s)).Count();

var paragraphs = text.Split('\n').Where(p => !string.IsNullOrWhiteSpace(p)).Count();

// Basic structure scoring

var structureScore = 0.7m; // Base score

if (sentences >= 3) structureScore += 0.2m; // Has multiple sentences

if (paragraphs >= 2) structureScore += 0.1m; // Has multiple paragraphs

return Math.Min(structureScore, 1m);

}

private string GenerateEssayFeedback(decimal contentScore, decimal grammarScore, decimal structureScore)

{

var feedback = new List<string>();

if (contentScore >= 0.8m) feedback.Add("Excellent content relevance and understanding");

else if (contentScore >= 0.6m) feedback.Add("Good content but could be more detailed");

else feedback.Add("Content needs improvement - ensure you address all key points");

if (grammarScore >= 0.8m) feedback.Add("Good grammar and language use");

else feedback.Add("Pay attention to grammar, spelling, and sentence structure");

if (structureScore >= 0.8m) feedback.Add("Well-organized response");

else feedback.Add("Work on organizing your thoughts into clear paragraphs");

return string.Join(". ", feedback);

}

private string GenerateOverallFeedback(List<AIScoreBreakdown> breakdowns, decimal overallScore)

{

if (overallScore >= 80) return "Excellent work! You have demonstrated strong understanding.";

if (overallScore >= 70) return "Good work! You're on the right track with room for improvement.";

if (overallScore >= 60) return "Fair performance. Focus on improving weaker areas.";

if (overallScore >= 50) return "Below average performance. Consider reviewing the material.";

return "Needs significant improvement. Please seek additional help.";

}

private (bool needed, string reason) DetermineIfTeacherReviewNeeded(AIAssessmentResult assessment, List<AIScoreBreakdown> breakdowns)

{

// Flag for review if confidence is low

if (assessment.ConfidenceLevel < 0.7m)

return (true, $"Low AI confidence: {assessment.Conf

//////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Flag for review if confidence is low

if (assessment.ConfidenceLevel < 0.7m)

return (true, $"Low AI confidence: {assessment.ConfidenceLevel:P}");

// Flag for review if handwriting confidence is low

if (assessment.HandwritingConfidence > 0 && assessment.HandwritingConfidence < 0.75m)

return (true, $"Low handwriting recognition confidence: {assessment.HandwritingConfidence:P}");

// Flag for review if score is borderline (around pass/fail boundary)

if (assessment.AIScore >= 45 && assessment.AIScore <= 55)

return (true, "Borderline score requires teacher verification");

// Flag for review if there are significant discrepancies in criterion scores

var criterionScores = breakdowns.Select(b => b.Score / b.MaxScore).ToList();

if (criterionScores.Any() && criterionScores.Max() - criterionScores.Min() > 0.4m)

return (true, "Significant variation in criterion scores");

return (false, "");

}

private async Task NotifyTeacherForReviewAsync(int teacherId, int assessmentResultId)

{

var notification = new Notification

{

Title = "Assignment Review Required",

Message = $"An AI assessment requires your review. Assessment ID: {assessmentResultId}",

Type = NotificationType.Assignment,

TeacherId = teacherId,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

private async Task RecordTeacherCorrectionForLearningAsync(AIAssessmentResult assessment)

{

// This would be used to improve the AI model over time

// Record teacher corrections for machine learning improvement

var correctionData = new

{

AIScore = assessment.AIScore,

TeacherScore = assessment.TeacherScore,

OriginalText = assessment.HandwritingRecognitionText,

QuestionType = assessment.SubmissionAnswer?.Question?.Type,

CorrectionReason = assessment.VerificationStatus,

Timestamp = DateTime.UtcNow

};

// In a real implementation, this would be sent to an ML pipeline

// for model retraining and improvement

}

}

// =================== ENHANCED REPORT CARD SERVICE ===================

public interface IReportCardService

{

Task<DetailedReportCardDto> GenerateDetailedReportCardAsync(int studentId, int termId);

Task<byte[]> GenerateReportCardPdfAsync(int studentId, int termId);

Task<ClassReportSummaryDto> GenerateClassReportSummaryAsync(int classId, int termId);

Task<byte[]> GenerateClassReportPdfAsync(int classId, int termId);

}

public class DetailedReportCardDto

{

public StudentInfoDto Student { get; set; }

public SchoolInfoDto School { get; set; }

public TermInfoDto Term { get; set; }

public List<SubjectPerformanceDto> SubjectPerformances { get; set; } = new();

public OverallPerformanceDto OverallPerformance { get; set; }

public AttendanceSummaryDto Attendance { get; set; }

public string TeacherComments { get; set; }

public string HeadTeacherComments { get; set; }

public DateTime GeneratedAt { get; set; }

public GradingSchemeInfoDto GradingScheme { get; set; }

}

public class StudentInfoDto

{

public string FullName { get; set; }

public string StudentNumber { get; set; }

public string ClassName { get; set; }

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; }

}

public class SchoolInfoDto

{

public string Name { get; set; }

public string Address { get; set; }

public string Phone { get; set; }

public string Email { get; set; }

public string Logo { get; set; }

}

public class TermInfoDto

{

public string Name { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public string SchoolYear { get; set; }

}

public class SubjectPerformanceDto

{

public string SubjectName { get; set; }

public string SubjectCode { get; set; }

public string TeacherName { get; set; }

// Detailed marks breakdown

public List<AssessmentMarkDto> AssessmentMarks { get; set; } = new();

public decimal ContinuousAssessmentMark { get; set; }

public decimal ExamMark { get; set; }

public decimal TotalMark { get; set; }

public decimal Percentage { get; set; }

// Grading information

public string Grade { get; set; }

public int? Unit { get; set; }

public decimal GradePoint { get; set; }

// Performance indicators

public string Comments { get; set; }

public int SubjectPosition { get; set; }

public int TotalStudentsInSubject { get; set; }

public decimal ClassAverage { get; set; }

public string PerformanceTrend { get; set; } // "Improving", "Declining", "Stable"

}

public class AssessmentMarkDto

{

public string AssessmentType { get; set; } // "Test 1", "Assignment", "Project"

public decimal Mark { get; set; }

public decimal MaxMark { get; set; }

public decimal Percentage { get; set; }

public DateTime DateAssessed { get; set; }

}

public class OverallPerformanceDto

{

public decimal TotalMarks { get; set; }

public decimal MaxPossibleMarks { get; set; }

public decimal OverallPercentage { get; set; }

public string OverallGrade { get; set; }

public int? OverallUnit { get; set; }

public decimal GPA { get; set; }

public int ClassPosition { get; set; }

public int TotalStudentsInClass { get; set; }

public decimal ClassAverage { get; set; }

public string PerformanceSummary { get; set; }

}

public class AttendanceSummaryDto

{

public int TotalSchoolDays { get; set; }

public int DaysPresent { get; set; }

public int DaysAbsent { get; set; }

public int DaysLate { get; set; }

public decimal AttendancePercentage { get; set; }

public string AttendanceGrade { get; set; }

}

public class GradingSchemeInfoDto

{

public string Name { get; set; }

public string GradingBody { get; set; }

public List<GradeScaleInfoDto> GradeScales { get; set; } = new();

}

public class GradeScaleInfoDto

{

public string Symbol { get; set; }

public int? Unit { get; set; }

public string Description { get; set; }

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public decimal GradePoint { get; set; }

}

public class ClassReportSummaryDto

{

public string ClassName { get; set; }

public string SchoolName { get; set; }

public string TermName { get; set; }

public List<StudentSummaryDto> StudentSummaries { get; set; } = new();

public ClassStatisticsDto Statistics { get; set; }

}

public class StudentSummaryDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public decimal OverallPercentage { get; set; }

public string OverallGrade { get; set; }

public int ClassPosition { get; set; }

public decimal AttendancePercentage { get; set; }

}

public class ClassStatisticsDto

{

public int TotalStudents { get; set; }

public decimal ClassAverage { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public decimal StandardDeviation { get; set; }

public int PassCount { get; set; }

public int FailCount { get; set; }

public decimal PassRate { get; set; }

}

public class ReportCardService : IReportCardService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

private readonly IAssignmentService \_assignmentService;

public ReportCardService(

SchoolManagementContext context,

IGradingService gradingService,

IAssignmentService assignmentService)

{

\_context = context;

\_gradingService = gradingService;

\_assignmentService = assignmentService;

}

public async Task<DetailedReportCardDto> GenerateDetailedReportCardAsync(int studentId, int termId)

{

var student = await \_context.Students

.Include(s => s.Class)

.ThenInclude(c => c.School)

.ThenInclude(sc => sc.GradingBody)

.FirstOrDefaultAsync(s => s.Id == studentId);

var term = await \_context.Terms

.Include(t => t.SchoolYear)

.FirstOrDefaultAsync(t => t.Id == termId);

// Get all subject results

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Subject)

.ThenInclude(s => s.GradingScheme)

.ThenInclude(gs => gs.GradeScales)

.Include(r => r.Teacher)

.ToListAsync();

// Get continuous assessment marks

var subjectPerformances = new List<SubjectPerformanceDto>();

decimal totalMarks = 0;

decimal maxPossibleMarks = 0;

var allGradePoints = new List<decimal>();

foreach (var result in results)

{

var continuousMarks = await \_assignmentService.CalculateContinuousAssessmentMarkAsync(

studentId, result.SubjectId, termId);

// Get detailed assessment breakdown

var assessmentMarks = await GetAssessmentBreakdownAsync(studentId, result.SubjectId, termId);

// Calculate subject position

var subjectPosition = await CalculateSubjectPositionAsync(studentId, result.SubjectId, termId);

// Get class average for this subject

var classAverage = await CalculateSubjectClassAverageAsync(student.ClassId, result.SubjectId, termId);

// Calculate performance trend

var trend = await CalculatePerformanceTrendAsync(studentId, result.SubjectId, termId);

// Get grade point for GPA calculation

var gradeScale = result.Subject.GradingScheme.GradeScales

.FirstOrDefault(gs => gs.Symbol == result.Grade);

var gradePoint = gradeScale?.GradePoint ?? 0;

allGradePoints.Add(gradePoint);

var subjectPerformance = new SubjectPerformanceDto

{

SubjectName = result.Subject.Name,

SubjectCode = result.Subject.Code,

TeacherName = $"{result.Teacher.FirstName} {result.Teacher.LastName}",

AssessmentMarks = assessmentMarks,

ContinuousAssessmentMark = continuousMarks,

ExamMark = result.Score,

TotalMark = (continuousMarks \* 0.4m) + (result.Score \* 0.6m), // 40% CA, 60% Exam

Percentage = result.Score,

Grade = result.Grade,

Unit = result.Unit,

GradePoint = gradePoint,

Comments = result.Comments,

SubjectPosition = subjectPosition.position,

TotalStudentsInSubject = subjectPosition.totalStudents,

ClassAverage = classAverage,

PerformanceTrend = trend

};

subjectPerformances.Add(subjectPerformance);

totalMarks += subjectPerformance.TotalMark;

maxPossibleMarks += 100; // Assuming 100 is max for each subject

}

// Calculate overall performance

var overallPercentage = maxPossibleMarks > 0 ? (totalMarks / maxPossibleMarks) \* 100 : 0;

var classPosition = await CalculateClassPositionAsync(studentId, termId);

var classAvg = await CalculateClassAverageAsync(student.ClassId, termId);

var gpa = allGradePoints.Any() ? allGradePoints.Average() : 0;

// Get overall grade

var overallGradeInfo = await \_gradingService.CalculateGradeAsync(

overallPercentage,

student.Class.School.GradingBody.GradingSchemes.FirstOrDefault()?.Id ?? 0);

// Get attendance summary

var attendanceSummary = await GetAttendanceSummaryAsync(studentId, term.StartDate, term.EndDate);

// Get teacher comments

var teacherComments = await GetTeacherCommentsAsync(studentId, termId);

var headTeacherComments = await GetHeadTeacherCommentsAsync(studentId, termId);

return new DetailedReportCardDto

{

Student = new StudentInfoDto

{

FullName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

ClassName = student.Class.Name,

DateOfBirth = student.DateOfBirth,

Gender = student.Gender

},

School = new SchoolInfoDto

{

Name = student.Class.School.Name,

Address = student.Class.School.Address,

Phone = student.Class.School.Phone,

Email = student.Class.School.Email,

Logo = student.Class.School.Logo

},

Term = new TermInfoDto

{

Name = term.Name,

StartDate = term.StartDate,

EndDate = term.EndDate,

SchoolYear = term.SchoolYear.Name

},

SubjectPerformances = subjectPerformances,

OverallPerformance = new OverallPerformanceDto

{

TotalMarks = totalMarks,

MaxPossibleMarks = maxPossibleMarks,

OverallPercentage = overallPercentage,

OverallGrade = overallGradeInfo.grade,

OverallUnit = overallGradeInfo.unit,

GPA = gpa,

ClassPosition = classPosition.position,

TotalStudentsInClass = classPosition.totalStudents,

ClassAverage = classAvg,

PerformanceSummary = GeneratePerformanceSummary(overallPercentage, classPosition.position, classPosition.totalStudents)

},

Attendance = attendanceSummary,

TeacherComments = teacherComments,

HeadTeacherComments = headTeacherComments,

GeneratedAt = DateTime.UtcNow,

GradingScheme = new GradingSchemeInfoDto

{

Name = student.Class.School.GradingBody.GradingSchemes.FirstOrDefault()?.Name,

GradingBody = student.Class.School.GradingBody.Name,

GradeScales = student.Class.School.GradingBody.GradingSchemes.FirstOrDefault()?.GradeScales

.Select(gs => new GradeScaleInfoDto

{

Symbol = gs.Symbol,

Unit = gs.Unit,

Description = gs.Description,

MinPercentage = gs.MinPercentage,

MaxPercentage = gs.MaxPercentage,

GradePoint = gs.GradePoint

}).ToList() ?? new List<GradeScaleInfoDto>()

}

};

}

public async Task<byte[]> GenerateReportCardPdfAsync(int studentId, int termId)

{

var reportCard = await GenerateDetailedReportCardAsync(studentId, termId);

// Generate comprehensive PDF report card

var html = GenerateReportCardHtml(reportCard);

// Convert HTML to PDF (using a library like SelectPdf, wkHtmlToPdf, or similar)

// For now, returning the HTML as bytes for demonstration

return System.Text.Encoding.UTF8.GetBytes(html);

}

public async Task<ClassReportSummaryDto> GenerateClassReportSummaryAsync(int classId, int termId)

{

var classEntity = await \_context.Classes

.Include(c => c.School)

.FirstOrDefaultAsync(c => c.Id == classId);

var term = await \_context.Terms.FindAsync(termId);

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.Include(s => s.Results.Where(r => r.TermId == termId))

.ToListAsync();

var studentSummaries = new List<StudentSummaryDto>();

var allPercentages = new List<decimal>();

foreach (var student in students)

{

var studentResults = student.Results.Where(r => r.TermId == termId).ToList();

var overallPercentage = studentResults.Any() ? studentResults.Average(r => r.Score) : 0;

allPercentages.Add(overallPercentage);

var overallGrade = "";

if (studentResults.Any())

{

var firstSubject = studentResults.First().Subject;

var gradeInfo = await \_gradingService.CalculateGradeAsync(overallPercentage, firstSubject.GradingSchemeId);

overallGrade = gradeInfo.grade;

}

var attendancePercentage = await GetStudentAttendancePercentageAsync(student.Id, term.StartDate, term.EndDate);

studentSummaries.Add(new StudentSummaryDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

OverallPercentage = overallPercentage,

OverallGrade = overallGrade,

AttendancePercentage = attendancePercentage

});

}

// Sort by percentage and assign positions

var sortedSummaries = studentSummaries.OrderByDescending(s => s.OverallPercentage).ToList();

for (int i = 0; i < sortedSummaries.Count; i++)

{

sortedSummaries[i].ClassPosition = i + 1;

}

// Calculate class statistics

var statistics = CalculateClassStatistics(allPercentages);

return new ClassReportSummaryDto

{

ClassName = classEntity.Name,

SchoolName = classEntity.School.Name,

TermName = term.Name,

StudentSummaries = sortedSummaries,

Statistics = statistics

};

}

public async Task<byte[]> GenerateClassReportPdfAsync(int classId, int termId)

{

var classReport = await GenerateClassReportSummaryAsync(classId, termId);

var html = GenerateClassReportHtml(classReport);

return System.Text.Encoding.UTF8.GetBytes(html);

}

private async Task<List<AssessmentMarkDto>> GetAssessmentBreakdownAsync(int studentId, int subjectId, int termId)

{

var submissions = await \_context.AssignmentSubmissions

.Where(s => s.StudentId == studentId

&& s.Assignment.SubjectId == subjectId

&& s.Status == SubmissionStatus.Graded)

.Include(s => s.Assignment)

.ToListAsync();

return submissions.Select(s => new AssessmentMarkDto

{

AssessmentType = $"{s.Assignment.Type} - {s.Assignment.Title}",

Mark = s.Score ?? 0,

MaxMark = s.Assignment.TotalMarks,

Percentage = s.Assignment.TotalMarks > 0 ? ((s.Score ?? 0) / s.Assignment.TotalMarks) \* 100 : 0,

DateAssessed = s.SubmittedAt

}).ToList();

}

private async Task<(int position, int totalStudents)> CalculateSubjectPositionAsync(int studentId, int subjectId, int termId)

{

var subjectResults = await \_context.Results

.Where(r => r.SubjectId == subjectId && r.TermId == termId)

.OrderByDescending(r => r.Score)

.Select(r => new { r.StudentId, r.Score })

.ToListAsync();

var position = subjectResults.FindIndex(r => r.StudentId == studentId) + 1;

return (position, subjectResults.Count);

}

private async Task<decimal> CalculateSubjectClassAverageAsync(int classId, int subjectId, int termId)

{

var classResults = await \_context.Results

.Where(r => r.SubjectId == subjectId && r.TermId == termId && r.Student.ClassId == classId)

.AverageAsync(r => (decimal?)r.Score);

return classResults ?? 0;

}

private async Task<string> CalculatePerformanceTrendAsync(int studentId, int subjectId, int termId)

{

// Get previous term's results for comparison

var currentTerm = await \_context.Terms.FindAsync(termId);

var previousTerm = await \_context.Terms

.Where(t => t.SchoolYearId == currentTerm.SchoolYearId && t.EndDate < currentTerm.StartDate)

.OrderByDescending(t => t.EndDate)

.FirstOrDefaultAsync();

if (previousTerm == null) return "New";

var currentResult = await \_context.Results

.FirstOrDefaultAsync(r => r.StudentId == studentId && r.SubjectId == subjectId && r.TermId == termId);

var previousResult = await \_context.Results

.FirstOrDefaultAsync(r => r.StudentId == studentId && r.SubjectId == subjectId && r.TermId == previousTerm.Id);

if (currentResult == null || previousResult == null) return "Insufficient Data";

var difference = currentResult.Score - previousResult.Score;

return difference switch

{

> 5 => "Improving",

< -5 => "Declining",

\_ => "Stable"

};

}

private async Task<(int position, int totalStudents)> CalculateClassPositionAsync(int studentId, int termId)

{

var student = await \_context.Students.FindAsync(studentId);

var classAverages = await \_context.Results

.Where(r => r.TermId == termId && r.Student.ClassId == student.ClassId)

.GroupBy(r => r.StudentId)

.Select(g => new { StudentId = g.Key, Average = g.Average(r => r.Score) })

.OrderByDescending(x => x.Average)

.ToListAsync();

var position = classAverages.FindIndex(x => x.StudentId == studentId) + 1;

return (position, classAverages.Count);

}

private async Task<decimal> CalculateClassAverageAsync(int classId, int termId)

{

var classResults = await \_context.Results

.Where(r => r.TermId == termId && r.Student.ClassId == classId)

.GroupBy(r => r.StudentId)

.Select(g => g.Average(r => r.Score))

.ToListAsync();

return classResults.Any() ? classResults.Average() : 0;

}

private async Task<AttendanceSummaryDto> GetAttendanceSummaryAsync(int studentId, DateTime startDate, DateTime endDate)

{

var attendances = await \_context.Attendances

.Where(a => a.StudentId == studentId && a.Date >= startDate && a.Date <= endDate)

.ToListAsync();

var totalDays = CalculateSchoolDays(startDate, endDate);

var presentDays = attendances.Count(a => a.Status == AttendanceStatus.Present);

var absentDays = attendances.Count(a => a.Status == AttendanceStatus.Absent);

var lateDays = attendances.Count(a => a.Status == AttendanceStatus.Late);

var attendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0;

var attendanceGrade = attendancePercentage switch

{

>= 95 => "Excellent",

>= 90 => "Very Good",

>= 85 => "Good",

>= 80 => "Satisfactory",

>= 75 => "Needs Improvement",

\_ => "Poor"

};

return new AttendanceSummaryDto

{

TotalSchoolDays = totalDays,

DaysPresent = presentDays,

DaysAbsent = absentDays,

DaysLate = lateDays,

AttendancePercentage = attendancePercentage,

AttendanceGrade = attendanceGrade

};

}

private async Task<decimal> GetStudentAttendancePercentageAsync(int studentId, DateTime startDate, DateTime endDate)

{

var attendance = await GetAttendanceSummaryAsync(studentId, startDate, endDate);

return attendance.AttendancePercentage;

}

private int CalculateSchoolDays(DateTime startDate, DateTime endDate)

{

var days = 0;

for (var date = startDate; date <= endDate; date = date.AddDays(1))

{

if (date.DayOfWeek != DayOfWeek.Saturday && date.DayOfWeek != DayOfWeek.Sunday)

days++;

}

return days;

}

private async Task<string> GetTeacherCommentsAsync(int studentId, int termId)

{

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Teacher)

.ToListAsync();

if (!results.Any()) return "No teacher comments available.";

var comments = results

.Where(r => !string.IsNullOrEmpty(r.Comments))

.Select(r => $"{r.Subject.Name}: {r.Comments}")

.ToList();

return comments.Any() ? string.Join(" | ", comments) : "Good progress overall.";

}

private async Task<string> GetHeadTeacherCommentsAsync(int studentId, int termId)

{

// This could be stored in a separate table or calculated based on overall performance

var student = await \_context.Students

.Include(s => s.Results.Where(r => r.TermId == termId))

.FirstOrDefaultAsync(s => s.Id == studentId);

if (student == null || !student.Results.Any()) return "Keep up the good work!";

var overallAverage = student.Results.Average(r => r.Score);

return overallAverage switch

{

>= 80 => "Excellent performance! Continue with this outstanding work.",

>= 70 => "Good work! Keep striving for excellence.",

>= 60 => "Satisfactory progress. Focus on improving weaker areas.",

>= 50 => "You can do better. Please seek additional support.",

\_ => "Significant improvement needed. Parent conference recommended."

};

}

private string GeneratePerformanceSummary(decimal percentage, int position, int totalStudents)

{

var positionText = position switch

{

1 => "1st",

2 => "2nd",

3 => "3rd",

\_ => $"{position}th"

};

var performanceLevel = percentage switch

{

>= 80 => "excellent",

>= 70 => "good",

>= 60 => "satisfactory",

>= 50 => "below average",

\_ => "poor"

};

return $"Achieved {positionText} position out of {totalStudents} students with {performanceLevel} performance ({percentage:F1}%).";

}

private ClassStatisticsDto CalculateClassStatistics(List<decimal> percentages)

{

if (!percentages.Any()) return new ClassStatisticsDto();

var average = percentages.Average();

var variance = percentages.Average(p => Math.Pow((double)(p - average), 2));

var standardDeviation = (decimal)Math.Sqrt(variance);

var passCount = percentages.Count(p => p >= 50); // Assuming 50% is pass mark

var failCount = percentages.Count - passCount;

var passRate = percentages.Count > 0 ? (decimal)passCount / percentages.Count \* 100 : 0;

return new ClassStatisticsDto

{

TotalStudents = percentages.Count,

ClassAverage = average,

HighestScore = percentages.Max(),

LowestScore = percentages.Min(),

StandardDeviation = standardDeviation,

PassCount = passCount,

FailCount = failCount,

PassRate = passRate

};

}

private string GenerateReportCardHtml(DetailedReportCardDto reportCard)

{

return $@"

<!DOCTYPE html>

<html>

<head>

<title>Report Card - {reportCard.Student.FullName}</title>

<style>

body {{ font-family: Arial, sans-serif; margin: 20px; }}

.header {{ text-align: center; border-bottom: 2px solid #333; padding-bottom: 20px; }}

.school-info {{ text-align: center; margin-bottom: 20px; }}

.student-info {{ display: flex; justify-content: space-between; margin: 20px 0; }}

.subjects-table {{ width: 100%; border-collapse: collapse; margin: 20px 0; }}

.subjects-table th, .subjects-table td {{ border: 1px solid #333; padding: 8px; text-align: center; }}

.subjects-table th {{ background-color: #f0f0f0; }}

.grading-scale {{ margin-top: 20px; }}

.comments {{ margin: 20px 0; padding: 15px; background-color: #f9f9f9; }}

.footer {{ text-align: center; margin-top: 30px; font-size: 12px; }}

</style>

</head>

<body>

<div class='header'>

<h1>{reportCard.School.Name}</h1>

<p>{reportCard.School.Address}</p>

<p>Phone: {reportCard.// =================== MODELS/ENTITIES ===================

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

// Core System Entities

public class GradingBody

{

public int Id { get; set; }

public string Name { get; set; } // e.g., "Zimbabwe Education Board", "Cambridge"

public string Country { get; set; }

public string Description { get; set; }

public DateTime CreatedAt { get; set; }

public List<GradingScheme> GradingSchemes { get; set; } = new();

public List<School> Schools { get; set; } = new();

}

public class GradingScheme

{

public int Id { get; set; }

public string Name { get; set; } // e.g., "Primary Grading", "Secondary Grading"

public int GradingBodyId { get; set; }

public GradingBody GradingBody { get; set; }

public List<GradeScale> GradeScales { get; set; } = new();

public List<Subject> Subjects { get; set; } = new();

}

public class GradeScale

{

public int Id { get; set; }

public int GradingSchemeId { get; set; }

public GradingScheme GradingScheme { get; set; }

public string Symbol { get; set; } // A, B, C, D, F

public int? Unit { get; set; } // 1, 2, 3, 4, 5 (nullable for systems that don't use units)

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public string Description { get; set; } // "Excellent", "Good", "Average"

public decimal GradePoint { get; set; } // For GPA calculations

}

public class School

{

public int Id { get; set; }

public string Name { get; set; }

public string Address { get; set; }

public string Phone { get; set; }

public string Email { get; set; }

public string Logo { get; set; }

public int GradingBodyId { get; set; }

public GradingBody GradingBody { get; set; }

public bool IsOnlineSchoolingEnabled { get; set; }

public DateTime CreatedAt { get; set; }

// Navigation properties

public List<Class> Classes { get; set; } = new();

public List<Teacher> Teachers { get; set; } = new();

public List<Student> Students { get; set; } = new();

public List<Subject> Subjects { get; set; } = new();

public List<SchoolYear> SchoolYears { get; set; } = new();

public List<Exam> Exams { get; set; } = new();

public List<OnlineTutor> OnlineTutors { get; set; } = new();

}

public class SchoolYear

{

public int Id { get; set; }

public string Name { get; set; } // "2024/2025"

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<Term> Terms { get; set; } = new();

}

public class Term

{

public int Id { get; set; }

public string Name { get; set; } // "Term 1", "First Semester"

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public int SchoolYearId { get; set; }

public SchoolYear SchoolYear { get; set; }

public List<Result> Results { get; set; } = new();

public List<Exam> Exams { get; set; } = new();

}

public class Class

{

public int Id { get; set; }

public string Name { get; set; } // "Grade 1A", "Form 4B"

public string Level { get; set; } // "Primary", "Secondary"

public int Capacity { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int? ClassTeacherId { get; set; } // Head teacher

public Teacher ClassTeacher { get; set; }

public List<Student> Students { get; set; } = new();

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Timetable> Timetables { get; set; } = new();

public List<Attendance> Attendances { get; set; } = new();

}

public class Subject

{

public int Id { get; set; }

public string Name { get; set; }

public string Code { get; set; } // "MATH101", "ENG101"

public string Description { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int GradingSchemeId { get; set; }

public GradingScheme GradingScheme { get; set; }

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Result> Results { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<OnlineTutorSubject> OnlineTutorSubjects { get; set; } = new();

}

public class ClassSubject

{

public int Id { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public List<TimetableSlot> TimetableSlots { get; set; } = new();

}

// User Management

public abstract class User

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public string Address { get; set; }

public string ProfilePicture { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? LastLogin { get; set; }

public bool IsActive { get; set; }

public string UserType { get; set; } // Discriminator for inheritance

}

public class Teacher : User

{

public string EmployeeId { get; set; }

public string Qualification { get; set; }

public DateTime DateOfHire { get; set; }

public decimal Salary { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Class> ManagedClasses { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<HandwritingValidation> HandwritingValidations { get; set; } = new();

}

public class Student : User

{

public string StudentNumber { get; set; }

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; }

public DateTime EnrollmentDate { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<Parent> Parents { get; set; } = new();

public List<Result> Results { get; set; } = new();

public List<Attendance> Attendances { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<HandwritingSample> HandwritingSamples { get; set; } = new();

public List<OnlineTutoringSession> TutoringSessionsAsStudent { get; set; } = new();

public List<StudentTimetable> StudentTimetables { get; set; } = new();

}

public class Parent : User

{

public string Relationship { get; set; } // Father, Mother, Guardian

public string Occupation { get; set; }

public string WhatsAppNumber { get; set; }

public bool ReceiveNotifications { get; set; }

public bool ReceiveWhatsAppNotifications { get; set; }

public bool ReceiveEmailNotifications { get; set; }

public bool ReceiveSMSNotifications { get; set; }

public List<Student> Children { get; set; } = new();

public List<Notification> Notifications { get; set; } = new();

}

// Assessment & Results

public class Result

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int TermId { get; set; }

public Term Term { get; set; }

public decimal Score { get; set; }

public string Grade { get; set; } // Calculated from GradeScale

public int? Unit { get; set; } // Calculated from GradeScale

public string Comments { get; set; }

public DateTime DateRecorded { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public string AssessmentType { get; set; } // "Continuous", "Exam", "Assignment"

}

public class Exam

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int TermId { get; set; }

public Term Term { get; set; }

public List<ExamSubject> ExamSubjects { get; set; } = new();

}

public class ExamSubject

{

public int Id { get; set; }

public int ExamId { get; set; }

public Exam Exam { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public DateTime ExamDate { get; set; }

public TimeSpan Duration { get; set; }

public decimal TotalMarks { get; set; }

public string Instructions { get; set; }

}

// Attendance

public class Attendance

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

public string Remarks { get; set; }

public int RecordedByTeacherId { get; set; }

public Teacher RecordedByTeacher { get; set; }

}

public enum AttendanceStatus

{

Present,

Absent,

Late,

Excused

}

// Assignment & Homework System

public class Assignment

{

public int Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public AssignmentType Type { get; set; }

public DateTime DueDate { get; set; }

public decimal TotalMarks { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public bool IsOnlinePlatformWork { get; set; }

public string Instructions { get; set; }

public DateTime CreatedAt { get; set; }

public List<AssignmentSubmission> Submissions { get; set; } = new();

public List<AssignmentQuestion> Questions { get; set; } = new();

}

public enum AssignmentType

{

Homework,

Classwork,

Assignment,

Project,

Quiz

}

public class AssignmentQuestion

{

public int Id { get; set; }

public int AssignmentId { get; set; }

public Assignment Assignment { get; set; }

public string Question { get; set; }

public QuestionType Type { get; set; }

public string CorrectAnswer { get; set; }

public decimal Marks { get; set; }

public int OrderIndex { get; set; }

public List<QuestionOption> Options { get; set; } = new(); // For multiple choice

}

public enum QuestionType

{

MultipleChoice,

TrueFalse,

ShortAnswer,

Essay,

Handwritten

}

public class QuestionOption

{

public int Id { get; set; }

public int QuestionId { get; set; }

public AssignmentQuestion Question { get; set; }

public string OptionText { get; set; }

public bool IsCorrect { get; set; }

public char OptionLetter { get; set; } // A, B, C, D

}

public class AssignmentSubmission

{

public int Id { get; set; }

public int AssignmentId { get; set; }

public Assignment Assignment { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public DateTime SubmittedAt { get; set; }

public decimal? Score { get; set; }

public string Feedback { get; set; }

public SubmissionStatus Status { get; set; }

public bool IsAutoGraded { get; set; }

public List<SubmissionAnswer> Answers { get; set; } = new();

}

public enum SubmissionStatus

{

Submitted,

Graded,

Late,

Missing

}

public class SubmissionAnswer

{

public int Id { get; set; }

public int SubmissionId { get; set; }

public AssignmentSubmission Submission { get; set; }

public int QuestionId { get; set; }

public AssignmentQuestion Question { get; set; }

public string Answer { get; set; }

public string HandwrittenImagePath { get; set; } // For handwritten answers

public decimal? Score { get; set; }

public bool IsCorrect { get; set; }

public string ProcessedText { get; set; } // AI-processed handwritten text

public decimal Confidence { get; set; } // AI confidence level

}

// Timetable System

public class Timetable

{

public int Id { get; set; }

public string Name { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SchoolYearId { get; set; }

public SchoolYear SchoolYear { get; set; }

public bool IsActive { get; set; }

public DateTime CreatedAt { get; set; }

public List<TimetableSlot> Slots { get; set; } = new();

}

public class TimetableSlot

{

public int Id { get; set; }

public int TimetableId { get; set; }

public Timetable Timetable { get; set; }

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public int ClassSubjectId { get; set; }

public ClassSubject ClassSubject { get; set; }

public string Room { get; set; }

public SlotType Type { get; set; }

}

public enum SlotType

{

Regular,

Break,

Lunch,

Assembly,

Sports

}

public class StudentTimetable

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int TimetableSlotId { get; set; }

public TimetableSlot TimetableSlot { get; set; }

public bool IsOptional { get; set; } // For elective subjects

}

// Handwriting Recognition System

public class HandwritingSample

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public string ImagePath { get; set; }

public string ExpectedText { get; set; }

public string RecognizedText { get; set; }

public decimal Confidence { get; set; }

public HandwritingType Type { get; set; }

public bool IsValidated { get; set; }

public int? ValidatedByTeacherId { get; set; }

public Teacher ValidatedByTeacher { get; set; }

public DateTime CreatedAt { get; set; }

public bool IsTrainingData { get; set; }

}

public enum HandwritingType

{

Alphabet,

Number,

Word,

Sentence,

Answer

}

public class HandwritingValidation

{

public int Id { get; set; }

public int HandwritingSampleId { get; set; }

public HandwritingSample HandwritingSample { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public string CorrectedText { get; set; }

public DateTime ValidatedAt { get; set; }

public ValidationStatus Status { get; set; }

}

public enum ValidationStatus

{

Pending,

Approved,

Corrected,

Rejected

}

public class PersonalizedModel

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public string ModelPath { get; set; } // Local device path

public string CloudModelPath { get; set; } // Cloud backup path

public DateTime LastTrainingDate { get; set; }

public int SampleCount { get; set; }

public decimal Accuracy { get; set; }

public bool IsDeployedLocally { get; set; }

public bool IsDeployedOnCloud { get; set; }

}

// Online Tutoring System

public class OnlineTutor

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public string Bio { get; set; }

public string Qualifications { get; set; }

public decimal HourlyRate { get; set; }

public bool IsVerified { get; set; }

public decimal Rating { get; set; }

public int TotalSessions { get; set; }

public DateTime CreatedAt { get; set; }

public int? SchoolId { get; set; } // Optional - tutor might be independent

public School School { get; set; }

public List<OnlineTutorSubject> TutorSubjects { get; set; } = new();

public List<TutorAvailability> Availability { get; set; } = new();

public List<OnlineTutoringSession> Sessions { get; set; } = new();

}

public class OnlineTutorSubject

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public string GradeLevel { get; set; } // "Primary", "Secondary", "A-Level"

}

public class TutorAvailability

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public bool IsAvailable { get; set; }

}

public class OnlineTutoringSession

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int? SubjectId { get; set; }

public Subject Subject { get; set; }

public DateTime ScheduledDateTime { get; set; }

public TimeSpan Duration { get; set; }

public SessionType Type { get; set; }

public SessionStatus Status { get; set; }

public decimal Cost { get; set; }

public PaymentFrequency PaymentFrequency { get; set; }

public string MeetingUrl { get; set; }

public string SessionNotes { get; set; }

public int? Rating { get; set; }

public string Review { get; set; }

public DateTime? ActualStartTime { get; set; }

public DateTime? ActualEndTime { get; set; }

}

public enum SessionType

{

OneOnOne,

GroupSession

}

public enum SessionStatus

{

Scheduled,

InProgress,

Completed,

Cancelled,

NoShow

}

public enum PaymentFrequency

{

OneTime,

Weekly,

Monthly

}

// Notifications & Communication

public class Notification

{

public int Id { get; set; }

public string Title { get; set; }

public string Message { get; set; }

public NotificationType Type { get; set; }

public int? ParentId { get; set; }

public Parent Parent { get; set; }

public int? StudentId { get; set; }

public Student Student { get; set; }

public int? TeacherId { get; set; }

public Teacher Teacher { get; set; }

public bool IsRead { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? ReadAt { get; set; }

public NotificationChannel Channel { get; set; }

public string ExternalId { get; set; } // For WhatsApp/SMS tracking

}

public enum NotificationType

{

Attendance,

Results,

Assignment,

Exam,

GeneralInfo,

Payment,

Disciplinary,

TutoringSession

}

public enum NotificationChannel

{

InApp,

Email,

SMS,

WhatsApp,

Push

}

public class SocialMediaPost

{

public int Id { get; set; }

public string Content { get; set; }

public string ImagePath { get; set; }

public SocialMediaPlatform Platform { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public DateTime PostedAt { get; set; }

public string ExternalPostId { get; set; }

public int Likes { get; set; }

public int Shares { get; set; }

public int Comments { get; set; }

}

public enum SocialMediaPlatform

{

Facebook,

Twitter,

Instagram,

WhatsApp,

LinkedIn

}

// =================== DBCONTEXT ===================

using Microsoft.EntityFrameworkCore;

public class SchoolManagementContext : DbContext

{

public SchoolManagementContext(DbContextOptions<SchoolManagementContext> options) : base(options) { }

// Core Entities

public DbSet<GradingBody> GradingBodies { get; set; }

public DbSet<GradingScheme> GradingSchemes { get; set; }

public DbSet<GradeScale> GradeScales { get; set; }

public DbSet<School> Schools { get; set; }

public DbSet<SchoolYear> SchoolYears { get; set; }

public DbSet<Term> Terms { get; set; }

public DbSet<Class> Classes { get; set; }

public DbSet<Subject> Subjects { get; set; }

public DbSet<ClassSubject> ClassSubjects { get; set; }

// Users

public DbSet<User> Users { get; set; }

public DbSet<Teacher> Teachers { get; set; }

public DbSet<Student> Students { get; set; }

public DbSet<Parent> Parents { get; set; }

// Assessment

public DbSet<Result> Results { get; set; }

public DbSet<Exam> Exams { get; set; }

public DbSet<ExamSubject> ExamSubjects { get; set; }

public DbSet<Assignment> Assignments { get; set; }

public DbSet<AssignmentQuestion> AssignmentQuestions { get; set; }

public DbSet<QuestionOption> QuestionOptions { get; set; }

public DbSet<AssignmentSubmission> AssignmentSubmissions { get; set; }

public DbSet<SubmissionAnswer> SubmissionAnswers { get; set; }

// Attendance & Timetable

public DbSet<Attendance> Attendances { get; set; }

public DbSet<Timetable> Timetables { get; set; }

public DbSet<TimetableSlot> TimetableSlots { get; set; }

public DbSet<StudentTimetable> StudentTimetables { get; set; }

// Handwriting Recognition

public DbSet<HandwritingSample> HandwritingSamples { get; set; }

public DbSet<HandwritingValidation> HandwritingValidations { get; set; }

public DbSet<PersonalizedModel> PersonalizedModels { get; set; }

// Online Tutoring

public DbSet<OnlineTutor> OnlineTutors { get; set; }

public DbSet<OnlineTutorSubject> OnlineTutorSubjects { get; set; }

public DbSet<TutorAvailability> TutorAvailabilities { get; set; }

public DbSet<OnlineTutoringSession> OnlineTutoringSessions { get; set; }

// Communication

public DbSet<Notification> Notifications { get; set; }

public DbSet<SocialMediaPost> SocialMediaPosts { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// User inheritance configuration

modelBuilder.Entity<User>()

.HasDiscriminator<string>("UserType")

.HasValue<Teacher>("Teacher")

.HasValue<Student>("Student")

.HasValue<Parent>("Parent");

// Configure relationships

ConfigureGradingSystem(modelBuilder);

ConfigureSchoolStructure(modelBuilder);

ConfigureUserRelationships(modelBuilder);

ConfigureAssessmentSystem(modelBuilder);

ConfigureTimetableSystem(modelBuilder);

ConfigureHandwritingSystem(modelBuilder);

ConfigureTutoringSystem(modelBuilder);

ConfigureCommunicationSystem(modelBuilder);

ConfigureIndexes(modelBuilder);

ConfigureConstraints(modelBuilder);

}

private void ConfigureGradingSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeScale>()

.HasOne(gs => gs.GradingScheme)

.WithMany(gs => gs.GradeScales)

.HasForeignKey(gs => gs.GradingSchemeId);

modelBuilder.Entity<GradingScheme>()

.HasOne(gs => gs.GradingBody)

.WithMany(gb => gb.GradingSchemes)

.HasForeignKey(gs => gs.GradingBodyId);

}

private void ConfigureSchoolStructure(ModelBuilder modelBuilder)

{

modelBuilder.Entity<School>()

.HasOne(s => s.GradingBody)

.WithMany(gb => gb.Schools)

.HasForeignKey(s => s.GradingBodyId);

modelBuilder.Entity<Class>()

.HasOne(c => c.ClassTeacher)

.WithMany(t => t.ManagedClasses)

.HasForeignKey(c => c.ClassTeacherId)

.OnDelete(DeleteBehavior.SetNull);

modelBuilder.Entity<ClassSubject>()

.HasKey(cs => cs.Id);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Class)

.WithMany(c => c.ClassSubjects)

.HasForeignKey(cs => cs.ClassId);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Subject)

.WithMany(s => s.ClassSubjects)

.HasForeignKey(cs => cs.SubjectId);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Teacher)

.WithMany(t => t.ClassSubjects)

.HasForeignKey(cs => cs.TeacherId);

}

private void ConfigureUserRelationships(ModelBuilder modelBuilder)

{

// Student-Parent many-to-many

modelBuilder.Entity<Student>()

.HasMany(s => s.Parents)

.WithMany(p => p.Children)

.UsingEntity<Dictionary<string, object>>(

"StudentParent",

j => j.HasOne<Parent>().WithMany().HasForeignKey("ParentId"),

j => j.HasOne<Student>().WithMany().HasForeignKey("StudentId"));

}

private void ConfigureAssessmentSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Result>()

.Property(r => r.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<AssignmentQuestion>()

.Property(aq => aq.Marks)

.HasPrecision(5, 2);

modelBuilder.Entity<SubmissionAnswer>()

.Property(sa => sa.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<SubmissionAnswer>()

.Property(sa => sa.Confidence)

.HasPrecision(5, 4);

}

private void ConfigureTimetableSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<StudentTimetable>()

.HasKey(st => new { st.StudentId, st.TimetableSlotId });

}

private void ConfigureHandwritingSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<HandwritingSample>()

.Property(hs => hs.Confidence)

.HasPrecision(5, 4);

modelBuilder.Entity<PersonalizedModel>()

.Property(pm => pm.Accuracy)

.HasPrecision(5, 4);

}

private void ConfigureTutoringSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<OnlineTutorSubject>()

.HasKey(ots => new { ots.TutorId, ots.SubjectId });

modelBuilder.Entity<OnlineTutoringSession>()

.Property(ots => ots.Cost)

.HasPrecision(10, 2);

modelBuilder.Entity<OnlineTutor>()

.Property(ot => ot.HourlyRate)

.HasPrecision(10, 2);

modelBuilder.Entity<OnlineTutor>()

.Property(ot => ot.Rating)

.HasPrecision(3, 2);

}

private void ConfigureCommunicationSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Notification>()

.HasOne(n => n.Parent)

.WithMany(p => p.Notifications)

.HasForeignKey(n => n.ParentId)

.OnDelete(DeleteBehavior.Cascade);

}

private void ConfigureIndexes(ModelBuilder modelBuilder)

{

// Performance indexes

modelBuilder.Entity<Student>()

.HasIndex(s => s.StudentNumber)

.IsUnique();

modelBuilder.Entity<Teacher>()

.HasIndex(t => t.EmployeeId)

.IsUnique();

modelBuilder.Entity<User>()

.HasIndex(u => u.Email)

.IsUnique();

modelBuilder.Entity<Attendance>()

.HasIndex(a => new { a.StudentId, a.Date });

modelBuilder.Entity<r>()

.HasIndex(r => new { r.StudentId, r.SubjectId, r.TermId });

}

private void ConfigureConstraints(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeScale>()

.HasCheckConstraint("CK\_GradeScale\_Percentage",

"MinPercentage >= 0 AND MaxPercentage <= 100 AND MinPercentage <= MaxPercentage");

modelBuilder.Entity<TimetableSlot>()

.HasCheckConstraint("CK\_TimetableSlot\_Time", "StartTime < EndTime");

}

}

// =================== DTOS ===================

public class StudentResultDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public string ClassName { get; set; }

public string TermName { get; set; }

public List<SubjectResultDto> SubjectResults { get; set; } = new();

public decimal OverallAverage { get; set; }

public string OverallGrade { get; set; }

public int? OverallUnit { get; set; }

public int Position { get; set; }

public int TotalStudents { get; set; }

}

public class SubjectResultDto

{

public string SubjectName { get; set; }

public decimal Score { get; set; }

public string Grade { get; set; }

public int? Unit { get; set; }

public string Comments { get; set; }

public string TeacherName { get; set; }

}

public class TimetableDto

{

public int Id { get; set; }

public string ClassName { get; set; }

public List<TimetableSlotDto> Slots { get; set; } = new();

}

public class TimetableSlotDto

{

public DayOfWeek DayOfWeek { get; set; }

public string StartTime { get; set; }

public string EndTime { get; set; }

public string SubjectName { get; set; }

public string TeacherName { get; set; }

public string Room { get; set; }

public SlotType Type { get; set; }

}

public class AttendanceReportDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public int TotalDays { get; set; }

public int PresentDays { get; set; }

public int AbsentDays { get; set; }

public int LateDays { get; set; }

public decimal AttendancePercentage { get; set; }

}

public class HandwritingRecognitionDto

{

public string ImageBase64 { get; set; }

public int StudentId { get; set; }

public HandwritingType Type { get; set; }

public string ExpectedText { get; set; }

}

public class TutoringSessionDto

{

public int TutorId { get; set; }

public int StudentId { get; set; }

public int? SubjectId { get; set; }

public DateTime ScheduledDateTime { get; set; }

public TimeSpan Duration { get; set; }

public SessionType Type { get; set; }

public PaymentFrequency PaymentFrequency { get; set; }

}

// =================== SERVICE INTERFACES ===================

public interface IGradingService

{

Task<GradingBody> CreateGradingBodyAsync(GradingBody gradingBody);

Task<GradingScheme> CreateGradingSchemeAsync(GradingScheme gradingScheme);

Task<GradeScale> AddGradeScaleAsync(GradeScale gradeScale);

Task<(string grade, int? unit)> CalculateGradeAsync(decimal score, int gradingSchemeId);

Task<List<GradingBody>> GetAllGradingBodiesAsync();

Task<List<GradingScheme>> GetGradingSchemesByBodyAsync(int gradingBodyId);

}

public interface IStudentService

{

Task<Student> CreateStudentAsync(Student student);

Task<Student> UpdateStudentAsync(Student student);

Task<List<Student>> GetStudentsByClassAsync(int classId);

Task<List<Student>> GetStudentsBySchoolAsync(int schoolId);

Task<Student> GetStudentByNumberAsync(string studentNumber, int schoolId);

Task<bool> AssignStudentToClassAsync(int studentId, int classId);

Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId);

Task<byte[]> GenerateStudentReportCardAsync(int studentId, int termId);

}

public interface ITeacherService

{

Task<Teacher> CreateTeacherAsync(Teacher teacher);

Task<List<Teacher>> GetTeachersBySchoolAsync(int schoolId);

Task<bool> AssignTeacherToSubjectAsync(int teacherId, int classId, int subjectId);

Task<List<ClassSubject>> GetTeacherAssignmentsAsync(int teacherId);

}

public interface IResultService

{

Task<r> RecordResultAsync(r result);

Task<List<r>> BulkRecordResultsAsync(List<r> results);

Task<List<StudentResultDto>> GetClassResultsAsync(int classId, int termId);

Task<byte[]> GenerateClassReportAsync(int classId, int termId);

Task<StudentResultDto> CalculateStudentPositionAsync(int studentId, int termId);

}

public interface IAttendanceService

{

Task<Attendance> RecordAttendanceAsync(Attendance attendance);

Task<List<Attendance>> BulkRecordAttendanceAsync(List<Attendance> attendances);

Task<List<AttendanceReportDto>> GetClassAttendanceReportAsync(int classId, DateTime startDate, DateTime endDate);

Task<AttendanceReportDto> GetStudentAttendanceReportAsync(int studentId, DateTime startDate, DateTime endDate);

}

public interface ITimetableService

{

Task<Timetable> GenerateTimetableAsync(int classId, int schoolYearId);

Task<TimetableDto> GetClassTimetableAsync(int classId);

Task<TimetableDto> GetStudentTimetableAsync(int studentId);

Task<TimetableSlot> UpdateTimetableSlotAsync(TimetableSlot slot);

Task<bool> ValidateTimetableRulesAsync(int timetableId);

Task<List<TimetableSlot>> GetTeacherScheduleAsync(int teacherId, DateTime date);

}

public interface INotificationService

{

Task<Notification> CreateNotificationAsync(Notification notification);

Task SendNotificationAsync(int notificationId);

Task SendBulkNotificationsAsync(List<int> notificationIds);

Task<List<Notification>> GetParentNotificationsAsync(int parentId);

Task<bool> MarkNotificationAsReadAsync(int notificationId);

Task SendWhatsAppNotificationAsync(string phoneNumber, string message);

Task SendEmailNotificationAsync(string email, string subject, string message);

}

public interface IAssignmentService

{

Task<Assignment> CreateAssignmentAsync(Assignment assignment);

Task<AssignmentSubmission> SubmitAssignmentAsync(AssignmentSubmission submission);

Task<AssignmentSubmission> AutoGradeAssignmentAsync(int submissionId);

Task<List<Assignment>> GetClassAssignmentsAsync(int classId);

Task<List<Assignment>> GetStudentAssignmentsAsync(int studentId);

Task<decimal> CalculateContinuousAssessmentMarkAsync(int studentId, int subjectId, int termId);

}

public interface IHandwritingRecognitionService

{

Task<string> RecognizeHandwritingAsync(string imagePath, int studentId);

Task<HandwritingSample> CreateHandwritingSampleAsync(HandwritingRecognitionDto dto);

Task<bool> ValidateHandwritingSampleAsync(int sampleId, string correctedText, int teacherId);

Task TrainPersonalizedModelAsync(int studentId);

Task<PersonalizedModel> DeployModelLocallyAsync(int studentId);

Task<string> GenerateTrainingContentAsync(HandwritingType type, string difficulty = "beginner");

Task<bool> ProcessHandwritingOfflineAsync(string imagePath, int studentId);

}

public interface IOnlineTutoringService

{

Task<OnlineTutor> RegisterTutorAsync(OnlineTutor tutor);

Task<OnlineTutoringSession> BookSessionAsync(TutoringSessionDto sessionDto);

Task<List<OnlineTutor>> SearchTutorsAsync(int? subjectId, string gradeLevel, decimal? maxRate);

Task<List<TutorAvailability>> GetTutorAvailabilityAsync(int tutorId, DateTime date);

Task<OnlineTutoringSession> StartSessionAsync(int sessionId);

Task<OnlineTutoringSession> EndSessionAsync(int sessionId, string notes, int? rating, string review);

Task<decimal> CalculateTutorEarningsAsync(int tutorId, DateTime startDate, DateTime endDate);

}

public interface IExamService

{

Task<Exam> CreateExamAsync(Exam exam);

Task<ExamSubject> AddExamSubjectAsync(ExamSubject examSubject);

Task<List<Exam>> GetSchoolExamsAsync(int schoolId, int termId);

Task<byte[]> GenerateExamTimetableAsync(int examId);

}

public interface ISocialMediaService

{

Task<SocialMediaPost> CreatePostAsync(SocialMediaPost post);

Task<bool> PublishToFacebookAsync(int postId);

Task<bool> PublishToWhatsAppAsync(int postId, List<string> phoneNumbers);

Task<bool> PublishToInstagramAsync(int postId);

Task<List<SocialMediaPost>> GetSchoolPostsAsync(int schoolId);

}

// =================== SERVICE IMPLEMENTATIONS ===================

public class GradingService : IGradingService

{

private readonly SchoolManagementContext \_context;

public GradingService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<GradingBody> CreateGradingBodyAsync(GradingBody gradingBody)

{

gradingBody.CreatedAt = DateTime.UtcNow;

\_context.GradingBodies.Add(gradingBody);

await \_context.SaveChangesAsync();

return gradingBody;

}

public async Task<GradingScheme> CreateGradingSchemeAsync(GradingScheme gradingScheme)

{

\_context.GradingSchemes.Add(gradingScheme);

await \_context.SaveChangesAsync();

return gradingScheme;

}

public async Task<GradeScale> AddGradeScaleAsync(GradeScale gradeScale)

{

\_context.GradeScales.Add(gradeScale);

await \_context.SaveChangesAsync();

return gradeScale;

}

public async Task<(string grade, int? unit)> CalculateGradeAsync(decimal score, int gradingSchemeId)

{

var gradeScale = await \_context.GradeScales

.Where(gs => gs.GradingSchemeId == gradingSchemeId

&& score >= gs.MinPercentage

&& score <= gs.MaxPercentage)

.FirstOrDefaultAsync();

return gradeScale != null ? (gradeScale.Symbol, gradeScale.Unit) : ("F", null);

}

public async Task<List<GradingBody>> GetAllGradingBodiesAsync()

{

return await \_context.GradingBodies

.Include(gb => gb.GradingSchemes)

.ThenInclude(gs => gs.GradeScales)

.ToListAsync();

}

public async Task<List<GradingScheme>> GetGradingSchemesByBodyAsync(int gradingBodyId)

{

return await \_context.GradingSchemes

.Where(gs => gs.GradingBodyId == gradingBodyId)

.Include(gs => gs.GradeScales)

.ToListAsync();

}

}

public class StudentService : IStudentService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

public StudentService(SchoolManagementContext context, IGradingService gradingService)

{

\_context = context;

\_gradingService = gradingService;

}

public async Task<Student> CreateStudentAsync(Student student)

{

student.CreatedAt = DateTime.UtcNow;

student.IsActive = true;

student.UserType = "Student";

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

return student;

}

public async Task<Student> UpdateStudentAsync(Student student)

{

\_context.Students.Update(student);

await \_context.SaveChangesAsync();

return student;

}

public async Task<List<Student>> GetStudentsByClassAsync(int classId)

{

return await \_context.Students

.Where(s => s.ClassId == classId && s.IsActive)

.Include(s => s.Parents)

.OrderBy(s => s.LastName)

.ThenBy(s => s.FirstName)

.ToListAsync();

}

public async Task<List<Student>> GetStudentsBySchoolAsync(int schoolId)

{

return await \_context.Students

.Where(s => s.SchoolId == schoolId && s.IsActive)

.Include(s => s.Class)

.Include(s => s.Parents)

.ToListAsync();

}

public async Task<Student> GetStudentByNumberAsync(string studentNumber, int schoolId)

{

return await \_context.Students

.Where(s => s.StudentNumber == studentNumber && s.SchoolId == schoolId)

.Include(s => s.Class)

.Include(s => s.Parents)

.FirstOrDefaultAsync();

}

public async Task<bool> AssignStudentToClassAsync(int studentId, int classId)

{

var student = await \_context.Students.FindAsync(studentId);

if (student == null) return false;

student.ClassId = classId;

await \_context.SaveChangesAsync();

return true;

}

public async Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId)

{

var student = await \_context.Students

.Include(s => s.Class)

.FirstOrDefaultAsync(s => s.Id == studentId);

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Subject)

.Include(r => r.Subject.GradingScheme)

.ThenInclude(gs => gs.GradeScales)

.Include(r => r.Teacher)

.ToListAsync();

var term = await \_context.Terms.FindAsync(termId);

var subjectResults = results.Select(r => new SubjectResultDto

{

SubjectName = r.Subject.Name,

Score = r.Score,

Grade = r.Grade,

Unit = r.Unit,

Comments = r.Comments,

TeacherName = $"{r.Teacher.FirstName} {r.Teacher.LastName}"

}).ToList();

var overallAverage = results.Any() ? results.Average(r => r.Score) : 0;

var gradingScheme = results.FirstOrDefault()?.Subject.GradingScheme;

var overallGrade = "";

int? overallUnit = null;

if (gradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(overallAverage, gradingScheme.Id);

overallGrade = gradeInfo.grade;

overallUnit = gradeInfo.unit;

}

// Calculate position in class

var classAverages = await \_context.Results

.Where(r => r.TermId == termId && r.Student.ClassId == student.ClassId)

.GroupBy(r => r.StudentId)

.Select(g => new { StudentId = g.Key, Average = g.Average(r => r.Score) })

.OrderByDescending(x => x.Average)

.ToListAsync();

var position = classAverages.FindIndex(x => x.StudentId == studentId) + 1;

return new StudentResultDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

ClassName = student.Class.Name,

TermName = term.Name,

SubjectResults = subjectResults,

OverallAverage = overallAverage,

OverallGrade = overallGrade,

OverallUnit = overallUnit,

Position = position,

TotalStudents = classAverages.Count

};

}

public async Task<byte[]> GenerateStudentReportCardAsync(int studentId, int termId)

{

var results = await GetStudentTermResultsAsync(studentId, termId);

// Here you would use a PDF library like iTextSharp or similar

// For now, returning placeholder

var reportContent = $"""

STUDENT REPORT CARD

Student: {results.StudentName}

Student Number: {results.StudentNumber}

Class: {results.ClassName}

Term: {results.TermName}

SUBJECT RESULTS:

{string.Join("\n", results.SubjectResults.Select(sr =>

$"{sr.SubjectName}: {sr.Score}% ({sr.Grade}{(sr.Unit.HasValue ? $" - Unit {sr.Unit}" : "")}) - {sr.Comments}"))}

OVERALL PERFORMANCE:

Average: {results.OverallAverage:F2}%

Grade: {results.OverallGrade}{(results.OverallUnit.HasValue ? $" - Unit {results.OverallUnit}" : "")}

Position: {results.Position} out of {results.TotalStudents}

""";

return System.Text.Encoding.UTF8.GetBytes(reportContent);

}

}

public class ResultService : IResultService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

private readonly INotificationService \_notificationService;

public ResultService(SchoolManagementContext context, IGradingService gradingService, INotificationService notificationService)

{

\_context = context;

\_gradingService = gradingService;

\_notificationService = notificationService;

}

public async Task<r> RecordResultAsync(r result)

{

// Calculate grade and unit based on score

var subject = await \_context.Subjects

.Include(s => s.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == result.SubjectId);

if (subject?.GradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(result.Score, subject.GradingScheme.Id);

result.Grade = gradeInfo.grade;

result.Unit = gradeInfo.unit;

}

result.DateRecorded = DateTime.UtcNow;

\_context.Results.Add(result);

await \_context.SaveChangesAsync();

// Send notification to parents

await SendResultNotificationToParentsAsync(result);

return result;

}

public async Task<List<r>> BulkRecordResultsAsync(List<r> results)

{

foreach (var result in results)

{

var subject = await \_context.Subjects

.Include(s => s.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == result.SubjectId);

if (subject?.GradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(result.Score, subject.GradingScheme.Id);

result.Grade = gradeInfo.grade;

result.Unit = gradeInfo.unit;

}

result.DateRecorded = DateTime.UtcNow;

}

\_context.Results.AddRange(results);

await \_context.SaveChangesAsync();

// Send notifications

foreach (var result in results)

{

await SendResultNotificationToParentsAsync(result);

}

return results;

}

public async Task<List<StudentResultDto>> GetClassResultsAsync(int classId, int termId)

{

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.Include(s => s.Results.Where(r => r.TermId == termId))

.ThenInclude(r => r.Subject)

.ThenInclude(s => s.GradingScheme)

.ToListAsync();

var studentResults = new List<StudentResultDto>();

foreach (var student in students)

{

var subjectResults = student.Results.Select(r => new SubjectResultDto

{

SubjectName = r.Subject.Name,

Score = r.Score,

Grade = r.Grade,

Unit = r.Unit,

Comments = r.Comments

}).ToList();

var overallAverage = student.Results.Any() ? student.Results.Average(r => r.Score) : 0;

studentResults.Add(new StudentResultDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

SubjectResults = subjectResults,

OverallAverage = overallAverage

});

}

// Calculate positions

var sortedResults = studentResults.OrderByDescending(sr => sr.OverallAverage).ToList();

for (int i = 0; i < sortedResults.Count; i++)

{

sortedResults[i].Position = i + 1;

sortedResults[i].TotalStudents = sortedResults.Count;

}

return sortedResults;

}

public async Task<byte[]> GenerateClassReportAsync(int classId, int termId)

{

var classResults = await GetClassResultsAsync(classId, termId);

var classInfo = await \_context.Classes

.Include(c => c.School)

.FirstOrDefaultAsync(c => c.Id == classId);

var term = await \_context.Terms.FindAsync(termId);

var reportContent = $"""

CLASS PERFORMANCE REPORT

School: {classInfo.School.Name}

Class: {classInfo.Name}

Term: {term.Name}

STUDENT RESULTS:

{string.Join("\n", classResults.Select(sr =>

$"{sr.Position}. {sr.StudentName} ({sr.StudentNumber}) - Average: {sr.OverallAverage:F2}%"))}

CLASS STATISTICS:

Total Students: {classResults.Count}

Class Average: {(classResults.Any() ? classResults.Average(sr => sr.OverallAverage) : 0):F2}%

Highest Score: {(classResults.Any() ? classResults.Max(sr => sr.OverallAverage) : 0):F2}%

Lowest Score: {(classResults.Any() ? classResults.Min(sr => sr.OverallAverage) : 0):F2}%

""";

return System.Text.Encoding.UTF8.GetBytes(reportContent);

}

public async Task<StudentResultDto> CalculateStudentPositionAsync(int studentId, int termId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetStudentTermResultsAsync(studentId, termId);

}

private async Task SendResultNotificationToParentsAsync(r result)

{

var student = await \_context.Students

.Include(s => s.Parents)

.Include(s => s.Class)

.FirstOrDefaultAsync(s => s.Id == result.StudentId);

var subject = await \_context.Subjects.FindAsync(result.SubjectId);

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "New Result Posted",

Message = $"New {subject.Name} result for {student.FirstName}: {result.Score}% ({result.Grade})",

Type = NotificationType.Results,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

}

private async Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId)

{

// Implementation moved to StudentService for better organization

var studentService = new StudentService(\_context, \_gradingService);

return await studentService.GetStudentTermResultsAsync(studentId, termId);

}

}

public class AttendanceService : IAttendanceService

{

private readonly SchoolManagementContext \_context;

private readonly INotificationService \_notificationService;

public AttendanceService(SchoolManagementContext context, INotificationService notificationService)

{

\_context = context;

\_notificationService = notificationService;

}

public async Task<Attendance> RecordAttendanceAsync(Attendance attendance)

{

// Check if attendance already exists for this student and date

var existingAttendance = await \_context.Attendances

.FirstOrDefaultAsync(a => a.StudentId == attendance.StudentId

&& a.Date.Date == attendance.Date.Date);

if (existingAttendance != null)

{

existingAttendance.Status = attendance.Status;

existingAttendance.Remarks = attendance.Remarks;

existingAttendance.RecordedByTeacherId = attendance.RecordedByTeacherId;

}

else

{

\_context.Attendances.Add(attendance);

}

await \_context.SaveChangesAsync();

// Send notification if absent

if (attendance.Status == AttendanceStatus.Absent)

{

await SendAbsenteeNotificationAsync(attendance);

}

return existingAttendance ?? attendance;

}

public async Task<List<Attendance>> BulkRecordAttendanceAsync(List<Attendance> attendances)

{

var results = new List<Attendance>();

foreach (var attendance in attendances)

{

var result = await RecordAttendanceAsync(attendance);

results.Add(result);

}

return results;

}

public async Task<List<AttendanceReportDto>> GetClassAttendanceReportAsync(int classId, DateTime startDate, DateTime endDate)

{

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.ToListAsync();

var attendanceData = await \_context.Attendances

.Where(a => a.ClassId == classId

&& a.Date >= startDate

&& a.Date <= endDate)

.GroupBy(a => a.StudentId)

.Select(g => new

{

StudentId = g.Key,

TotalDays = g.Count(),

PresentDays = g.Count(a => a.Status == AttendanceStatus.Present),

AbsentDays = g.Count(a => a.Status == AttendanceStatus.Absent),

LateDays = g.Count(a => a.Status == AttendanceStatus.Late)

})

.ToListAsync();

var totalSchoolDays = await CalculateSchoolDaysAsync(startDate, endDate);

return students.Select(s =>

{

var attendance = attendanceData.FirstOrDefault(a => a.StudentId == s.Id);

var presentDays = attendance?.PresentDays ?? 0;

var totalDays = Math.Max(attendance?.TotalDays ?? 0, totalSchoolDays);

return new AttendanceReportDto

{

StudentName = $"{s.FirstName} {s.LastName}",

StudentNumber = s.StudentNumber,

TotalDays = totalDays,

PresentDays = presentDays,

AbsentDays = attendance?.AbsentDays ?? 0,

LateDays = attendance?.LateDays ?? 0,

AttendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0

};

}).ToList();

}

public async Task<AttendanceReportDto> GetStudentAttendanceReportAsync(int studentId, DateTime startDate, DateTime endDate)

{

var student = await \_context.Students.FindAsync(studentId);

var attendances = await \_context.Attendances

.Where(a => a.StudentId == studentId

&& a.Date >= startDate

&& a.Date <= endDate)

.ToListAsync();

var totalDays = await CalculateSchoolDaysAsync(startDate, endDate);

var presentDays = attendances.Count(a => a.Status == AttendanceStatus.Present);

var absentDays = attendances.Count(a => a.Status == AttendanceStatus.Absent);

var lateDays = attendances.Count(a => a.Status == AttendanceStatus.Late);

return new AttendanceReportDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

TotalDays = Math.Max(attendances.Count, totalDays),

PresentDays = presentDays,

AbsentDays = absentDays,

LateDays = lateDays,

AttendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0

};

}

private async Task<int> CalculateSchoolDaysAsync(DateTime startDate, DateTime endDate)

{

// Calculate weekdays between dates (excluding weekends)

var days = 0;

for (var date = startDate; date <= endDate; date = date.AddDays(1))

{

if (date.DayOfWeek != DayOfWeek.Saturday && date.DayOfWeek != DayOfWeek.Sunday)

days++;

}

return days;

}

private async Task SendAbsenteeNotificationAsync(Attendance attendance)

{

var student = await \_context.Students

.Include(s => s.Parents)

.FirstOrDefaultAsync(s => s.Id == attendance.StudentId);

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "Student Absent",

Message = $"{student.FirstName} was marked absent on {attendance.Date:yyyy-MM-dd}",

Type = NotificationType.Attendance,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

}

}

public class TimetableService : ITimetableService

{

private readonly SchoolManagementContext \_context;

public TimetableService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<Timetable> GenerateTimetableAsync(int classId, int schoolYearId)

{

var classEntity = await \_context.Classes

.Include(c => c.ClassSubjects)

.ThenInclude(cs => cs.Subject)

.Include(c => c.ClassSubjects)

.ThenInclude(cs => cs.Teacher)

.FirstOrDefaultAsync(c => c.Id == classId);

var timetable = new Timetable

{

Name = $"{classEntity.Name} Timetable {DateTime.Now.Year}",

ClassId = classId,

SchoolYearId = schoolYearId,

IsActive = true,

CreatedAt = DateTime.UtcNow

};

\_context.Timetables.Add(timetable);

await \_context.SaveChangesAsync();

// Generate basic timetable structure

await GenerateBasicTimetableStructureAsync(timetable, classEntity.ClassSubjects.ToList());

return timetable;

}

private async Task GenerateBasicTimetableStructureAsync(Timetable timetable, List<ClassSubject> classSubjects)

{

var timeSlots = new[]

{

(new TimeSpan(8, 0, 0), new TimeSpan(8, 45, 0)),

(new TimeSpan(8, 45, 0), new TimeSpan(9, 30, 0)),

(new TimeSpan(9, 30, 0), new TimeSpan(9, 45, 0)), // Break

(new TimeSpan(9, 45, 0), new TimeSpan(10, 30, 0)),

(new TimeSpan(10, 30, 0), new TimeSpan(11, 15, 0)),

(new TimeSpan(11, 15, 0), new TimeSpan(12, 0, 0)),

(new TimeSpan(12, 0, 0), new TimeSpan(13, 0, 0)), // Lunch

(new TimeSpan(13, 0, 0), new TimeSpan(13, 45, 0)),

(new TimeSpan(13, 45, 0), new TimeSpan(14, 30, 0))

};

var workingDays = new[] { DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday, DayOfWeek.Thursday, DayOfWeek.Friday };

var subjectRotation = 0;

foreach (var day in workingDays)

{

for (int i = 0; i < timeSlots.Length; i++)

{

var (startTime, endTime) = timeSlots[i];

TimetableSlot slot;

if (i == 2) // Break time

{

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Type = SlotType.Break,

Room = "Playground"

};

}

else if (i == 6) // Lunch time

{

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Type = SlotType.Lunch,

Room = "Cafeteria"

};

}

else if (classSubjects.Any())

{

var classSubject = classSubjects[subjectRotation % classSubjects.Count];

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

ClassSubjectId = classSubject.Id,

Type = SlotType.Regular,

Room = $"Room {subjectRotation + 1}"

};

subjectRotation++;

}

else continue;

\_context.TimetableSlots.Add(slot);

}

}

await \_context.SaveChangesAsync();

}

public async Task<TimetableDto> GetClassTimetableAsync(int classId)

{

var timetable = await \_context.Timetables

.Where(t => t.ClassId == classId && t.IsActive)

.Include(t => t.Slots)

.ThenInclude(s => s.ClassSubject)

.ThenInclude(cs => cs.Subject)

.Include(t => t.Slots)

.ThenInclude(s => s.ClassSubject)

.ThenInclude(cs => cs.Teacher)

.Include(t => t.Class)

.FirstOrDefaultAsync();

if (timetable == null) return null;

var slots = timetable.Slots.Select(s => new TimetableSlotDto

{

DayOfWeek = s.DayOfWeek,

StartTime = s.StartTime.ToString(@"hh\:mm"),

EndTime = s.EndTime.ToString(@"hh\:mm"),

SubjectName = s.ClassSubject?.Subject?.Name ?? s.Type.ToString(),

TeacherName = s.ClassSubject?.Teacher != null

? $"{s.ClassSubject.Teacher.FirstName} {s.ClassSubject.Teacher.LastName}"

: "",

Room = s.Room,

Type = s.Type

}).ToList();

return new TimetableDto

{

Id = timetable.Id,

ClassName = timetable.Class.Name,

Slots = slots

};

}

public async Task<TimetableDto> GetStudentTimetableAsync(int studentId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetClassTimetableAsync(student.ClassId);

}

public async Task<TimetableSlot> UpdateTimetableSlotAsync(TimetableSlot slot)

{

\_context.TimetableSlots.Update(slot);

await \_context.SaveChangesAsync();

return slot;

}

public async Task<bool> ValidateTimetableRulesAsync(int timetableId)

{

var slots = await \_context.TimetableSlots

.Where(s => s.TimetableId == timetableId)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Teacher)

.ToListAsync();

// Check for teacher conflicts

var teacherConflicts = slots

.Where(s => s.ClassSubject?.Teacher != null)

.GroupBy(s => new { s.DayOfWeek, s.ClassSubject.TeacherId })

.Where(g => g.Any(s1 => g.Any(s2 => s1.Id != s2.Id

&& s1.StartTime < s2.EndTime

&& s2.StartTime < s1.EndTime)))

.Any();

// Check for room conflicts

var roomConflicts = slots

.Where(s => !string.IsNullOrEmpty(s.Room))

.GroupBy(s => new { s.DayOfWeek, s.Room })

.Where(g => g.Any(s1 => g.Any(s2 => s1.Id != s2.Id

&& s1.StartTime < s2.EndTime

&& s2.StartTime < s1.EndTime)))

.Any();

return !teacherConflicts && !roomConflicts;

}

public async Task<List<TimetableSlot>> GetTeacherScheduleAsync(int teacherId, DateTime date)

{

return await \_context.TimetableSlots

.Where(s => s.ClassSubject.TeacherId == teacherId)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Subject)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Class)

.ToListAsync();

}

}

public class AssignmentService : IAssignmentService

{

private readonly SchoolManagementContext \_context;

private readonly INotificationService \_notificationService;

private readonly IHandwritingRecognitionService \_handwritingService;

public AssignmentService(SchoolManagementContext context, INotificationService notificationService, IHandwritingRecognitionService handwritingService)

{

\_context = context;

\_notificationService = notificationService;

\_handwritingService = handwritingService;

}

public async Task<Assignment> CreateAssignmentAsync(Assignment assignment)

{

assignment.CreatedAt = DateTime.UtcNow;

\_context.Assignments.Add(assignment);

await \_context.SaveChangesAsync();

// Notify students/parents about new assignment

await SendAssignmentNotificationAsync(assignment);

return assignment;

}

public async Task<AssignmentSubmission> SubmitAssignmentAsync(AssignmentSubmission submission)

{

submission.SubmittedAt = DateTime.UtcNow;

submission.Status = submission.SubmittedAt <= await GetAssignmentDueDateAsync(submission.AssignmentId)

? SubmissionStatus.Submitted

: SubmissionStatus.Late;

\_context.AssignmentSubmissions.Add(submission);

await \_context.SaveChangesAsync();

// Process handwritten answers

foreach (var answer in submission.Answers.Where(a => !string.IsNullOrEmpty(a.HandwrittenImagePath)))

{

var recognizedText = await \_handwritingService.RecognizeHandwritingAsync(answer.HandwrittenImagePath, submission.StudentId);

answer.ProcessedText = recognizedText;

}

// Auto-grade if it's an online platform assignment

var assignment = await \_context.Assignments.FindAsync(submission.AssignmentId);

if (assignment.IsOnlinePlatformWork)

{

await AutoGradeAssignmentAsync(submission.Id);

}

return submission;

}

public async Task<AssignmentSubmission> AutoGradeAssignmentAsync(int submissionId)

{

var submission = await \_context.AssignmentSubmissions

.Include(s => s.Assignment)

.ThenInclude(a => a.Questions)

.ThenInclude(q => q.Options)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == submissionId);

decimal totalScore = 0;

decimal maxScore = submission.Assignment.Questions.Sum(q => q.Marks);

foreach (var answer in submission.Answers)

{

var question = submission.Assignment.Questions.First(q => q.Id == answer.QuestionId);

switch (question.Type)

{

case QuestionType.MultipleChoice:

var correctOption = question.Options.FirstOrDefault(o => o.IsCorrect);

if (correctOption != null && answer.Answer == correctOption.OptionLetter.ToString())

{

answer.IsCorrect = true;

answer.Score = question.Marks;

totalScore += question.Marks;

}

break;

case QuestionType.TrueFalse:

if (string.Equals(answer.Answer, question.CorrectAnswer, StringComparison.OrdinalIgnoreCase))

{

answer.IsCorrect = true;

answer.Score = question.Marks;

totalScore += question.Marks;

}

break;

case QuestionType.ShortAnswer:

// Simple string matching - could be enhanced with fuzzy matching

var similarity = CalculateStringSimilarity(answer.Answer, question.CorrectAnswer);

if (similarity > 0.8m)

{

answer.IsCorrect = true;

answer.Score = question.Marks \* similarity;

totalScore += answer.Score.Value;

}

break;

case QuestionType.Handwritten:

// Use processed handwritten text for comparison

var handwritingSimilarity = CalculateStringSimilarity(answer.ProcessedText, question.CorrectAnswer);

if (handwritingSimilarity > 0.7m) // Lower threshold for handwriting

{

answer.IsCorrect = true;

answer.Score = question.Marks \* handwritingSimilarity;

totalScore += answer.Score.Value;

}

break;

}

}

submission.Score = maxScore > 0 ? (totalScore / maxScore) \* 100 : 0;

submission.Status = SubmissionStatus.Graded;

submission.IsAutoGraded = true;

await \_context.SaveChangesAsync();

return submission;

}

public async Task<List<Assignment>> GetClassAssignmentsAsync(int classId)

{

return await \_context.Assignments

.Where(a => a.ClassId == classId)

.Include(a => a.Subject)

.Include(a => a.Teacher)

.Include(a => a.Questions)

.OrderByDescending(a => a.CreatedAt)

.ToListAsync();

}

public async Task<List<Assignment>> GetStudentAssignmentsAsync(int studentId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetClassAssignmentsAsync(student.ClassId);

}

public async Task<decimal> CalculateContinuousAssessmentMarkAsync(int studentId, int subjectId, int termId)

{

var submissions = await \_context.AssignmentSubmissions

.Where(s => s.StudentId == studentId

&& s.Assignment.SubjectId == subjectId

&& s.Status == SubmissionStatus.Graded)

.Include(s => s.Assignment)

.ToListAsync();

if (!submissions.Any()) return 0;

// Weight different assignment types

var weightedScores = submissions.Select(s => new

{

Score = s.Score ?? 0,

Weight = s.Assignment.Type switch

{

AssignmentType.Homework => 0.2m,

AssignmentType.Classwork => 0.3m,

AssignmentType.Assignment => 0.3m,

AssignmentType.Project => 0.4m,

AssignmentType.Quiz => 0.25m,

\_ => 0.25m

}

});

var totalWeightedScore = weightedScores.Sum(ws => ws.Score \* ws.Weight);

var totalWeight = weightedScores.Sum(ws => ws.Weight);

return totalWeight > 0 ? totalWeightedScore / totalWeight : 0;

}

private decimal CalculateStringSimilarity(string str1, string str2)

{

if (string.IsNullOrEmpty(str1) || string.IsNullOrEmpty(str2)) return 0;

// Simple Levenshtein distance-based similarity

var distance = LevenshteinDistance(str1.ToLower(), str2.ToLower());

var maxLength = Math.Max(str1.Length, str2.Length);

return maxLength > 0 ? 1m - (decimal)distance / maxLength : 0;

}

private int LevenshteinDistance(string s1, string s2)

{

if (s1.Length == 0) return s2.Length;

if (s2.Length == 0) return s1.Length;

int[,] d = new int[s1.Length + 1, s2.Length + 1];

for (int i = 0; i <= s1.Length; i++) d[i, 0] = i;

for (int j = 0; j <= s2.Length; j++) d[0, j] = j;

for (int i = 1; i <= s1.Length; i++)

{

for (int j = 1; j <= s2.Length; j++)

{

int cost = s1[i - 1] == s2[j - 1] ? 0 : 1;

d[i, j] = Math.Min(Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1), d[i - 1, j - 1] + cost);

}

}

return d[s1.Length, s2.Length];

}

private async Task<DateTime> GetAssignmentDueDateAsync(int assignmentId)

{

var assignment = await \_context.Assignments.FindAsync(assignmentId);

return assignment.DueDate;

}

private async Task SendAssignmentNotificationAsync(Assignment assignment)

{

var students = await \_context.Students

.Where(s => s.ClassId == assignment.ClassId)

.Include(s => s.Parents)

.ToListAsync();

foreach (var student in students)

{

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "New Assignment",

Message = $"New {assignment.Type} assigned: {assignment.Title}. Due: {assignment.DueDate:yyyy-MM-dd}",

Type = NotificationType.Assignment,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

}

}

}

}

public class HandwritingRecognitionService : IHandwritingRecognitionService

{

private readonly SchoolManagementContext \_context;

private readonly IConfiguration \_configuration;

private readonly HttpClient \_httpClient;

public HandwritingRecognitionService(SchoolManagementContext context, IConfiguration configuration, HttpClient httpClient)

{

\_context = context;

\_configuration = configuration;

\_httpClient = httpClient;

}

public async Task<string> RecognizeHandwritingAsync(string imagePath, int studentId)

{

// First try local model if available

var localResult = await ProcessHandwritingOfflineAsync(imagePath, studentId);

if (localResult) return await GetLocalRecognitionResultAsync(imagePath);

// Fallback to cloud-based recognition

return await ProcessHandwritingCloudAsync(imagePath, studentId);

}

public async Task<HandwritingSample> CreateHandwritingSampleAsync(HandwritingRecognitionDto dto)

{

var imageBytes = Convert.FromBase64String(dto.ImageBase64);

var imagePath = await SaveImageAsync(imageBytes, dto.StudentId);

var recognizedText = await RecognizeHandwritingAsync(imagePath, dto.StudentId);

var sample = new HandwritingSample

{

StudentId = dto.StudentId,

ImagePath = imagePath,

ExpectedText = dto.ExpectedText,

RecognizedText = recognizedText,

Type = dto.Type,

CreatedAt = DateTime.UtcNow,

IsTrainingData = true,

Confidence = CalculateConfidence(dto.ExpectedText, recognizedText)

};

\_context.HandwritingSamples.Add(sample);

await \_context.SaveChangesAsync();

return sample;

}

public async Task<bool> ValidateHandwritingSampleAsync(int sampleId, string correctedText, int teacherId)

{

var sample = await \_context.HandwritingSamples.FindAsync(sampleId);

if (sample == null) return false;

var validation = new HandwritingValidation

{

HandwritingSampleId = sampleId,

TeacherId = teacherId,

CorrectedText = correctedText,

ValidatedAt = DateTime.UtcNow,

Status = ValidationStatus.Approved

};

\_context.HandwritingValidations.Add(validation);

sample.IsValidated = true;

sample.RecognizedText = correctedText;

await \_context.SaveChangesAsync();

// Trigger model retraining if enough samples

await CheckAndTriggerModelRetrainingAsync(sample.StudentId);

return true;

}

public async Task TrainPersonalizedModelAsync(int studentId)

{

var trainingSamples = await \_context.HandwritingSamples

.Where(s => s.StudentId == studentId && s.IsValidated)

.ToListAsync();

if (trainingSamples.Count < 50) // Minimum samples for training

{

throw new InvalidOperationException("Insufficient training samples. Minimum 50 validated samples required.");

}

// Prepare training data

var trainingData = trainingSamples.Select(s => new

{

ImagePath = s.ImagePath,

GroundTruth = s.RecognizedText, // Use validated text

Type = s.Type

}).ToList();

// Call ML training service (this would be implemented with ML.NET or similar)

var modelPath = await TrainModelAsync(studentId, trainingData);

var existingModel = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId);

if (existingModel != null)

{

existingModel.ModelPath = modelPath;

existingModel.LastTrainingDate = DateTime.UtcNow;

existingModel.SampleCount = trainingSamples.Count;

existingModel.Accuracy = await CalculateModelAccuracyAsync(modelPath, studentId);

}

else

{

var newModel = new PersonalizedModel

{

StudentId = studentId,

ModelPath = modelPath,

LastTrainingDate = DateTime.UtcNow,

SampleCount = trainingSamples.Count,

Accuracy = await CalculateModelAccuracyAsync(modelPath, studentId),

IsDeployedLocally = false,

IsDeployedOnCloud = true

};

\_context.PersonalizedModels.Add(newModel);

}

await \_context.SaveChangesAsync();

}

public async Task<PersonalizedModel> DeployModelLocallyAsync(int studentId)

{

var model = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId);

if (model == null) return null;

// Deploy model to local device (implementation depends on your mobile/desktop app architecture)

var localPath = await DeployToLocalDeviceAsync(model.CloudModelPath, studentId);

model.ModelPath = localPath;

model.IsDeployedLocally = true;

await \_context.SaveChangesAsync();

return model;

}

public async Task<string> GenerateTrainingContentAsync(HandwritingType type, string difficulty = "beginner")

{

return type switch

{

HandwritingType.Alphabet => GenerateAlphabetContent(difficulty),

HandwritingType.Number => GenerateNumberContent(difficulty),

HandwritingType.Word => GenerateWordContent(difficulty),

HandwritingType.Sentence => GenerateSentenceContent(difficulty),

\_ => "Practice writing: Hello World"

};

}

public async Task<bool> ProcessHandwritingOfflineAsync(string imagePath, int studentId)

{

var model = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId && m.IsDeployedLocally);

if (model == null) return false;

// Process using local model (implementation depends on your ML framework)

try

{

await ProcessWithLocalModelAsync(imagePath, model.ModelPath);

return true;

}

catch

{

return false;

}

}

private async Task<string> ProcessHandwritingCloudAsync(string imagePath, int studentId)

{

// Call cloud-based handwriting recognition API

var cloudApiUrl = \_configuration["HandwritingRecognition:CloudApiUrl"];

using var content = new MultipartFormDataContent();

var imageBytes = await File.ReadAllBytesAsync(imagePath);

content.Add(new ByteArrayContent(imageBytes), "image", "handwriting.jpg");

content.Add(new StringContent(studentId.ToString()), "studentId");

var response = await \_httpClient.PostAsync(cloudApiUrl, content);

var result = await response.Content.ReadAsStringAsync();

return result; // Assume API returns recognized text

}

private decimal CalculateConfidence(string expected, string recognized)

{

if (string.IsNullOrEmpty(expected) || string.IsNullOrEmpty(recognized)) return 0;

var similarity = 1m - (decimal)LevenshteinDistance(expected.ToLower(), recognized.ToLower()) / Math.Max(expected.Length, recognized.Length);

return Math.Max(0, Math.Min(1, similarity));

}

private int LevenshteinDistance(string s1, string s2)

{

// Same implementation as in AssignmentService

if (s1.Length == 0) return s2.Length;

if (s2.Length == 0) return s1.Length;

int[,] d = new int[s1.Length + 1, s2.Length + 1];

for (int i = 0; i <= s1.Length; i++) d[i, 0] = i;

for (int j = 0; j <= s2.Length; j++) d[0, j] = j;

for (int i = 1; i <= s1.Length; i++)

{

for (int j = 1; j <= s2.Length; j++)

{

int cost = s1[i - 1] == s2[j - 1] ? 0 : 1;

d[i, j] = Math.Min(Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1), d[i - 1, j - 1] + cost);

}

}

return d[s1.Length, s2.Length];

}

private string GenerateAlphabetContent(string difficulty)

{

return difficulty switch

{

"beginner" => "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z",

"intermediate" => "a b c d e f g h i j k l m n o p q r s t u v w x y z",

"advanced" => "Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz",

\_ => "A B C D E"

};

}

private string GenerateNumberContent(string difficulty)

{

return difficulty switch

{

"beginner" => "1 2 3 4 5 6 7 8 9 0",

"intermediate" => "12 34 56 78 90 123 456 789",

"advanced" => "1,234 5,678 9,012 3,456 7,890",

\_ => "1 2 3 4 5"

};

}

private string GenerateWordContent(string difficulty)

{

return difficulty switch

{

"beginner" => "cat dog sun fun run",

"intermediate" => "school book pencil teacher student",

"advanced" => "education mathematics science literature",

\_ => "cat dog"

};

}

private string GenerateSentenceContent(string difficulty)

{

return difficulty switch

{

"beginner" => "The cat sat on the mat.",

"intermediate" => "I love going to school every day.",

"advanced" => "Education is the most powerful weapon which you can use to change the world.",

\_ => "Hello world."

};

}

private async Task<string> SaveImageAsync(byte[] imageBytes, int studentId)

{

var uploadsPath = Path.Combine("uploads", "handwriting", studentId.ToString());

Directory.CreateDirectory(uploadsPath);

var fileName = $"{Guid.NewGuid()}.jpg";

var filePath = Path.Combine(uploadsPath, fileName);

await File.WriteAllBytesAsync(filePath, imageBytes);

return filePath;

}

private async Task CheckAndTriggerModelRetrainingAsync(int studentId)

{

var validatedSamples = await \_context.HandwritingSamples

.CountAsync(s => s.StudentId == studentId && s.IsValidated);

if (validatedSamples >= 50 && validatedSamples % 25 == 0) // Retrain every 25 new samples

{

await TrainPersonalizedModelAsync(studentId);

}

}

private async Task<string> TrainModelAsync(int studentId, object trainingData)

{

// Implementation would use ML.NET or similar framework

var modelPath = $"models/student\_{studentId}\_{DateTime.UtcNow:yyyyMMdd}.model";

// Training logic here...

return modelPath;

}

private async Task<decimal> CalculateModelAccuracyAsync(string modelPath, int studentId)

{

// Test model accuracy on validation set

return 0.85m; // Placeholder

}

private async Task<string> DeployToLocalDeviceAsync(string cloudPath, int studentId)

{

// Deploy to local device storage

return $"local/models/student\_{studentId}.model";

}

private async Task<string> GetLocalRecognitionResultAsync(string imagePath)

{

// Get result from local processing

return "Sample recognized text";

}

private async Task ProcessWithLocalModelAsync(string imagePath, string modelPath)

{

// Process image with local model

await Task.Delay(100); // Placeholder

}

}

// =================== AI ASSESSMENT & VERIFICATION SYSTEM ===================

public interface IAIAssessmentService

{

Task<AIAssessmentResult> AssessSubmissionAsync(int submissionId);

Task<List<AIAssessmentResult>> BatchAssessSubmissionsAsync(List<int> submissionIds);

Task<AIAssessmentResult> ProcessHandwrittenAnswerAsync(int submissionAnswerId);

Task<bool> FlagForTeacherReviewAsync(int assessmentResultId, string reason);

Task<List<AIAssessmentResult>> GetPendingReviewsAsync(int teacherId);

Task<AIAssessmentResult> TeacherVerifyAssessmentAsync(int assessmentResultId, int teacherId, TeacherVerificationDto verification);

}

public class AIAssessmentResult

{

public int Id { get; set; }

public int? SubmissionId { get; set; }

public AssignmentSubmission Submission { get; set; }

public int? SubmissionAnswerId { get; set; }

public SubmissionAnswer SubmissionAnswer { get; set; }

public decimal AIScore { get; set; }

public decimal ConfidenceLevel { get; set; }

public string AIFeedback { get; set; }

public AIAssessmentStatus Status { get; set; }

public bool RequiresTeacherReview { get; set; }

public string ReviewReason { get; set; }

public DateTime ProcessedAt { get; set; }

// Teacher verification

public int? VerifiedByTeacherId { get; set; }

public Teacher VerifiedByTeacher { get; set; }

public decimal? TeacherScore { get; set; }

public string TeacherFeedback { get; set; }

public DateTime? VerifiedAt { get; set; }

public VerificationStatus? VerificationStatus { get; set; }

// Detailed AI analysis

public string HandwritingRecognitionText { get; set; }

public decimal HandwritingConfidence { get; set; }

public List<AIScoreBreakdown> ScoreBreakdowns { get; set; } = new();

}

public enum AIAssessmentStatus

{

Processing,

Completed,

Failed,

PendingReview,

Verified,

Disputed

}

public enum VerificationStatus

{

Approved,

Modified,

Rejected,

NeedsReprocessing

}

public class AIScoreBreakdown

{

public int Id { get; set; }

public int AIAssessmentResultId { get; set; }

public AIAssessmentResult AIAssessmentResult { get; set; }

public string Criterion { get; set; } // "Accuracy", "Completeness", "Clarity", "Grammar"

public decimal Score { get; set; }

public decimal MaxScore { get; set; }

public string Explanation { get; set; }

}

public class TeacherVerificationDto

{

public decimal? OverrideScore { get; set; }

public string TeacherFeedback { get; set; }

public VerificationStatus VerificationStatus { get; set; }

public List<CriterionVerification> CriterionVerifications { get; set; } = new();

}

public class CriterionVerification

{

public string Criterion { get; set; }

public decimal TeacherScore { get; set; }

public string TeacherComment { get; set; }

}

// Add to DbContext

public class SchoolManagementContext : DbContext

{

// ... existing DbSets ...

public DbSet<AIAssessmentResult> AIAssessmentResults { get; set; }

public DbSet<AIScoreBreakdown> AIScoreBreakdowns { get; set; }

// ... existing configuration methods ...

private void ConfigureAIAssessmentSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.AIScore)

.HasPrecision(5, 2);

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.ConfidenceLevel)

.HasPrecision(5, 4);

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.HandwritingConfidence)

.HasPrecision(5, 4);

modelBuilder.Entity<AIScoreBreakdown>()

.Property(sb => sb.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<AIScoreBreakdown>()

.Property(sb => sb.MaxScore)

.HasPrecision(5, 2);

}

}

public class AIAssessmentService : IAIAssessmentService

{

private readonly SchoolManagementContext \_context;

private readonly IHandwritingRecognitionService \_handwritingService;

private readonly INotificationService \_notificationService;

private readonly HttpClient \_httpClient;

private readonly IConfiguration \_configuration;

public AIAssessmentService(

SchoolManagementContext context,

IHandwritingRecognitionService handwritingService,

INotificationService notificationService,

HttpClient httpClient,

IConfiguration configuration)

{

\_context = context;

\_handwritingService = handwritingService;

\_notificationService = notificationService;

\_httpClient = httpClient;

\_configuration = configuration;

}

public async Task<AIAssessmentResult> AssessSubmissionAsync(int submissionId)

{

var submission = await \_context.AssignmentSubmissions

.Include(s => s.Assignment)

.ThenInclude(a => a.Questions)

.ThenInclude(q => q.Options)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == submissionId);

if (submission == null) return null;

var assessmentResult = new AIAssessmentResult

{

SubmissionId = submissionId,

Status = AIAssessmentStatus.Processing,

ProcessedAt = DateTime.UtcNow

};

\_context.AIAssessmentResults.Add(assessmentResult);

await \_context.SaveChangesAsync();

try

{

decimal totalScore = 0;

decimal maxPossibleScore = submission.Assignment.Questions.Sum(q => q.Marks);

var allBreakdowns = new List<AIScoreBreakdown>();

foreach (var answer in submission.Answers)

{

var question = submission.Assignment.Questions.First(q => q.Id == answer.QuestionId);

var answerAssessment = await AssessIndividualAnswerAsync(answer, question);

totalScore += answerAssessment.Score;

allBreakdowns.AddRange(answerAssessment.Breakdowns);

// Update the answer with AI results

answer.Score = answerAssessment.Score;

answer.IsCorrect = answerAssessment.Score >= (question.Marks \* 0.7m); // 70% threshold

answer.ProcessedText = answerAssessment.ProcessedText;

answer.Confidence = answerAssessment.Confidence;

}

// Calculate final score and confidence

assessmentResult.AIScore = maxPossibleScore > 0 ? (totalScore / maxPossibleScore) \* 100 : 0;

assessmentResult.ConfidenceLevel = allBreakdowns.Any() ? allBreakdowns.Average(b => b.Score / b.MaxScore) : 0;

assessmentResult.AIFeedback = GenerateOverallFeedback(allBreakdowns, assessmentResult.AIScore);

assessmentResult.Status = AIAssessmentStatus.Completed;

assessmentResult.ScoreBreakdowns = allBreakdowns;

// Determine if teacher review is needed

var needsReview = DetermineIfTeacherReviewNeeded(assessmentResult, allBreakdowns);

if (needsReview.needed)

{

await FlagForTeacherReviewAsync(assessmentResult.Id, needsReview.reason);

}

// Update submission

submission.Score = assessmentResult.AIScore;

submission.Status = needsReview.needed ? SubmissionStatus.Submitted : SubmissionStatus.Graded;

submission.IsAutoGraded = true;

await \_context.SaveChangesAsync();

// Notify if teacher review is needed

if (needsReview.needed)

{

await NotifyTeacherForReviewAsync(submission.Assignment.TeacherId, assessmentResult.Id);

}

return assessmentResult;

}

catch (Exception ex)

{

assessmentResult.Status = AIAssessmentStatus.Failed;

assessmentResult.AIFeedback = $"Assessment failed: {ex.Message}";

await \_context.SaveChangesAsync();

return assessmentResult;

}

}

public async Task<List<AIAssessmentResult>> BatchAssessSubmissionsAsync(List<int> submissionIds)

{

var results = new List<AIAssessmentResult>();

// Process in batches to avoid overwhelming the system

const int batchSize = 10;

for (int i = 0; i < submissionIds.Count; i += batchSize)

{

var batch = submissionIds.Skip(i).Take(batchSize);

var batchTasks = batch.Select(AssessSubmissionAsync);

var batchResults = await Task.WhenAll(batchTasks);

results.AddRange(batchResults.Where(r => r != null));

}

return results;

}

public async Task<AIAssessmentResult> ProcessHandwrittenAnswerAsync(int submissionAnswerId)

{

var answer = await \_context.SubmissionAnswers

.Include(a => a.Question)

.Include(a => a.Submission)

.ThenInclude(s => s.Student)

.FirstOrDefaultAsync(a => a.Id == submissionAnswerId);

if (answer == null || string.IsNullOrEmpty(answer.HandwrittenImagePath))

return null;

var assessmentResult = new AIAssessmentResult

{

SubmissionAnswerId = submissionAnswerId,

Status = AIAssessmentStatus.Processing,

ProcessedAt = DateTime.UtcNow

};

\_context.AIAssessmentResults.Add(assessmentResult);

await \_context.SaveChangesAsync();

try

{

// Step 1: Handwriting Recognition

var recognizedText = await \_handwritingService.RecognizeHandwritingAsync(

answer.HandwrittenImagePath,

answer.Submission.StudentId);

assessmentResult.HandwritingRecognitionText = recognizedText;

// Calculate handwriting confidence

var handwritingConfidence = await CalculateHandwritingConfidenceAsync(

answer.HandwrittenImagePath, recognizedText);

assessmentResult.HandwritingConfidence = handwritingConfidence;

// Step 2: Content Assessment

var contentAssessment = await AssessAnswerContentAsync(

recognizedText,

answer.Question.CorrectAnswer,

answer.Question.Type,

answer.Question.Marks);

assessmentResult.AIScore = contentAssessment.Score;

assessmentResult.ConfidenceLevel = Math.Min(handwritingConfidence, contentAssessment.Confidence);

assessmentResult.AIFeedback = contentAssessment.Feedback;

assessmentResult.ScoreBreakdowns = contentAssessment.Breakdowns;

// Update the original answer

answer.ProcessedText = recognizedText;

answer.Score = contentAssessment.Score;

answer.Confidence = assessmentResult.ConfidenceLevel;

answer.IsCorrect = contentAssessment.Score >= (answer.Question.Marks \* 0.7m);

// Determine if manual review is needed

var needsReview = handwritingConfidence < 0.8m || contentAssessment.Confidence < 0.8m;

if (needsReview)

{

await FlagForTeacherReviewAsync(assessmentResult.Id,

$"Low confidence: Handwriting={handwritingConfidence:P}, Content={contentAssessment.Confidence:P}");

}

assessmentResult.Status = AIAssessmentStatus.Completed;

await \_context.SaveChangesAsync();

return assessmentResult;

}

catch (Exception ex)

{

assessmentResult.Status = AIAssessmentStatus.Failed;

assessmentResult.AIFeedback = $"Processing failed: {ex.Message}";

await \_context.SaveChangesAsync();

return assessmentResult;

}

}

public async Task<bool> FlagForTeacherReviewAsync(int assessmentResultId, string reason)

{

var assessmentResult = await \_context.AIAssessmentResults.FindAsync(assessmentResultId);

if (assessmentResult == null) return false;

assessmentResult.RequiresTeacherReview = true;

assessmentResult.ReviewReason = reason;

assessmentResult.Status = AIAssessmentStatus.PendingReview;

await \_context.SaveChangesAsync();

return true;

}

public async Task<List<AIAssessmentResult>> GetPendingReviewsAsync(int teacherId)

{

return await \_context.AIAssessmentResults

.Where(ar => ar.RequiresTeacherReview

&& ar.Status == AIAssessmentStatus.PendingReview

&& (ar.Submission.Assignment.TeacherId == teacherId ||

ar.SubmissionAnswer.Submission.Assignment.TeacherId == teacherId))

.Include(ar => ar.Submission)

.ThenInclude(s => s.Student)

.Include(ar => ar.Submission)

.ThenInclude(s => s.Assignment)

.Include(ar => ar.SubmissionAnswer)

.ThenInclude(sa => sa.Question)

.Include(ar => ar.ScoreBreakdowns)

.OrderByDescending(ar => ar.ProcessedAt)

.ToListAsync();

}

public async Task<AIAssessmentResult> TeacherVerifyAssessmentAsync(int assessmentResultId, int teacherId, TeacherVerificationDto verification)

{

var assessmentResult = await \_context.AIAssessmentResults

.Include(ar => ar.ScoreBreakdowns)

.Include(ar => ar.Submission)

.ThenInclude(s => s.Assignment)

.FirstOrDefaultAsync(ar => ar.Id == assessmentResultId);

if (assessmentResult == null) return null;

// Record teacher verification

assessmentResult.VerifiedByTeacherId = teacherId;

assessmentResult.TeacherScore = verification.OverrideScore ?? assessmentResult.AIScore;

assessmentResult.TeacherFeedback = verification.TeacherFeedback;

assessmentResult.VerifiedAt = DateTime.UtcNow;

assessmentResult.VerificationStatus = verification.VerificationStatus;

assessmentResult.Status = AIAssessmentStatus.Verified;

assessmentResult.RequiresTeacherReview = false;

// Update criterion scores if provided

foreach (var criterionVerification in verification.CriterionVerifications)

{

var breakdown = assessmentResult.ScoreBreakdowns

.FirstOrDefault(b => b.Criterion == criterionVerification.Criterion);

if (breakdown != null)

{

breakdown.Score = criterionVerification.TeacherScore;

breakdown.Explanation = criterionVerification.TeacherComment;

}

}

// Update the associated submission/answer

if (assessmentResult.SubmissionId.HasValue)

{

var submission = assessmentResult.Submission;

submission.Score = assessmentResult.TeacherScore;

submission.Status = SubmissionStatus.Graded;

submission.Feedback = verification.TeacherFeedback;

}

else if (assessmentResult.SubmissionAnswerId.HasValue)

{

var answer = await \_context.SubmissionAnswers.FindAsync(assessmentResult.SubmissionAnswerId);

if (answer != null)

{

answer.Score = assessmentResult.TeacherScore;

answer.IsCorrect = assessmentResult.TeacherScore >= (answer.Question.Marks \* 0.7m);

}

}

await \_context.SaveChangesAsync();

// Learn from teacher corrections for future AI improvements

await RecordTeacherCorrectionForLearningAsync(assessmentResult);

return assessmentResult;

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessIndividualAnswerAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

switch (question.Type)

{

case QuestionType.MultipleChoice:

return await AssessMultipleChoiceAsync(answer, question);

case QuestionType.TrueFalse:

return await AssessTrueFalseAsync(answer, question);

case QuestionType.ShortAnswer:

return await AssessShortAnswerAsync(answer, question);

case QuestionType.Essay:

return await AssessEssayAsync(answer, question);

case QuestionType.Handwritten:

if (!string.IsNullOrEmpty(answer.HandwrittenImagePath))

{

var handwrittenResult = await ProcessHandwrittenAnswerAsync(answer.Id);

return (handwrittenResult?.AIScore ?? 0,

handwrittenResult?.ConfidenceLevel ?? 0,

handwrittenResult?.AIFeedback ?? "Processing failed",

handwrittenResult?.ScoreBreakdowns ?? new List<AIScoreBreakdown>());

}

return await AssessShortAnswerAsync(answer, question);

default:

return (0, 0, "Unknown question type", breakdowns);

}

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessMultipleChoiceAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var correctOption = question.Options.FirstOrDefault(o => o.IsCorrect);

var isCorrect = correctOption != null &&

string.Equals(answer.Answer, correctOption.OptionLetter.ToString(), StringComparison.OrdinalIgnoreCase);

var score = isCorrect ? question.Marks : 0;

var feedback = isCorrect ? "Correct answer" : $"Incorrect. The correct answer is {correctOption?.OptionLetter}";

var breakdown = new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = score,

MaxScore = question.Marks,

Explanation = feedback

};

return (score, 1.0m, feedback, new List<AIScoreBreakdown> { breakdown });

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessTrueFalseAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var isCorrect = string.Equals(answer.Answer, question.CorrectAnswer, StringComparison.OrdinalIgnoreCase);

var score = isCorrect ? question.Marks : 0;

var feedback = isCorrect ? "Correct answer" : $"Incorrect. The correct answer is {question.CorrectAnswer}";

var breakdown = new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = score,

MaxScore = question.Marks,

Explanation = feedback

};

return (score, 1.0m, feedback, new List<AIScoreBreakdown> { breakdown });

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessShortAnswerAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

// Assess accuracy

var accuracy = CalculateTextSimilarity(answer.Answer, question.CorrectAnswer);

var accuracyScore = question.Marks \* accuracy;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = accuracyScore,

MaxScore = question.Marks,

Explanation = $"Answer similarity to expected response: {accuracy:P}"

});

var totalScore = accuracyScore;

var confidence = accuracy > 0.6m ? 0.9m : 0.7m; // Lower confidence for low similarity

var feedback = accuracy switch

{

>= 0.9m => "Excellent answer, very close to expected response",

>= 0.7m => "Good answer, mostly correct",

>= 0.5m => "Partially correct, but missing some key points",

\_ => "Answer needs improvement, significantly different from expected response"

};

return (totalScore, confidence, feedback, breakdowns);

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessEssayAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

// This would ideally use advanced NLP/AI services like OpenAI GPT or Azure Cognitive Services

// For now, implementing basic assessment criteria

// Content relevance (40% of marks)

var contentScore = await AssessContentRelevanceAsync(answer.Answer, question.CorrectAnswer);

var contentMarks = question.Marks \* 0.4m \* contentScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Content Relevance",

Score = contentMarks,

MaxScore = question.Marks \* 0.4m,

Explanation = $"Content relevance score: {contentScore:P}"

});

// Grammar and language (30% of marks)

var grammarScore = await AssessGrammarAsync(answer.Answer);

var grammarMarks = question.Marks \* 0.3m \* grammarScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Grammar & Language",

Score = grammarMarks,

MaxScore = question.Marks \* 0.3m,

Explanation = $"Grammar and language quality: {grammarScore:P}"

});

// Structure and organization (30% of marks)

var structureScore = await AssessStructureAsync(answer.Answer);

var structureMarks = question.Marks \* 0.3m \* structureScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Structure & Organization",

Score = structureMarks,

MaxScore = question.Marks \* 0.3m,

Explanation = $"Structure and organization: {structureScore:P}"

});

var totalScore = contentMarks + grammarMarks + structureMarks;

var averageScore = (contentScore + grammarScore + structureScore) / 3;

// Lower confidence for essays as they're more subjective

var confidence = averageScore > 0.7m ? 0.75m : 0.6m;

var feedback = GenerateEssayFeedback(contentScore, grammarScore, structureScore);

return (totalScore, confidence, feedback, breakdowns);

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessAnswerContentAsync(

string recognizedText, string correctAnswer, QuestionType questionType, decimal maxMarks)

{

// Use the appropriate assessment method based on question type

var dummyAnswer = new SubmissionAnswer { Answer = recognizedText };

var dummyQuestion = new AssignmentQuestion

{

CorrectAnswer = correctAnswer,

Type = questionType,

Marks = maxMarks

};

return questionType switch

{

QuestionType.ShortAnswer => await AssessShortAnswerAsync(dummyAnswer, dummyQuestion),

QuestionType.Essay => await AssessEssayAsync(dummyAnswer, dummyQuestion),

\_ => await AssessShortAnswerAsync(dummyAnswer, dummyQuestion)

};

}

private decimal CalculateTextSimilarity(string text1, string text2)

{

if (string.IsNullOrEmpty(text1) || string.IsNullOrEmpty(text2)) return 0;

// Normalize texts

text1 = text1.ToLower().Trim();

text2 = text2.ToLower().Trim();

// Simple word-based similarity

var words1 = text1.Split(' ', StringSplitOptions.RemoveEmptyEntries);

var words2 = text2.Split(' ', StringSplitOptions.RemoveEmptyEntries);

var commonWords = words1.Intersect(words2).Count();

var totalWords = Math.Max(words1.Length, words2.Length);

return totalWords > 0 ? (decimal)commonWords / totalWords : 0;

}

private async Task<decimal> CalculateHandwritingConfidenceAsync(string imagePath, string recognizedText)

{

// This would use image quality metrics and OCR confidence scores

// For now, return a simulated confidence based on text length and clarity

if (string.IsNullOrEmpty(recognizedText)) return 0;

// Simulate confidence calculation

var baseConfidence = 0.8m;

var lengthFactor = Math.Min(recognizedText.Length / 50m, 1m); // Longer text = higher confidence

var clarityFactor = recognizedText.Count(char.IsLetter) / (decimal)recognizedText.Length;

return Math.Min(baseConfidence \* lengthFactor \* clarityFactor, 1m);

}

private async Task<decimal> AssessContentRelevanceAsync(string studentAnswer, string expectedAnswer)

{

return CalculateTextSimilarity(studentAnswer, expectedAnswer);

}

private async Task<decimal> AssessGrammarAsync(string text)

{

if (string.IsNullOrEmpty(text)) return 0;

// Basic grammar assessment

var sentences = text.Split('.', '!', '?').Where(s => !string.IsNullOrWhiteSpace(s)).ToArray();

if (sentences.Length == 0) return 0;

var grammarScore = 0.8m; // Base score

// Simple checks

var hasCapitalizedSentences = sentences.Count(s => char.IsUpper(s.Trim().FirstOrDefault())) / (decimal)sentences.Length;

var hasProperPunctuation = (text.Count(c => ".!?".Contains(c)) >= sentences.Length) ? 1m : 0.7m;

return (grammarScore + hasCapitalizedSentences + hasProperPunctuation) / 3;

}

private async Task<decimal> AssessStructureAsync(string text)

{

if (string.IsNullOrEmpty(text)) return 0;

var sentences = text.Split('.', '!', '?').Where(s => !string.IsNullOrWhiteSpace(s)).Count();

var paragraphs = text.Split('\n').Where(p => !string.IsNullOrWhiteSpace(p)).Count();

// Basic structure scoring

var structureScore = 0.7m; // Base score

if (sentences >= 3) structureScore += 0.2m; // Has multiple sentences

if (paragraphs >= 2) structureScore += 0.1m; // Has multiple paragraphs

return Math.Min(structureScore, 1m);

}

private string GenerateEssayFeedback(decimal contentScore, decimal grammarScore, decimal structureScore)

{

var feedback = new List<string>();

if (contentScore >= 0.8m) feedback.Add("Excellent content relevance and understanding");

else if (contentScore >= 0.6m) feedback.Add("Good content but could be more detailed");

else feedback.Add("Content needs improvement - ensure you address all key points");

if (grammarScore >= 0.8m) feedback.Add("Good grammar and language use");

else feedback.Add("Pay attention to grammar, spelling, and sentence structure");

if (structureScore >= 0.8m) feedback.Add("Well-organized response");

else feedback.Add("Work on organizing your thoughts into clear paragraphs");

return string.Join(". ", feedback);

}

private string GenerateOverallFeedback(List<AIScoreBreakdown> breakdowns, decimal overallScore)

{

if (overallScore >= 80) return "Excellent work! You have demonstrated strong understanding.";

if (overallScore >= 70) return "Good work! You're on the right track with room for improvement.";

if (overallScore >= 60) return "Fair performance. Focus on improving weaker areas.";

if (overallScore >= 50) return "Below average performance. Consider reviewing the material.";

return "Needs significant improvement. Please seek additional help.";

}

private (bool needed, string reason) DetermineIfTeacherReviewNeeded(AIAssessmentResult assessment, List<AIScoreBreakdown> breakdowns)

{

// Flag for review if confidence is low

if (assessment.ConfidenceLevel < 0.7m)

return (true, $"Low AI confidence: {assessment.Conf

<p>Phone: {reportCard.School.Phone} | Email: {reportCard.School.Email}</p>

<h2>STUDENT REPORT CARD</h2>

</div>

<div class='student-info'>

<div>

<strong>Student Name:</strong> {reportCard.Student.FullName}<br>

<strong>Student Number:</strong> {reportCard.Student.StudentNumber}<br>

<strong>Class:</strong> {reportCard.Student.ClassName}<br>

<strong>Date of Birth:</strong> {reportCard.Student.DateOfBirth:yyyy-MM-dd}

</div>

<div>

<strong>Term:</strong> {reportCard.Term.Name}<br>

<strong>School Year:</strong> {reportCard.Term.SchoolYear}<br>

<strong>Generated:</strong> {reportCard.GeneratedAt:yyyy-MM-dd}

</div>

</div>

<table class='subjects-table'>

<thead>

<tr>

<th>Subject</th>

<th>Teacher</th>

<th>CA Mark<br>(40%)</th>

<th>Exam Mark<br>(60%)</th>

<th>Total Mark</th>

<th>Grade</th>

<th>Unit</th>

<th>Position</th>

<th>Class Avg</th>

<th>Comments</th>

</tr>

</thead>

<tbody>

{string.Join("", reportCard.SubjectPerformances.Select(sp => $@"

<tr>

<td>{sp.SubjectName}</td>

<td>{sp.TeacherName}</td>

<td>{sp.ContinuousAssessmentMark:F1}</td>

<td>{sp.ExamMark:F1}</td>

<td>{sp.TotalMark:F1}</td>

<td>{sp.Grade}</td>

<td>{sp.Unit?.ToString() ?? "-"}</td>

<td>{sp.SubjectPosition}/{sp.TotalStudentsInSubject}</td>

<td>{sp.ClassAverage:F1}</td>

<td>{sp.Comments}</td>

</tr>"))}

</tbody>

<tfoot>

<tr style='background-color: #e0e0e0; font-weight: bold;'>

<td colspan='2'>TOTALS</td>

<td>-</td>

<td>-</td>

<td>{reportCard.OverallPerformance.OverallPercentage:F1}%</td>

<td>{reportCard.OverallPerformance.OverallGrade}</td>

<td>{reportCard.OverallPerformance.OverallUnit?.ToString() ?? "-"}</td>

<td>{reportCard.OverallPerformance.ClassPosition}/{reportCard.OverallPerformance.TotalStudentsInClass}</td>

<td>{reportCard.OverallPerformance.ClassAverage:F1}</td>

<td>GPA: {reportCard.OverallPerformance.GPA:F2}</td>

</tr>

</tfoot>

</table>

<div class='grading-scale'>

<h3>Grading Scale ({reportCard.GradingScheme.Name})</h3>

<table style='width: 50%; border-collapse: collapse;'>

<tr><th>Grade</th><th>Unit</th><th>Percentage</th><th>Description</th><th>Points</th></tr>

{string.Join("", reportCard.GradingScheme.GradeScales.Select(gs => $@"

<tr>

<td>{gs.Symbol}</td>

<td>{gs.Unit?.ToString() ?? "-"}</td>

<td>{gs.MinPercentage:F0}-{gs.MaxPercentage:F0}%</td>

<td>{gs.Description}</td>

<td>{gs.GradePoint:F1}</td>

</tr>"))}

</table>

</div>

<div class='attendance-summary'>

<h3>Attendance Summary</h3>

<p><strong>Total School Days:</strong> {reportCard.Attendance.TotalSchoolDays}</p>

<p><strong>Days Present:</strong> {reportCard.Attendance.DaysPresent} ({reportCard.Attendance.AttendancePercentage:F1}%)</p>

<p><strong>Days Absent:</strong> {reportCard.Attendance.DaysAbsent}</p>

<p><strong>Days Late:</strong> {reportCard.Attendance.DaysLate}</p>

<p><strong>Attendance Grade:</strong> {reportCard.Attendance.AttendanceGrade}</p>

</div>

<div class='comments'>

<h3>Teacher Comments</h3>

<p>{reportCard.TeacherComments}</p>

<h3>Head Teacher's Comments</h3>

<p>{reportCard.HeadTeacherComments}</p>

</div>

<div class='footer'>

<p>This report was generated on {reportCard.GeneratedAt:yyyy-MM-dd HH:mm}</p>

<p>For inquiries, contact {reportCard.School.Name} at {reportCard.School.Phone}</p>

</div>

</body>

</html>";

}

private string GenerateClassReportHtml(ClassReportSummaryDto classReport)

{

return $@"

<!DOCTYPE html>

<html>

<head>

<title>Class Report - {classReport.ClassName}</title>

<style>

body {{ font-family: Arial, sans-serif; margin: 20px; }}

.header {{ text-align: center; border-bottom: 2px solid #333; padding-bottom: 20px; }}

.summary-table {{ width: 100%; border-collapse: collapse; margin: 20px 0; }}

.summary-table th, .summary-table td {{ border: 1px solid #333; padding: 8px; text-align: center; }}

.summary-table th {{ background-color: #f0f0f0; }}

.statistics {{ display: flex; justify-content: space-around; margin: 30px 0; }}

.stat-box {{ text-align: center; padding: 20px; background-color: #f5f5f5; border-radius: 8px; }}

</style>

</head>

<body>

<div class='header'>

<h1>{classReport.SchoolName}</h1>

<h2>CLASS PERFORMANCE REPORT</h2>

<p><strong>Class:</strong> {classReport.ClassName} | <strong>Term:</strong> {classReport.TermName}</p>

</div>

<div class='statistics'>

<div class='stat-box'>

<h3>Total Students</h3>

<h2>{classReport.Statistics.TotalStudents}</h2>

</div>

<div class='stat-box'>

<h3>Class Average</h3>

<h2>{classReport.Statistics.ClassAverage:F1}%</h2>

</div>

<div class='stat-box'>

<h3>Pass Rate</h3>

<h2>{classReport.Statistics.PassRate:F1}%</h2>

</div>

<div class='stat-box'>

<h3>Highest Score</h3>

<h2>{classReport.Statistics.HighestScore:F1}%</h2>

</div>

</div>

<table class='summary-table'>

<thead>

<tr>

<th>Position</th>

<th>Student Name</th>

<th>Student Number</th>

<th>Overall %</th>

<th>Grade</th>

<th>Attendance %</th>

</tr>

</thead>

<tbody>

{string.Join("", classReport.StudentSummaries.Select(ss => $@"

<tr>

<td>{ss.ClassPosition}</td>

<td>{ss.StudentName}</td>

<td>{ss.StudentNumber}</td>

<td>{ss.OverallPercentage:F1}%</td>

<td>{ss.OverallGrade}</td>

<td>{ss.AttendancePercentage:F1}%</td>

</tr>"))}

</tbody>

</table>

<div class='footer'>

<p><strong>Statistics Summary:</strong></p>

<p>Students Passed: {classReport.Statistics.PassCount} | Students Failed: {classReport.Statistics.FailCount}</p>

<p>Standard Deviation: {classReport.Statistics.StandardDeviation:F2}</p>

<p>Lowest Score: {classReport.Statistics.LowestScore:F1}%</p>

</div>

</body>

</html>";

}

}

// =================== ONLINE TUTORING SERVICE IMPLEMENTATION ===================

public class OnlineTutoringService : IOnlineTutoringService

{

private readonly SchoolManagementContext \_context;

private readonly INotificationService \_notificationService;

private readonly IConfiguration \_configuration;

public OnlineTutoringService(

SchoolManagementContext context,

INotificationService notificationService,

IConfiguration configuration)

{

\_context = context;

\_notificationService = notificationService;

\_configuration = configuration;

}

public async Task<OnlineTutor> RegisterTutorAsync(OnlineTutor tutor)

{

tutor.CreatedAt = DateTime.UtcNow;

tutor.Rating = 0;

tutor.TotalSessions = 0;

\_context.OnlineTutors.Add(tutor);

await \_context.SaveChangesAsync();

return tutor;

}

public async Task<OnlineTutoringSession> BookSessionAsync(TutoringSessionDto sessionDto)

{

var tutor = await \_context.OnlineTutors.FindAsync(sessionDto.TutorId);

var student = await \_context.Students.FindAsync(sessionDto.StudentId);

if (tutor == null || student == null)

throw new ArgumentException("Invalid tutor or student ID");

// Check tutor availability

var isAvailable = await CheckTutorAvailabilityAsync(sessionDto.TutorId, sessionDto.ScheduledDateTime);

if (!isAvailable)

throw new InvalidOperationException("Tutor is not available at the requested time");

var session = new OnlineTutoringSession

{

TutorId = sessionDto.TutorId,

StudentId = sessionDto.StudentId,

SubjectId = sessionDto.SubjectId,

ScheduledDateTime = sessionDto.ScheduledDateTime,

Duration = sessionDto.Duration,

Type = sessionDto.Type,

Status = SessionStatus.Scheduled,

Cost = CalculateSessionCost(tutor.HourlyRate, sessionDto.Duration),

PaymentFrequency = sessionDto.PaymentFrequency,

MeetingUrl = GenerateMeetingUrl()

};

\_context.OnlineTutoringSessions.Add(session);

await \_context.SaveChangesAsync();

// Send notifications

await NotifySessionBookedAsync(session);

return session;

}

public async Task<List<OnlineTutor>> SearchTutorsAsync(int? subjectId, string gradeLevel, decimal? maxRate)

{

var query = \_context.OnlineTutors.AsQueryable();

if (subjectId.HasValue)

{

query = query.Where(t => t.TutorSubjects.Any(ts => ts.SubjectId == subjectId.Value));

}

if (!string.IsNullOrEmpty(gradeLevel))

{

query = query.Where(t => t.TutorSubjects.Any(ts => ts.GradeLevel == gradeLevel));

}

if (maxRate.HasValue)

{

query = query.Where(t => t.HourlyRate <= maxRate.Value);

}

return await query

.Where(t => t.IsVerified)

.Include(t => t.TutorSubjects)

.ThenInclude(ts => ts.Subject)

.OrderByDescending(t => t.Rating)

.ToListAsync();

}

public async Task<List<TutorAvailability>> GetTutorAvailabilityAsync(int tutorId, DateTime date)

{

return await \_context.TutorAvailabilities

.Where(a => a.TutorId == tutorId && a.IsAvailable)

.ToListAsync();

}

public async Task<OnlineTutoringSession> StartSessionAsync(int sessionId)

{

var session = await \_context.OnlineTutoringSessions.FindAsync(sessionId);

if (session == null) return null;

session.Status = SessionStatus.InProgress;

session.ActualStartTime = DateTime.UtcNow;

await \_context.SaveChangesAsync();

return session;

}

public async Task<OnlineTutoringSession> EndSessionAsync(int sessionId, string notes, int? rating, string review)

{

var session = await \_context.OnlineTutoringSessions

.Include(s => s.Tutor)

.FirstOrDefaultAsync(s => s.Id == sessionId);

if (session == null) return null;

session.Status = SessionStatus.Completed;

session.ActualEndTime = DateTime.UtcNow;

session.SessionNotes = notes;

session.Rating = rating;

session.Review = review;

// Update tutor statistics

if (rating.HasValue)

{

await UpdateTutorRatingAsync(session.TutorId, rating.Value);

}

session.Tutor.TotalSessions += 1;

await \_context.SaveChangesAsync();

return session;

}

public async Task<decimal> CalculateTutorEarningsAsync(int tutorId, DateTime startDate, DateTime endDate)

{

var completedSessions = await \_context.OnlineTutoringSessions

.Where(s => s.TutorId == tutorId

&& s.Status == SessionStatus.Completed

&& s.ActualStartTime >= startDate

&& s.ActualStartTime <= endDate)

.ToListAsync();

return completedSessions.Sum(s => s.Cost);

}

private async Task<bool> CheckTutorAvailabilityAsync(int tutorId, DateTime requestedDateTime)

{

var dayOfWeek = requestedDateTime.DayOfWeek;

var timeOfDay = requestedDateTime.TimeOfDay;

var availability = await \_context.TutorAvailabilities

.FirstOrDefaultAsync(a => a.TutorId == tutorId

&& a.DayOfWeek == dayOfWeek

&& a.StartTime <= timeOfDay

&& a.EndTime >= timeOfDay

&& a.IsAvailable);

if (availability == null) return false;

// Check for existing sessions at the same time

var conflictingSessions = await \_context.OnlineTutoringSessions

.AnyAsync(s => s.TutorId == tutorId

&& s.Status != SessionStatus.Cancelled

&& s.ScheduledDateTime.Date == requestedDateTime.Date

&& s.ScheduledDateTime.TimeOfDay == timeOfDay);

return !conflictingSessions;

}

private decimal CalculateSessionCost(decimal hourlyRate, TimeSpan duration)

{

var hours = (decimal)duration.TotalHours;

return hourlyRate \* hours;

}

private string GenerateMeetingUrl()

{

// In a real implementation, this would integrate with video conferencing services

// like Zoom, Google Meet, or Microsoft Teams

var meetingId = Guid.NewGuid().ToString("N")[..12];

return $"https://meet.schoolplatform.com/session/{meetingId}";

}

private async Task NotifySessionBookedAsync(OnlineTutoringSession session)

{

var tutor = await \_context.OnlineTutors.FindAsync(session.TutorId);

var student = await \_context.Students

.Include(s => s.Parents)

.FirstOrDefaultAsync(s => s.Id == session.StudentId);

// Notify tutor

var tutorNotification = new Notification

{

Title = "New Tutoring Session Booked",

Message = $"New session booked with {student.FirstName} {student.LastName} on {session.ScheduledDateTime:yyyy-MM-dd HH:mm}",

Type = NotificationType.TutoringSession,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.Email

};

await \_notificationService.CreateNotificationAsync(tutorNotification);

// Notify parents

foreach (var parent in student.Parents)

{

var parentNotification = new Notification

{

Title = "Tutoring Session Confirmed",

Message = $"Tutoring session confirmed with {tutor.FirstName} {tutor.LastName} on {session.ScheduledDateTime:yyyy-MM-dd HH:mm}",

Type = NotificationType.TutoringSession,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(parentNotification);

}

}

private async Task UpdateTutorRatingAsync(int tutorId, int newRating)

{

var tutor = await \_context.OnlineTutors.FindAsync(tutorId);

if (tutor == null) return;

var allRatings = await \_context.OnlineTutoringSessions

.Where(s => s.TutorId == tutorId && s.Rating.HasValue)

.Select(s => s.Rating.Value)

.ToListAsync();

tutor.Rating = allRatings.Any() ? allRatings.Average() : 0;

await \_context.SaveChangesAsync();

}

}

// =================== NOTIFICATION SERVICE IMPLEMENTATION ===================

public class NotificationService : INotificationService

{

private readonly SchoolManagementContext \_context;

private readonly IConfiguration \_configuration;

private readonly HttpClient \_httpClient;

public NotificationService(

SchoolManagementContext context,

IConfiguration configuration,

HttpClient httpClient)

{

\_context = context;

\_configuration = configuration;

\_httpClient = httpClient;

}

public async Task<Notification> CreateNotificationAsync(Notification notification)

{

notification.IsRead = false;

notification.CreatedAt = DateTime.UtcNow;

\_context.Notifications.Add(notification);

await \_context.SaveChangesAsync();

return notification;

}

public async Task SendNotificationAsync(int notificationId)

{

var notification = await \_context.Notifications

.Include(n => n.Parent)

.Include(n => n.Student)

.Include(n => n.Teacher)

.FirstOrDefaultAsync(n => n.Id == notificationId);

if (notification == null) return;

switch (notification.Channel)

{

case NotificationChannel.Email:

await SendEmailNotificationAsync(notification);

break;

case NotificationChannel.SMS:

await SendSMSNotificationAsync(notification);

break;

case NotificationChannel.WhatsApp:

await SendWhatsAppNotificationInternal(notification);

break;

case NotificationChannel.Push:

await SendPushNotificationAsync(notification);

break;

case NotificationChannel.InApp:

// In-app notifications are stored in database and shown in UI

break;

}

}

public async Task SendBulkNotificationsAsync(List<int> notificationIds)

{

var tasks = notificationIds.Select(SendNotificationAsync);

await Task.WhenAll(tasks);

}

public async Task<List<Notification>> GetParentNotificationsAsync(int parentId)

{

return await \_context.Notifications

.Where(n => n.ParentId == parentId)

.OrderByDescending(n => n.CreatedAt)

.Take(50)

.ToListAsync();

}

public async Task<bool> MarkNotificationAsReadAsync(int notificationId)

{

var notification = await \_context.Notifications.FindAsync(notificationId);

if (notification == null) return false;

notification.IsRead = true;

notification.ReadAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

return true;

}

public async Task SendWhatsAppNotificationAsync(string phoneNumber, string message)

{

// Implementation would use WhatsApp Business API

var whatsappApiUrl = \_configuration["WhatsApp:ApiUrl"];

var accessToken = \_configuration["WhatsApp:AccessToken"];

var payload = new

{

messaging\_product = "whatsapp",

to = phoneNumber,

type = "text",

text = new { body = message }

};

var json = System.Text.Json.JsonSerializer.Serialize(payload);

var content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

\_httpClient.DefaultRequestHeaders.Authorization =

new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", accessToken);

try

{

var response = await \_httpClient.PostAsync(whatsappApiUrl, content);

// Log response for debugging

}

catch (Exception ex)

{

// Log error

}

}

public async Task SendEmailNotificationAsync(string email, string subject, string message)

{

// Implementation would use SendGrid, AWS SES, or similar service

var emailApiUrl = \_configuration["Email:ApiUrl"];

var apiKey = \_configuration["Email:ApiKey"];

var payload = new

{

from = \_configuration["Email:FromAddress"],

to = email,

subject = subject,

html = message

};

var json = System.Text.Json.JsonSerializer.Serialize(payload);

var content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

\_httpClient.DefaultRequestHeaders.Authorization =

new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", apiKey);

try

{

var response = await \_httpClient.PostAsync(emailApiUrl, content);

// Log response for debugging

}

catch (Exception ex)

{

// Log error

}

}

private async Task SendEmailNotificationAsync(Notification notification)

{

string email = "";

if (notification.Parent != null) email = notification.Parent.Email;

else if (notification.Teacher != null) email = notification.Teacher.Email;

if (!string.IsNullOrEmpty(email))

{

await SendEmailNotificationAsync(email, notification.Title, notification.Message);

}

}

private async Task SendSMSNotificationAsync(Notification notification)

{

// Implementation would use Twilio or similar SMS service

string phone = "";

if (notification.Parent != null) phone = notification.Parent.Phone;

else if (notification.Teacher != null) phone = notification.Teacher.Phone;

if (!string.IsNullOrEmpty(phone))

{

// SMS sending logic here

}

}

private async Task SendWhatsAppNotificationInternal(Notification notification)

{

string phone = "";

if (notification.Parent != null) phone = notification.Parent.WhatsAppNumber;

else if (notification.Teacher != null) phone = notification.Teacher.Phone;

if (!string.IsNullOrEmpty(phone))

{

await SendWhatsAppNotificationAsync(phone, notification.Message);

}

}

private async Task SendPushNotificationAsync(Notification notification)

{

// Implementation would use Firebase Cloud Messaging or similar

// Push notification logic here

}

}

// =================== EXAM SERVICE IMPLEMENTATION ===================

public class ExamService : IExamService

{

private readonly SchoolManagementContext \_context;

public ExamService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<Exam> CreateExamAsync(Exam exam)

{

\_context.Exams.Add(exam);

await \_context.SaveChangesAsync();

return exam;

}

public async Task<ExamSubject> AddExamSubjectAsync(ExamSubject examSubject)

{

\_context.ExamSubjects.Add(examSubject);

await \_context.SaveChangesAsync();

return examSubject;

}

public async Task<List<Exam>> GetSchoolExamsAsync(int schoolId, int termId)

{

return await \_context.Exams

.Where(e => e.SchoolId == schoolId && e.TermId == termId)

.Include(e => e.ExamSubjects)

.ThenInclude(es => es.Subject)

.OrderBy(e => e.StartDate)

.ToListAsync();

}

public async Task<byte[]> GenerateExamTimetableAsync(int examId)

{

var exam = await \_context.Exams

.Include(e => e.ExamSubjects)

.ThenInclude(es => es.Subject)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null) return null;

var html = $@"

<!DOCTYPE html>

<html>

<head>

<title>Exam Timetable - {exam.Name}</title>

<style>

body {{ font-family: Arial, sans-serif; margin: 20px; }}

.header {{ text-align: center; margin-bottom: 30px; }}

.timetable {{ width: 100%; border-collapse: collapse; }}

.timetable th, .timetable td {{ border: 1px solid #333; padding: 10px; text-align: left; }}

.timetable th {{ background-color: #f0f0f0; }}

</style>

</head>

<body>

<div class='header'>

<h1>{exam.Name}</h1>

<p>Period: {exam.StartDate:yyyy-MM-dd} to {exam.EndDate:yyyy-MM-dd}</p>

</div>

<table class='timetable'>

<thead>

<tr>

<th>Date</th>

<th>Subject</th>

<th>Duration</th>

<th>Total Marks</th>

<th>Instructions</th>

</tr>

</thead>

<tbody>

{string.Join("", exam.ExamSubjects.OrderBy(es => es.ExamDate).Select(es => $@"

<tr>

<td>{es.ExamDate:yyyy-MM-dd HH:mm}</td>

<td>{es.Subject.Name}</td>

<td>{es.Duration}</td>

<td>{es.TotalMarks}</td>

<td>{es.Instructions}</td>

</tr>"))}

</tbody>

</table>

</body>

</html>";

return System.Text.Encoding.UTF8.GetBytes(html);

}

}

// =================== STARTUP CONFIGURATION ===================

public class Startup

{

public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public void ConfigureServices(IServiceCollection services)

{

// Database

services.AddDbContext<SchoolManagementContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

// HttpClient

services.AddHttpClient();

// Services

services.AddScoped<IGradingService, GradingService>();

services.AddScoped<IStudentService, StudentService>();

services.AddScoped<ITeacherService, TeacherService>();

services.AddScoped<IResultService, ResultService>();

services.AddScoped<IAttendanceService, AttendanceService>();

services.AddScoped<ITimetableService, TimetableService>();

services.AddScoped<INotificationService, NotificationService>();

services.AddScoped<IAssignmentService, AssignmentService>();

services.AddScoped<IHandwritingRecognitionService, HandwritingRecognitionService>();

services.AddScoped<IAIAssessmentService, AIAssessmentService>();

services.AddScoped<IOnlineTutoringService, OnlineTutoringService>();

services.AddScoped<IReportCardService, ReportCardService>();

services.AddScoped<IExamService, ExamService>();

services.AddScoped<ISocialMediaService, SocialMediaService>();

// API Controllers

services.AddControllers();

services.AddEndpointsApiExplorer();

services.AddSwaggerGen();

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

}

// =================== SOCIAL MEDIA SERVICE ===================

public class SocialMediaService : ISocialMediaService

{

private readonly SchoolManagementContext \_context;

private readonly IConfiguration \_configuration;

private readonly HttpClient \_httpClient;

public SocialMediaService(

SchoolManagementContext context,

IConfiguration configuration,

HttpClient httpClient)

{

\_context = context;

\_configuration = configuration;

\_httpClient = httpClient;

}

public async Task<SocialMediaPost> CreatePostAsync(SocialMediaPost post)

{

post.PostedAt = DateTime.UtcNow;

\_context.SocialMediaPosts.Add(post);

await \_context.SaveChangesAsync();

return post;

}

public async Task<bool> PublishToFacebookAsync(int postId)

{

var post = await \_context.SocialMediaPosts.FindAsync(postId);

if (post == null) return false;

// Facebook API integration

var facebookApiUrl = $"https://graph.facebook.com/v18.0/{\_configuration["Facebook:PageId"]}/posts";

var accessToken = \_configuration["Facebook:AccessToken"];

var payload = new

{

message = post.Content,

access\_token = accessToken

};

try

{

var json = System.Text.Json.JsonSerializer.Serialize(payload);

var content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

var response = await \_httpClient.PostAsync(facebookApiUrl, content);

if (response.IsSuccessStatusCode)

{

var responseContent = await response.Content.ReadAsStringAsync();

var responseData = System.Text.Json.JsonSerializer.Deserialize<Dictionary<string, object>>(responseContent);

post.ExternalPostId = responseData.ContainsKey("id") ? responseData["id"].ToString() : "";

await \_context.SaveChangesAsync();

return true;

}

}

catch (Exception ex)

{

// Log error

Console.WriteLine($"Facebook posting error: {ex.Message}");

}

return false;

}

public async Task<bool> PublishToWhatsAppAsync(int postId, List<string> phoneNumbers)

{

var post = await \_context.SocialMediaPosts.FindAsync(postId);

if (post == null) return false;

var whatsappService = new NotificationService(\_context, \_configuration, \_httpClient);

var successCount = 0;

foreach (var phoneNumber in phoneNumbers)

{

try

{

await whatsappService.SendWhatsAppNotificationAsync(phoneNumber, post.Content);

successCount++;

}

catch (Exception ex)

{

Console.WriteLine($"WhatsApp sending error for {phoneNumber}: {ex.Message}");

}

}

return successCount > 0;

}

public async Task<bool> PublishToInstagramAsync(int postId)

{

var post = await \_context.SocialMediaPosts.FindAsync(postId);

if (post == null || string.IsNullOrEmpty(post.ImagePath)) return false;

// Instagram API integration (requires image)

var instagramApiUrl = $"https://graph.facebook.com/v18.0/{\_configuration["Instagram:AccountId"]}/media";

var accessToken = \_configuration["Instagram:AccessToken"];

try

{

// Step 1: Create media object

var mediaPayload = new

{

image\_url = post.ImagePath,

caption = post.Content,

access\_token = accessToken

};

var json = System.Text.Json.JsonSerializer.Serialize(mediaPayload);

var content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

var response = await \_httpClient.PostAsync(instagramApiUrl, content);

if (response.IsSuccessStatusCode)

{

var responseContent = await response.Content.ReadAsStringAsync();

var mediaData = System.Text.Json.JsonSerializer.Deserialize<Dictionary<string, object>>(responseContent);

var mediaId = mediaData["id"].ToString();

// Step 2: Publish media

var publishUrl = $"https://graph.facebook.com/v18.0/{\_configuration["Instagram:AccountId"]}/media\_publish";

var publishPayload = new

{

creation\_id = mediaId,

access\_token = accessToken

};

var publishJson = System.Text.Json.JsonSerializer.Serialize(publishPayload);

var publishContent = new StringContent(publishJson, System.Text.Encoding.UTF8, "application/json");

var publishResponse = await \_httpClient.PostAsync(publishUrl, publishContent);

if (publishResponse.IsSuccessStatusCode)

{

var publishResponseContent = await publishResponse.Content.ReadAsStringAsync();

var publishData = System.Text.Json.JsonSerializer.Deserialize<Dictionary<string, object>>(publishResponseContent);

post.ExternalPostId = publishData.ContainsKey("id") ? publishData["id"].ToString() : "";

await \_context.SaveChangesAsync();

return true;

}

}

}

catch (Exception ex)

{

Console.WriteLine($"Instagram posting error: {ex.Message}");

}

return false;

}

public async Task<List<SocialMediaPost>> GetSchoolPostsAsync(int schoolId)

{

return await \_context.SocialMediaPosts

.Where(p => p.SchoolId == schoolId)

.OrderByDescending(p => p.PostedAt)

.Take(50)

.ToListAsync();

}

}

// =================== TEACHER SERVICE IMPLEMENTATION ===================

public class TeacherService : ITeacherService

{

private readonly SchoolManagementContext \_context;

public TeacherService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<Teacher> CreateTeacherAsync(Teacher teacher)

{

teacher.CreatedAt = DateTime.UtcNow;

teacher.IsActive = true;

teacher.UserType = "Teacher";

teacher.DateOfHire = DateTime.UtcNow;

\_context.Teachers.Add(teacher);

await \_context.SaveChangesAsync();

return teacher;

}

public async Task<List<Teacher>> GetTeachersBySchoolAsync(int schoolId)

{

return await \_context.Teachers

.Where(t => t.SchoolId == schoolId && t.IsActive)

.OrderBy(t => t.LastName)

.ThenBy(t => t.FirstName)

.ToListAsync();

}

public async Task<bool> AssignTeacherToSubjectAsync(int teacherId, int classId, int subjectId)

{

var existingAssignment = await \_context.ClassSubjects

.FirstOrDefaultAsync(cs => cs.ClassId == classId && cs.SubjectId == subjectId);

if (existingAssignment != null)

{

existingAssignment.TeacherId = teacherId;

}

else

{

var newAssignment = new ClassSubject

{

ClassId = classId,

SubjectId = subjectId,

TeacherId = teacherId

};

\_context.ClassSubjects.Add(newAssignment);

}

await \_context.SaveChangesAsync();

return true;

}

public async Task<List<ClassSubject>> GetTeacherAssignmentsAsync(int teacherId)

{

return await \_context.ClassSubjects

.Where(cs => cs.TeacherId == teacherId)

.Include(cs => cs.Class)

.Include(cs => cs.Subject)

.ToListAsync();

}

}

// =================== SAMPLE API CONTROLLERS ===================

[ApiController]

[Route("api/[controller]")]

public class StudentsController : ControllerBase

{

private readonly IStudentService \_studentService;

private readonly IReportCardService \_reportCardService;

public StudentsController(IStudentService studentService, IReportCardService reportCardService)

{

\_studentService = studentService;

\_reportCardService = reportCardService;

}

[HttpPost]

public async Task<ActionResult<Student>> CreateStudent(Student student)

{

var createdStudent = await \_studentService.CreateStudentAsync(student);

return CreatedAtAction(nameof(GetStudent), new { id = createdStudent.Id }, createdStudent);

}

[HttpGet("{id}")]

public async Task<ActionResult<Student>> GetStudent(int id)

{

var student = await \_studentService.GetStudentByNumberAsync(id.ToString(), 1); // Simplified

return student != null ? Ok(student) : NotFound();

}

[HttpGet("class/{classId}")]

public async Task<ActionResult<List<Student>>> GetStudentsByClass(int classId)

{

var students = await \_studentService.GetStudentsByClassAsync(classId);

return Ok(students);

}

[HttpGet("{studentId}/report-card/{termId}")]

public async Task<ActionResult<DetailedReportCardDto>> GetReportCard(int studentId, int termId)

{

var reportCard = await \_reportCardService.GenerateDetailedReportCardAsync(studentId, termId);

return Ok(reportCard);

}

[HttpGet("{studentId}/report-card/{termId}/pdf")]

public async Task<IActionResult> GetReportCardPdf(int studentId, int termId)

{

var pdfBytes = await \_reportCardService.GenerateReportCardPdfAsync(studentId, termId);

return File(pdfBytes, "application/pdf", $"report-card-{studentId}-{termId}.pdf");

}

}

[ApiController]

[Route("api/[controller]")]

public class AssignmentsController : ControllerBase

{

private readonly IAssignmentService \_assignmentService;

private readonly IAIAssessmentService \_aiAssessmentService;

public AssignmentsController(IAssignmentService assignmentService, IAIAssessmentService aiAssessmentService)

{

\_assignmentService = assignmentService;

\_aiAssessmentService = aiAssessmentService;

}

[HttpPost]

public async Task<ActionResult<Assignment>> CreateAssignment(Assignment assignment)

{

var createdAssignment = await \_assignmentService.CreateAssignmentAsync(assignment);

return CreatedAtAction(nameof(GetAssignment), new { id = createdAssignment.Id }, createdAssignment);

}

[HttpGet("{id}")]

public async Task<ActionResult<Assignment>> GetAssignment(int id)

{

var assignment = await \_assignmentService.GetClassAssignmentsAsync(1); // Simplified

var specificAssignment = assignment.FirstOrDefault(a => a.Id == id);

return specificAssignment != null ? Ok(specificAssignment) : NotFound();

}

[HttpPost("submissions")]

public async Task<ActionResult<AssignmentSubmission>> SubmitAssignment(AssignmentSubmission submission)

{

var submittedAssignment = await \_assignmentService.SubmitAssignmentAsync(submission);

// Trigger AI assessment

var aiAssessment = await \_aiAssessmentService.AssessSubmissionAsync(submittedAssignment.Id);

return CreatedAtAction(nameof(GetSubmission), new { id = submittedAssignment.Id }, submittedAssignment);

}

[HttpGet("submissions/{id}")]

public async Task<ActionResult<AssignmentSubmission>> GetSubmission(int id)

{

// Implementation would fetch submission from database

return Ok();

}

[HttpGet("ai-assessments/pending-reviews/{teacherId}")]

public async Task<ActionResult<List<AIAssessmentResult>>> GetPendingReviews(int teacherId)

{

var pendingReviews = await \_aiAssessmentService.GetPendingReviewsAsync(teacherId);

return Ok(pendingReviews);

}

[HttpPost("ai-assessments/{assessmentId}/verify")]

public async Task<ActionResult<AIAssessmentResult>> VerifyAssessment(

int assessmentId,

int teacherId,

TeacherVerificationDto verification)

{

var verifiedAssessment = await \_aiAssessmentService.TeacherVerifyAssessmentAsync(

assessmentId, teacherId, verification);

return Ok(verifiedAssessment);

}

}

[ApiController]

[Route("api/[controller]")]

public class HandwritingController : ControllerBase

{

private readonly IHandwritingRecognitionService \_handwritingService;

public HandwritingController(IHandwritingRecognitionService handwritingService)

{

\_handwritingService = handwritingService;

}

[HttpPost("recognize")]

public async Task<ActionResult<HandwritingSample>> RecognizeHandwriting(HandwritingRecognitionDto dto)

{

var sample = await \_handwritingService.CreateHandwritingSampleAsync(dto);

return Ok(sample);

}

[HttpPost("validate/{sampleId}")]

public async Task<ActionResult<bool>> ValidateSample(int sampleId, string correctedText, int teacherId)

{

var result = await \_handwritingService.ValidateHandwritingSampleAsync(sampleId, correctedText, teacherId);

return Ok(result);

}

[HttpPost("train-model/{studentId}")]

public async Task<ActionResult> TrainModel(int studentId)

{

try

{

await \_handwritingService.TrainPersonalizedModelAsync(studentId);

return Ok(new { message = "Model training started successfully" });

}

catch (InvalidOperationException ex)

{

return BadRequest(new { error = ex.Message });

}

}

[HttpGet("generate-content")]

public async Task<ActionResult<string>> GenerateTrainingContent(HandwritingType type, string difficulty = "beginner")

{

var content = await \_handwritingService.GenerateTrainingContentAsync(type, difficulty);

return Ok(new { content });

}

}

[ApiController]

[Route("api/[controller]")]

public class TutoringController : ControllerBase

{

private readonly IOnlineTutoringService \_tutoringService;

public TutoringController(IOnlineTutoringService tutoringService)

{

\_tutoringService = tutoringService;

}

[HttpPost("tutors")]

public async Task<ActionResult<OnlineTutor>> RegisterTutor(OnlineTutor tutor)

{

var registeredTutor = await \_tutoringService.RegisterTutorAsync(tutor);

return CreatedAtAction(nameof(GetTutor), new { id = registeredTutor.Id }, registeredTutor);

}

[HttpGet("tutors/{id}")]

public async Task<ActionResult<OnlineTutor>> GetTutor(int id)

{

// Implementation would fetch tutor from database

return Ok();

}

[HttpGet("tutors/search")]

public async Task<ActionResult<List<OnlineTutor>>> SearchTutors(

int? subjectId,

string gradeLevel,

decimal? maxRate)

{

var tutors = await \_tutoringService.SearchTutorsAsync(subjectId, gradeLevel, maxRate);

return Ok(tutors);

}

[HttpPost("sessions")]

public async Task<ActionResult<OnlineTutoringSession>> BookSession(TutoringSessionDto sessionDto)

{

try

{

var session = await \_tutoringService.BookSessionAsync(sessionDto);

return CreatedAtAction(nameof(GetSession), new { id = session.Id }, session);

}

catch (Exception ex)

{

return BadRequest(new { error = ex.Message });

}

}

[HttpGet("sessions/{id}")]

public async Task<ActionResult<OnlineTutoringSession>> GetSession(int id)

{

// Implementation would fetch session from database

return Ok();

}

[HttpPut("sessions/{id}/start")]

public async Task<ActionResult<OnlineTutoringSession>> StartSession(int id)

{

var session = await \_tutoringService.StartSessionAsync(id);

return session != null ? Ok(session) : NotFound();

}

[HttpPut("sessions/{id}/end")]

public async Task<ActionResult<OnlineTutoringSession>> EndSession(

int id,

string notes,

int? rating,

string review)

{

var session = await \_tutoringService.EndSessionAsync(id, notes, rating, review);

return session != null ? Ok(session) : NotFound();

}

}

// =================== SAMPLE CONFIGURATION AND USAGE ===================

/\*

// appsettings.json configuration example:

{

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=SchoolManagement;Trusted\_Connection=true;"

},

"WhatsApp": {

"ApiUrl": "https://graph.facebook.com/v18.0/YOUR\_PHONE\_NUMBER\_ID/messages",

"AccessToken": "YOUR\_WHATSAPP\_ACCESS\_TOKEN"

},

"Email": {

"ApiUrl": "https://api.sendgrid.com/v3/mail/send",

"ApiKey": "YOUR\_SENDGRID\_API\_KEY",

"FromAddress": "noreply@yourschool.com"

},

"HandwritingRecognition": {

"CloudApiUrl": "https://your-ml-api.com/recognize"

},

"Facebook": {

"PageId": "YOUR\_FACEBOOK\_PAGE\_ID",

"AccessToken": "YOUR\_FACEBOOK\_ACCESS\_TOKEN"

},

"Instagram": {

"AccountId": "YOUR\_INSTAGRAM\_BUSINESS\_ACCOUNT\_ID",

"AccessToken": "YOUR\_INSTAGRAM\_ACCESS\_TOKEN"

}

}

// Usage examples:

// 1. Create a grading body and scheme

var gradingBody = new GradingBody

{

Name = "Zimbabwe School Examinations Council",

Country = "Zimbabwe",

Description = "National examination board for Zimbabwe"

};

var gradingScheme = new GradingScheme

{

Name = "O-Level Grading",

GradingBodyId = gradingBody.Id

};

var gradeScales = new List<GradeScale>

{

new() { Symbol = "A", Unit = 1, MinPercentage = 80, MaxPercentage = 100, Description = "Excellent", GradePoint = 5.0m },

new() { Symbol = "B", Unit = 2, MinPercentage = 70, MaxPercentage = 79, Description = "Very Good", GradePoint = 4.0m },

new() { Symbol = "C", Unit = 3, MinPercentage = 60, MaxPercentage = 69, Description = "Good", GradePoint = 3.0m },

new() { Symbol = "D", Unit = 4, MinPercentage = 50, MaxPercentage = 59, Description = "Satisfactory", GradePoint = 2.0m },

new() { Symbol = "E", Unit = 5, MinPercentage = 40, MaxPercentage = 49, Description = "Weak", GradePoint = 1.0m },

new() { Symbol = "F", Unit = null, MinPercentage = 0, MaxPercentage = 39, Description = "Fail", GradePoint = 0.0m }

};

// 2. Record student results with AI assessment

var assignment = new Assignment

{

Title = "Mathematics Test 1",

Type = AssignmentType.Quiz,

DueDate = DateTime.Now.AddDays(7),

TotalMarks = 50,

SubjectId = mathSubjectId,

ClassId = form1AClassId,

TeacherId = mathTeacherId,

IsOnlinePlatformWork = true,

Questions = new List<AssignmentQuestion>

{

new()

{

Question = "What is 2 + 2?",

Type = QuestionType.MultipleChoice,

CorrectAnswer = "4",

Marks = 5,

OrderIndex = 1,

Options = new List<QuestionOption>

{

new() { OptionText = "3", OptionLetter = 'A', IsCorrect = false },

new() { OptionText = "4", OptionLetter = 'B', IsCorrect = true },

new() { OptionText = "5", OptionLetter = 'C', IsCorrect = false },

new() { OptionText = "6", OptionLetter = 'D', IsCorrect = false }

}

}

}

};

// 3. Generate comprehensive report card

var reportCard = await reportCardService.GenerateDetailedReportCardAsync(studentId, termId);

// 4. Book online tutoring session

var tutoringSession = new TutoringSessionDto

{

TutorId = 1,

StudentId = 1,

SubjectId = 1,

ScheduledDateTime = DateTime.Now.AddDays(1),

Duration = TimeSpan.FromHours(1),

Type = SessionType.OneOnOne,

PaymentFrequency = PaymentFrequency.OneTime

};

var session = await tutoringService.BookSessionAsync(tutoringSession);

\*/ // Flag for review if confidence is low

if (assessment.ConfidenceLevel < 0.7m)

return (true, $"Low AI confidence: {assessment.ConfidenceLevel:P}");

// Flag for review if handwriting confidence is low

if (assessment.HandwritingConfidence > 0 && assessment.HandwritingConfidence < 0.75m)

return (true, $"Low handwriting recognition confidence: {assessment.HandwritingConfidence:P}");

// Flag for review if score is borderline (around pass/fail boundary)

if (assessment.AIScore >= 45 && assessment.AIScore <= 55)

return (true, "Borderline score requires teacher verification");

// Flag for review if there are significant discrepancies in criterion scores

var criterionScores = breakdowns.Select(b => b.Score / b.MaxScore).ToList();

if (criterionScores.Any() && criterionScores.Max() - criterionScores.Min() > 0.4m)

return (true, "Significant variation in criterion scores");

return (false, "");

}

private async Task NotifyTeacherForReviewAsync(int teacherId, int assessmentResultId)

{

var notification = new Notification

{

Title = "Assignment Review Required",

Message = $"An AI assessment requires your review. Assessment ID: {assessmentResultId}",

Type = NotificationType.Assignment,

TeacherId = teacherId,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

private async Task RecordTeacherCorrectionForLearningAsync(AIAssessmentResult assessment)

{

// This would be used to improve the AI model over time

// Record teacher corrections for machine learning improvement

var correctionData = new

{

AIScore = assessment.AIScore,

TeacherScore = assessment.TeacherScore,

OriginalText = assessment.HandwritingRecognitionText,

QuestionType = assessment.SubmissionAnswer?.Question?.Type,

CorrectionReason = assessment.VerificationStatus,

Timestamp = DateTime.UtcNow

};

// In a real implementation, this would be sent to an ML pipeline

// for model retraining and improvement

}

}

// =================== ENHANCED REPORT CARD SERVICE ===================

public interface IReportCardService

{

Task<DetailedReportCardDto> GenerateDetailedReportCardAsync(int studentId, int termId);

Task<byte[]> GenerateReportCardPdfAsync(int studentId, int termId);

Task<ClassReportSummaryDto> GenerateClassReportSummaryAsync(int classId, int termId);

Task<byte[]> GenerateClassReportPdfAsync(int classId, int termId);

}

public class DetailedReportCardDto

{

public StudentInfoDto Student { get; set; }

public SchoolInfoDto School { get; set; }

public TermInfoDto Term { get; set; }

public List<SubjectPerformanceDto> SubjectPerformances { get; set; } = new();

public OverallPerformanceDto OverallPerformance { get; set; }

public AttendanceSummaryDto Attendance { get; set; }

public string TeacherComments { get; set; }

public string HeadTeacherComments { get; set; }

public DateTime GeneratedAt { get; set; }

public GradingSchemeInfoDto GradingScheme { get; set; }

}

public class StudentInfoDto

{

public string FullName { get; set; }

public string StudentNumber { get; set; }

public string ClassName { get; set; }

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; }

}

public class SchoolInfoDto

{

public string Name { get; set; }

public string Address { get; set; }

public string Phone { get; set; }

public string Email { get; set; }

public string Logo { get; set; }

}

public class TermInfoDto

{

public string Name { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public string SchoolYear { get; set; }

}

public class SubjectPerformanceDto

{

public string SubjectName { get; set; }

public string SubjectCode { get; set; }

public string TeacherName { get; set; }

// Detailed marks breakdown

public List<AssessmentMarkDto> AssessmentMarks { get; set; } = new();

public decimal ContinuousAssessmentMark { get; set; }

public decimal ExamMark { get; set; }

public decimal TotalMark { get; set; }

public decimal Percentage { get; set; }

// Grading information

public string Grade { get; set; }

public int? Unit { get; set; }

public decimal GradePoint { get; set; }

// Performance indicators

public string Comments { get; set; }

public int SubjectPosition { get; set; }

public int TotalStudentsInSubject { get; set; }

public decimal ClassAverage { get; set; }

public string PerformanceTrend { get; set; } // "Improving", "Declining", "Stable"

}

public class AssessmentMarkDto

{

public string AssessmentType { get; set; } // "Test 1", "Assignment", "Project"

public decimal Mark { get; set; }

public decimal MaxMark { get; set; }

public decimal Percentage { get; set; }

public DateTime DateAssessed { get; set; }

}

public class OverallPerformanceDto

{

public decimal TotalMarks { get; set; }

public decimal MaxPossibleMarks { get; set; }

public decimal OverallPercentage { get; set; }

public string OverallGrade { get; set; }

public int? OverallUnit { get; set; }

public decimal GPA { get; set; }

public int ClassPosition { get; set; }

public int TotalStudentsInClass { get; set; }

public decimal ClassAverage { get; set; }

public string PerformanceSummary { get; set; }

}

public class AttendanceSummaryDto

{

public int TotalSchoolDays { get; set; }

public int DaysPresent { get; set; }

public int DaysAbsent { get; set; }

public int DaysLate { get; set; }

public decimal AttendancePercentage { get; set; }

public string AttendanceGrade { get; set; }

}

public class GradingSchemeInfoDto

{

public string Name { get; set; }

public string GradingBody { get; set; }

public List<GradeScaleInfoDto> GradeScales { get; set; } = new();

}

public class GradeScaleInfoDto

{

public string Symbol { get; set; }

public int? Unit { get; set; }

public string Description { get; set; }

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public decimal GradePoint { get; set; }

}

public class ClassReportSummaryDto

{

public string ClassName { get; set; }

public string SchoolName { get; set; }

public string TermName { get; set; }

public List<StudentSummaryDto> StudentSummaries { get; set; } = new();

public ClassStatisticsDto Statistics { get; set; }

}

public class StudentSummaryDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public decimal OverallPercentage { get; set; }

public string OverallGrade { get; set; }

public int ClassPosition { get; set; }

public decimal AttendancePercentage { get; set; }

}

public class ClassStatisticsDto

{

public int TotalStudents { get; set; }

public decimal ClassAverage { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public decimal StandardDeviation { get; set; }

public int PassCount { get; set; }

public int FailCount { get; set; }

public decimal PassRate { get; set; }

}

public class ReportCardService : IReportCardService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

private readonly IAssignmentService \_assignmentService;

public ReportCardService(

SchoolManagementContext context,

IGradingService gradingService,

IAssignmentService assignmentService)

{

\_context = context;

\_gradingService = gradingService;

\_assignmentService = assignmentService;

}

public async Task<DetailedReportCardDto> GenerateDetailedReportCardAsync(int studentId, int termId)

{

var student = await \_context.Students

.Include(s => s.Class)

.ThenInclude(c => c.School)

.ThenInclude(sc => sc.GradingBody)

.FirstOrDefaultAsync(s => s.Id == studentId);

var term = await \_context.Terms

.Include(t => t.SchoolYear)

.FirstOrDefaultAsync(t => t.Id == termId);

// Get all subject results

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Subject)

.ThenInclude(s => s.GradingScheme)

.ThenInclude(gs => gs.GradeScales)

.Include(r => r.Teacher)

.ToListAsync();

// Get continuous assessment marks

var subjectPerformances = new List<SubjectPerformanceDto>();

decimal totalMarks = 0;

decimal maxPossibleMarks = 0;

var allGradePoints = new List<decimal>();

foreach (var result in results)

{

var continuousMarks = await \_assignmentService.CalculateContinuousAssessmentMarkAsync(

studentId, result.SubjectId, termId);

// Get detailed assessment breakdown

var assessmentMarks = await GetAssessmentBreakdownAsync(studentId, result.SubjectId, termId);

// Calculate subject position

var subjectPosition = await CalculateSubjectPositionAsync(studentId, result.SubjectId, termId);

// Get class average for this subject

var classAverage = await CalculateSubjectClassAverageAsync(student.ClassId, result.SubjectId, termId);

// Calculate performance trend

var trend = await CalculatePerformanceTrendAsync(studentId, result.SubjectId, termId);

// Get grade point for GPA calculation

var gradeScale = result.Subject.GradingScheme.GradeScales

.FirstOrDefault(gs => gs.Symbol == result.Grade);

var gradePoint = gradeScale?.GradePoint ?? 0;

allGradePoints.Add(gradePoint);

var subjectPerformance = new SubjectPerformanceDto

{

SubjectName = result.Subject.Name,

SubjectCode = result.Subject.Code,

TeacherName = $"{result.Teacher.FirstName} {result.Teacher.LastName}",

AssessmentMarks = assessmentMarks,

ContinuousAssessmentMark = continuousMarks,

ExamMark = result.Score,

TotalMark = (continuousMarks \* 0.4m) + (result.Score \* 0.6m), // 40% CA, 60% Exam

Percentage = result.Score,

Grade = result.Grade,

Unit = result.Unit,

GradePoint = gradePoint,

Comments = result.Comments,

SubjectPosition = subjectPosition.position,

TotalStudentsInSubject = subjectPosition.totalStudents,

ClassAverage = classAverage,

PerformanceTrend = trend

};

subjectPerformances.Add(subjectPerformance);

totalMarks += subjectPerformance.TotalMark;

maxPossibleMarks += 100; // Assuming 100 is max for each subject

}

// Calculate overall performance

var overallPercentage = maxPossibleMarks > 0 ? (totalMarks / maxPossibleMarks) \* 100 : 0;

var classPosition = await CalculateClassPositionAsync(studentId, termId);

var classAvg = await CalculateClassAverageAsync(student.ClassId, termId);

var gpa = allGradePoints.Any() ? allGradePoints.Average() : 0;

// Get overall grade

var overallGradeInfo = await \_gradingService.CalculateGradeAsync(

overallPercentage,

student.Class.School.GradingBody.GradingSchemes.FirstOrDefault()?.Id ?? 0);

// Get attendance summary

var attendanceSummary = await GetAttendanceSummaryAsync(studentId, term.StartDate, term.EndDate);

// Get teacher comments

var teacherComments = await GetTeacherCommentsAsync(studentId, termId);

var headTeacherComments = await GetHeadTeacherCommentsAsync(studentId, termId);

return new DetailedReportCardDto

{

Student = new StudentInfoDto

{

FullName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

ClassName = student.Class.Name,

DateOfBirth = student.DateOfBirth,

Gender = student.Gender

},

School = new SchoolInfoDto

{

Name = student.Class.School.Name,

Address = student.Class.School.Address,

Phone = student.Class.School.Phone,

Email = student.Class.School.Email,

Logo = student.Class.School.Logo

},

Term = new TermInfoDto

{

Name = term.Name,

StartDate = term.StartDate,

EndDate = term.EndDate,

SchoolYear = term.SchoolYear.Name

},

SubjectPerformances = subjectPerformances,

OverallPerformance = new OverallPerformanceDto

{

TotalMarks = totalMarks,

MaxPossibleMarks = maxPossibleMarks,

OverallPercentage = overallPercentage,

OverallGrade = overallGradeInfo.grade,

OverallUnit = overallGradeInfo.unit,

GPA = gpa,

ClassPosition = classPosition.position,

TotalStudentsInClass = classPosition.totalStudents,

ClassAverage = classAvg,

PerformanceSummary = GeneratePerformanceSummary(overallPercentage, classPosition.position, classPosition.totalStudents)

},

Attendance = attendanceSummary,

TeacherComments = teacherComments,

HeadTeacherComments = headTeacherComments,

GeneratedAt = DateTime.UtcNow,

GradingScheme = new GradingSchemeInfoDto

{

Name = student.Class.School.GradingBody.GradingSchemes.FirstOrDefault()?.Name,

GradingBody = student.Class.School.GradingBody.Name,

GradeScales = student.Class.School.GradingBody.GradingSchemes.FirstOrDefault()?.GradeScales

.Select(gs => new GradeScaleInfoDto

{

Symbol = gs.Symbol,

Unit = gs.Unit,

Description = gs.Description,

MinPercentage = gs.MinPercentage,

MaxPercentage = gs.MaxPercentage,

GradePoint = gs.GradePoint

}).ToList() ?? new List<GradeScaleInfoDto>()

}

};

}

public async Task<byte[]> GenerateReportCardPdfAsync(int studentId, int termId)

{

var reportCard = await GenerateDetailedReportCardAsync(studentId, termId);

// Generate comprehensive PDF report card

var html = GenerateReportCardHtml(reportCard);

// Convert HTML to PDF (using a library like SelectPdf, wkHtmlToPdf, or similar)

// For now, returning the HTML as bytes for demonstration

return System.Text.Encoding.UTF8.GetBytes(html);

}

public async Task<ClassReportSummaryDto> GenerateClassReportSummaryAsync(int classId, int termId)

{

var classEntity = await \_context.Classes

.Include(c => c.School)

.FirstOrDefaultAsync(c => c.Id == classId);

var term = await \_context.Terms.FindAsync(termId);

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.Include(s => s.Results.Where(r => r.TermId == termId))

.ToListAsync();

var studentSummaries = new List<StudentSummaryDto>();

var allPercentages = new List<decimal>();

foreach (var student in students)

{

var studentResults = student.Results.Where(r => r.TermId == termId).ToList();

var overallPercentage = studentResults.Any() ? studentResults.Average(r => r.Score) : 0;

allPercentages.Add(overallPercentage);

var overallGrade = "";

if (studentResults.Any())

{

var firstSubject = studentResults.First().Subject;

var gradeInfo = await \_gradingService.CalculateGradeAsync(overallPercentage, firstSubject.GradingSchemeId);

overallGrade = gradeInfo.grade;

}

var attendancePercentage = await GetStudentAttendancePercentageAsync(student.Id, term.StartDate, term.EndDate);

studentSummaries.Add(new StudentSummaryDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

OverallPercentage = overallPercentage,

OverallGrade = overallGrade,

AttendancePercentage = attendancePercentage

});

}

// Sort by percentage and assign positions

var sortedSummaries = studentSummaries.OrderByDescending(s => s.OverallPercentage).ToList();

for (int i = 0; i < sortedSummaries.Count; i++)

{

sortedSummaries[i].ClassPosition = i + 1;

}

// Calculate class statistics

var statistics = CalculateClassStatistics(allPercentages);

return new ClassReportSummaryDto

{

ClassName = classEntity.Name,

SchoolName = classEntity.School.Name,

TermName = term.Name,

StudentSummaries = sortedSummaries,

Statistics = statistics

};

}

public async Task<byte[]> GenerateClassReportPdfAsync(int classId, int termId)

{

var classReport = await GenerateClassReportSummaryAsync(classId, termId);

var html = GenerateClassReportHtml(classReport);

return System.Text.Encoding.UTF8.GetBytes(html);

}

private async Task<List<AssessmentMarkDto>> GetAssessmentBreakdownAsync(int studentId, int subjectId, int termId)

{

var submissions = await \_context.AssignmentSubmissions

.Where(s => s.StudentId == studentId

&& s.Assignment.SubjectId == subjectId

&& s.Status == SubmissionStatus.Graded)

.Include(s => s.Assignment)

.ToListAsync();

return submissions.Select(s => new AssessmentMarkDto

{

AssessmentType = $"{s.Assignment.Type} - {s.Assignment.Title}",

Mark = s.Score ?? 0,

MaxMark = s.Assignment.TotalMarks,

Percentage = s.Assignment.TotalMarks > 0 ? ((s.Score ?? 0) / s.Assignment.TotalMarks) \* 100 : 0,

DateAssessed = s.SubmittedAt

}).ToList();

}

private async Task<(int position, int totalStudents)> CalculateSubjectPositionAsync(int studentId, int subjectId, int termId)

{

var subjectResults = await \_context.Results

.Where(r => r.SubjectId == subjectId && r.TermId == termId)

.OrderByDescending(r => r.Score)

.Select(r => new { r.StudentId, r.Score })

.ToListAsync();

var position = subjectResults.FindIndex(r => r.StudentId == studentId) + 1;

return (position, subjectResults.Count);

}

private async Task<decimal> CalculateSubjectClassAverageAsync(int classId, int subjectId, int termId)

{

var classResults = await \_context.Results

.Where(r => r.SubjectId == subjectId && r.TermId == termId && r.Student.ClassId == classId)

.AverageAsync(r => (decimal?)r.Score);

return classResults ?? 0;

}

private async Task<string> CalculatePerformanceTrendAsync(int studentId, int subjectId, int termId)

{

// Get previous term's results for comparison

var currentTerm = await \_context.Terms.FindAsync(termId);

var previousTerm = await \_context.Terms

.Where(t => t.SchoolYearId == currentTerm.SchoolYearId && t.EndDate < currentTerm.StartDate)

.OrderByDescending(t => t.EndDate)

.FirstOrDefaultAsync();

if (previousTerm == null) return "New";

var currentResult = await \_context.Results

.FirstOrDefaultAsync(r => r.StudentId == studentId && r.SubjectId == subjectId && r.TermId == termId);

var previousResult = await \_context.Results

.FirstOrDefaultAsync(r => r.StudentId == studentId && r.SubjectId == subjectId && r.TermId == previousTerm.Id);

if (currentResult == null || previousResult == null) return "Insufficient Data";

var difference = currentResult.Score - previousResult.Score;

return difference switch

{

> 5 => "Improving",

< -5 => "Declining",

\_ => "Stable"

};

}

private async Task<(int position, int totalStudents)> CalculateClassPositionAsync(int studentId, int termId)

{

var student = await \_context.Students.FindAsync(studentId);

var classAverages = await \_context.Results

.Where(r => r.TermId == termId && r.Student.ClassId == student.ClassId)

.GroupBy(r => r.StudentId)

.Select(g => new { StudentId = g.Key, Average = g.Average(r => r.Score) })

.OrderByDescending(x => x.Average)

.ToListAsync();

var position = classAverages.FindIndex(x => x.StudentId == studentId) + 1;

return (position, classAverages.Count);

}

private async Task<decimal> CalculateClassAverageAsync(int classId, int termId)

{

var classResults = await \_context.Results

.Where(r => r.TermId == termId && r.Student.ClassId == classId)

.GroupBy(r => r.StudentId)

.Select(g => g.Average(r => r.Score))

.ToListAsync();

return classResults.Any() ? classResults.Average() : 0;

}

private async Task<AttendanceSummaryDto> GetAttendanceSummaryAsync(int studentId, DateTime startDate, DateTime endDate)

{

var attendances = await \_context.Attendances

.Where(a => a.StudentId == studentId && a.Date >= startDate && a.Date <= endDate)

.ToListAsync();

var totalDays = CalculateSchoolDays(startDate, endDate);

var presentDays = attendances.Count(a => a.Status == AttendanceStatus.Present);

var absentDays = attendances.Count(a => a.Status == AttendanceStatus.Absent);

var lateDays = attendances.Count(a => a.Status == AttendanceStatus.Late);

var attendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0;

var attendanceGrade = attendancePercentage switch

{

>= 95 => "Excellent",

>= 90 => "Very Good",

>= 85 => "Good",

>= 80 => "Satisfactory",

>= 75 => "Needs Improvement",

\_ => "Poor"

};

return new AttendanceSummaryDto

{

TotalSchoolDays = totalDays,

DaysPresent = presentDays,

DaysAbsent = absentDays,

DaysLate = lateDays,

AttendancePercentage = attendancePercentage,

AttendanceGrade = attendanceGrade

};

}

private async Task<decimal> GetStudentAttendancePercentageAsync(int studentId, DateTime startDate, DateTime endDate)

{

var attendance = await GetAttendanceSummaryAsync(studentId, startDate, endDate);

return attendance.AttendancePercentage;

}

private int CalculateSchoolDays(DateTime startDate, DateTime endDate)

{

var days = 0;

for (var date = startDate; date <= endDate; date = date.AddDays(1))

{

if (date.DayOfWeek != DayOfWeek.Saturday && date.DayOfWeek != DayOfWeek.Sunday)

days++;

}

return days;

}

private async Task<string> GetTeacherCommentsAsync(int studentId, int termId)

{

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Teacher)

.ToListAsync();

if (!results.Any()) return "No teacher comments available.";

var comments = results

.Where(r => !string.IsNullOrEmpty(r.Comments))

.Select(r => $"{r.Subject.Name}: {r.Comments}")

.ToList();

return comments.Any() ? string.Join(" | ", comments) : "Good progress overall.";

}

private async Task<string> GetHeadTeacherCommentsAsync(int studentId, int termId)

{

// This could be stored in a separate table or calculated based on overall performance

var student = await \_context.Students

.Include(s => s.Results.Where(r => r.TermId == termId))

.FirstOrDefaultAsync(s => s.Id == studentId);

if (student == null || !student.Results.Any()) return "Keep up the good work!";

var overallAverage = student.Results.Average(r => r.Score);

return overallAverage switch

{

>= 80 => "Excellent performance! Continue with this outstanding work.",

>= 70 => "Good work! Keep striving for excellence.",

>= 60 => "Satisfactory progress. Focus on improving weaker areas.",

>= 50 => "You can do better. Please seek additional support.",

\_ => "Significant improvement needed. Parent conference recommended."

};

}

private string GeneratePerformanceSummary(decimal percentage, int position, int totalStudents)

{

var positionText = position switch

{

1 => "1st",

2 => "2nd",

3 => "3rd",

\_ => $"{position}th"

};

var performanceLevel = percentage switch

{

>= 80 => "excellent",

>= 70 => "good",

>= 60 => "satisfactory",

>= 50 => "below average",

\_ => "poor"

};

return $"Achieved {positionText} position out of {totalStudents} students with {performanceLevel} performance ({percentage:F1}%).";

}

private ClassStatisticsDto CalculateClassStatistics(List<decimal> percentages)

{

if (!percentages.Any()) return new ClassStatisticsDto();

var average = percentages.Average();

var variance = percentages.Average(p => Math.Pow((double)(p - average), 2));

var standardDeviation = (decimal)Math.Sqrt(variance);

var passCount = percentages.Count(p => p >= 50); // Assuming 50% is pass mark

var failCount = percentages.Count - passCount;

var passRate = percentages.Count > 0 ? (decimal)passCount / percentages.Count \* 100 : 0;

return new ClassStatisticsDto

{

TotalStudents = percentages.Count,

ClassAverage = average,

HighestScore = percentages.Max(),

LowestScore = percentages.Min(),

StandardDeviation = standardDeviation,

PassCount = passCount,

FailCount = failCount,

PassRate = passRate

};

}

private string GenerateReportCardHtml(DetailedReportCardDto reportCard)

{

return $@"

<!DOCTYPE html>

<html>

<head>

<title>Report Card - {reportCard.Student.FullName}</title>

<style>

body {{ font-family: Arial, sans-serif; margin: 20px; }}

.header {{ text-align: center; border-bottom: 2px solid #333; padding-bottom: 20px; }}

.school-info {{ text-align: center; margin-bottom: 20px; }}

.student-info {{ display: flex; justify-content: space-between; margin: 20px 0; }}

.subjects-table {{ width: 100%; border-collapse: collapse; margin: 20px 0; }}

.subjects-table th, .subjects-table td {{ border: 1px solid #333; padding: 8px; text-align: center; }}

.subjects-table th {{ background-color: #f0f0f0; }}

.grading-scale {{ margin-top: 20px; }}

.comments {{ margin: 20px 0; padding: 15px; background-color: #f9f9f9; }}

.footer {{ text-align: center; margin-top: 30px; font-size: 12px; }}

</style>

</head>

<body>

<div class='header'>

<h1>{reportCard.School.Name}</h1>

<p>{reportCard.School.Address}</p>

<p>Phone: {reportCard.// =================== MODELS/ENTITIES ===================

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

// Core System Entities

public class GradingBody

{

public int Id { get; set; }

public string Name { get; set; } // e.g., "Zimbabwe Education Board", "Cambridge"

public string Country { get; set; }

public string Description { get; set; }

public DateTime CreatedAt { get; set; }

public List<GradingScheme> GradingSchemes { get; set; } = new();

public List<School> Schools { get; set; } = new();

}

public class GradingScheme

{

public int Id { get; set; }

public string Name { get; set; } // e.g., "Primary Grading", "Secondary Grading"

public int GradingBodyId { get; set; }

public GradingBody GradingBody { get; set; }

public List<GradeScale> GradeScales { get; set; } = new();

public List<Subject> Subjects { get; set; } = new();

}

public class GradeScale

{

public int Id { get; set; }

public int GradingSchemeId { get; set; }

public GradingScheme GradingScheme { get; set; }

public string Symbol { get; set; } // A, B, C, D, F

public int? Unit { get; set; } // 1, 2, 3, 4, 5 (nullable for systems that don't use units)

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public string Description { get; set; } // "Excellent", "Good", "Average"

public decimal GradePoint { get; set; } // For GPA calculations

}

public class School

{

public int Id { get; set; }

public string Name { get; set; }

public string Address { get; set; }

public string Phone { get; set; }

public string Email { get; set; }

public string Logo { get; set; }

public int GradingBodyId { get; set; }

public GradingBody GradingBody { get; set; }

public bool IsOnlineSchoolingEnabled { get; set; }

public DateTime CreatedAt { get; set; }

// Navigation properties

public List<Class> Classes { get; set; } = new();

public List<Teacher> Teachers { get; set; } = new();

public List<Student> Students { get; set; } = new();

public List<Subject> Subjects { get; set; } = new();

public List<SchoolYear> SchoolYears { get; set; } = new();

public List<Exam> Exams { get; set; } = new();

public List<OnlineTutor> OnlineTutors { get; set; } = new();

}

public class SchoolYear

{

public int Id { get; set; }

public string Name { get; set; } // "2024/2025"

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<Term> Terms { get; set; } = new();

}

public class Term

{

public int Id { get; set; }

public string Name { get; set; } // "Term 1", "First Semester"

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public int SchoolYearId { get; set; }

public SchoolYear SchoolYear { get; set; }

public List<Result> Results { get; set; } = new();

public List<Exam> Exams { get; set; } = new();

}

public class Class

{

public int Id { get; set; }

public string Name { get; set; } // "Grade 1A", "Form 4B"

public string Level { get; set; } // "Primary", "Secondary"

public int Capacity { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int? ClassTeacherId { get; set; } // Head teacher

public Teacher ClassTeacher { get; set; }

public List<Student> Students { get; set; } = new();

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Timetable> Timetables { get; set; } = new();

public List<Attendance> Attendances { get; set; } = new();

}

public class Subject

{

public int Id { get; set; }

public string Name { get; set; }

public string Code { get; set; } // "MATH101", "ENG101"

public string Description { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int GradingSchemeId { get; set; }

public GradingScheme GradingScheme { get; set; }

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Result> Results { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<OnlineTutorSubject> OnlineTutorSubjects { get; set; } = new();

}

public class ClassSubject

{

public int Id { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public List<TimetableSlot> TimetableSlots { get; set; } = new();

}

// User Management

public abstract class User

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public string Address { get; set; }

public string ProfilePicture { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? LastLogin { get; set; }

public bool IsActive { get; set; }

public string UserType { get; set; } // Discriminator for inheritance

}

public class Teacher : User

{

public string EmployeeId { get; set; }

public string Qualification { get; set; }

public DateTime DateOfHire { get; set; }

public decimal Salary { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<ClassSubject> ClassSubjects { get; set; } = new();

public List<Class> ManagedClasses { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<HandwritingValidation> HandwritingValidations { get; set; } = new();

}

public class Student : User

{

public string StudentNumber { get; set; }

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; }

public DateTime EnrollmentDate { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public List<Parent> Parents { get; set; } = new();

public List<Result> Results { get; set; } = new();

public List<Attendance> Attendances { get; set; } = new();

public List<Assignment> Assignments { get; set; } = new();

public List<HandwritingSample> HandwritingSamples { get; set; } = new();

public List<OnlineTutoringSession> TutoringSessionsAsStudent { get; set; } = new();

public List<StudentTimetable> StudentTimetables { get; set; } = new();

}

public class Parent : User

{

public string Relationship { get; set; } // Father, Mother, Guardian

public string Occupation { get; set; }

public string WhatsAppNumber { get; set; }

public bool ReceiveNotifications { get; set; }

public bool ReceiveWhatsAppNotifications { get; set; }

public bool ReceiveEmailNotifications { get; set; }

public bool ReceiveSMSNotifications { get; set; }

public List<Student> Children { get; set; } = new();

public List<Notification> Notifications { get; set; } = new();

}

// Assessment & Results

public class Result

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int TermId { get; set; }

public Term Term { get; set; }

public decimal Score { get; set; }

public string Grade { get; set; } // Calculated from GradeScale

public int? Unit { get; set; } // Calculated from GradeScale

public string Comments { get; set; }

public DateTime DateRecorded { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public string AssessmentType { get; set; } // "Continuous", "Exam", "Assignment"

}

public class Exam

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public int TermId { get; set; }

public Term Term { get; set; }

public List<ExamSubject> ExamSubjects { get; set; } = new();

}

public class ExamSubject

{

public int Id { get; set; }

public int ExamId { get; set; }

public Exam Exam { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public DateTime ExamDate { get; set; }

public TimeSpan Duration { get; set; }

public decimal TotalMarks { get; set; }

public string Instructions { get; set; }

}

// Attendance

public class Attendance

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

public string Remarks { get; set; }

public int RecordedByTeacherId { get; set; }

public Teacher RecordedByTeacher { get; set; }

}

public enum AttendanceStatus

{

Present,

Absent,

Late,

Excused

}

// Assignment & Homework System

public class Assignment

{

public int Id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public AssignmentType Type { get; set; }

public DateTime DueDate { get; set; }

public decimal TotalMarks { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public bool IsOnlinePlatformWork { get; set; }

public string Instructions { get; set; }

public DateTime CreatedAt { get; set; }

public List<AssignmentSubmission> Submissions { get; set; } = new();

public List<AssignmentQuestion> Questions { get; set; } = new();

}

public enum AssignmentType

{

Homework,

Classwork,

Assignment,

Project,

Quiz

}

public class AssignmentQuestion

{

public int Id { get; set; }

public int AssignmentId { get; set; }

public Assignment Assignment { get; set; }

public string Question { get; set; }

public QuestionType Type { get; set; }

public string CorrectAnswer { get; set; }

public decimal Marks { get; set; }

public int OrderIndex { get; set; }

public List<QuestionOption> Options { get; set; } = new(); // For multiple choice

}

public enum QuestionType

{

MultipleChoice,

TrueFalse,

ShortAnswer,

Essay,

Handwritten

}

public class QuestionOption

{

public int Id { get; set; }

public int QuestionId { get; set; }

public AssignmentQuestion Question { get; set; }

public string OptionText { get; set; }

public bool IsCorrect { get; set; }

public char OptionLetter { get; set; } // A, B, C, D

}

public class AssignmentSubmission

{

public int Id { get; set; }

public int AssignmentId { get; set; }

public Assignment Assignment { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public DateTime SubmittedAt { get; set; }

public decimal? Score { get; set; }

public string Feedback { get; set; }

public SubmissionStatus Status { get; set; }

public bool IsAutoGraded { get; set; }

public List<SubmissionAnswer> Answers { get; set; } = new();

}

public enum SubmissionStatus

{

Submitted,

Graded,

Late,

Missing

}

public class SubmissionAnswer

{

public int Id { get; set; }

public int SubmissionId { get; set; }

public AssignmentSubmission Submission { get; set; }

public int QuestionId { get; set; }

public AssignmentQuestion Question { get; set; }

public string Answer { get; set; }

public string HandwrittenImagePath { get; set; } // For handwritten answers

public decimal? Score { get; set; }

public bool IsCorrect { get; set; }

public string ProcessedText { get; set; } // AI-processed handwritten text

public decimal Confidence { get; set; } // AI confidence level

}

// Timetable System

public class Timetable

{

public int Id { get; set; }

public string Name { get; set; }

public int ClassId { get; set; }

public Class Class { get; set; }

public int SchoolYearId { get; set; }

public SchoolYear SchoolYear { get; set; }

public bool IsActive { get; set; }

public DateTime CreatedAt { get; set; }

public List<TimetableSlot> Slots { get; set; } = new();

}

public class TimetableSlot

{

public int Id { get; set; }

public int TimetableId { get; set; }

public Timetable Timetable { get; set; }

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public int ClassSubjectId { get; set; }

public ClassSubject ClassSubject { get; set; }

public string Room { get; set; }

public SlotType Type { get; set; }

}

public enum SlotType

{

Regular,

Break,

Lunch,

Assembly,

Sports

}

public class StudentTimetable

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int TimetableSlotId { get; set; }

public TimetableSlot TimetableSlot { get; set; }

public bool IsOptional { get; set; } // For elective subjects

}

// Handwriting Recognition System

public class HandwritingSample

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public string ImagePath { get; set; }

public string ExpectedText { get; set; }

public string RecognizedText { get; set; }

public decimal Confidence { get; set; }

public HandwritingType Type { get; set; }

public bool IsValidated { get; set; }

public int? ValidatedByTeacherId { get; set; }

public Teacher ValidatedByTeacher { get; set; }

public DateTime CreatedAt { get; set; }

public bool IsTrainingData { get; set; }

}

public enum HandwritingType

{

Alphabet,

Number,

Word,

Sentence,

Answer

}

public class HandwritingValidation

{

public int Id { get; set; }

public int HandwritingSampleId { get; set; }

public HandwritingSample HandwritingSample { get; set; }

public int TeacherId { get; set; }

public Teacher Teacher { get; set; }

public string CorrectedText { get; set; }

public DateTime ValidatedAt { get; set; }

public ValidationStatus Status { get; set; }

}

public enum ValidationStatus

{

Pending,

Approved,

Corrected,

Rejected

}

public class PersonalizedModel

{

public int Id { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public string ModelPath { get; set; } // Local device path

public string CloudModelPath { get; set; } // Cloud backup path

public DateTime LastTrainingDate { get; set; }

public int SampleCount { get; set; }

public decimal Accuracy { get; set; }

public bool IsDeployedLocally { get; set; }

public bool IsDeployedOnCloud { get; set; }

}

// Online Tutoring System

public class OnlineTutor

{

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public string Bio { get; set; }

public string Qualifications { get; set; }

public decimal HourlyRate { get; set; }

public bool IsVerified { get; set; }

public decimal Rating { get; set; }

public int TotalSessions { get; set; }

public DateTime CreatedAt { get; set; }

public int? SchoolId { get; set; } // Optional - tutor might be independent

public School School { get; set; }

public List<OnlineTutorSubject> TutorSubjects { get; set; } = new();

public List<TutorAvailability> Availability { get; set; } = new();

public List<OnlineTutoringSession> Sessions { get; set; } = new();

}

public class OnlineTutorSubject

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public int SubjectId { get; set; }

public Subject Subject { get; set; }

public string GradeLevel { get; set; } // "Primary", "Secondary", "A-Level"

}

public class TutorAvailability

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public bool IsAvailable { get; set; }

}

public class OnlineTutoringSession

{

public int Id { get; set; }

public int TutorId { get; set; }

public OnlineTutor Tutor { get; set; }

public int StudentId { get; set; }

public Student Student { get; set; }

public int? SubjectId { get; set; }

public Subject Subject { get; set; }

public DateTime ScheduledDateTime { get; set; }

public TimeSpan Duration { get; set; }

public SessionType Type { get; set; }

public SessionStatus Status { get; set; }

public decimal Cost { get; set; }

public PaymentFrequency PaymentFrequency { get; set; }

public string MeetingUrl { get; set; }

public string SessionNotes { get; set; }

public int? Rating { get; set; }

public string Review { get; set; }

public DateTime? ActualStartTime { get; set; }

public DateTime? ActualEndTime { get; set; }

}

public enum SessionType

{

OneOnOne,

GroupSession

}

public enum SessionStatus

{

Scheduled,

InProgress,

Completed,

Cancelled,

NoShow

}

public enum PaymentFrequency

{

OneTime,

Weekly,

Monthly

}

// Notifications & Communication

public class Notification

{

public int Id { get; set; }

public string Title { get; set; }

public string Message { get; set; }

public NotificationType Type { get; set; }

public int? ParentId { get; set; }

public Parent Parent { get; set; }

public int? StudentId { get; set; }

public Student Student { get; set; }

public int? TeacherId { get; set; }

public Teacher Teacher { get; set; }

public bool IsRead { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? ReadAt { get; set; }

public NotificationChannel Channel { get; set; }

public string ExternalId { get; set; } // For WhatsApp/SMS tracking

}

public enum NotificationType

{

Attendance,

Results,

Assignment,

Exam,

GeneralInfo,

Payment,

Disciplinary,

TutoringSession

}

public enum NotificationChannel

{

InApp,

Email,

SMS,

WhatsApp,

Push

}

public class SocialMediaPost

{

public int Id { get; set; }

public string Content { get; set; }

public string ImagePath { get; set; }

public SocialMediaPlatform Platform { get; set; }

public int SchoolId { get; set; }

public School School { get; set; }

public DateTime PostedAt { get; set; }

public string ExternalPostId { get; set; }

public int Likes { get; set; }

public int Shares { get; set; }

public int Comments { get; set; }

}

public enum SocialMediaPlatform

{

Facebook,

Twitter,

Instagram,

WhatsApp,

LinkedIn

}

// =================== DBCONTEXT ===================

using Microsoft.EntityFrameworkCore;

public class SchoolManagementContext : DbContext

{

public SchoolManagementContext(DbContextOptions<SchoolManagementContext> options) : base(options) { }

// Core Entities

public DbSet<GradingBody> GradingBodies { get; set; }

public DbSet<GradingScheme> GradingSchemes { get; set; }

public DbSet<GradeScale> GradeScales { get; set; }

public DbSet<School> Schools { get; set; }

public DbSet<SchoolYear> SchoolYears { get; set; }

public DbSet<Term> Terms { get; set; }

public DbSet<Class> Classes { get; set; }

public DbSet<Subject> Subjects { get; set; }

public DbSet<ClassSubject> ClassSubjects { get; set; }

// Users

public DbSet<User> Users { get; set; }

public DbSet<Teacher> Teachers { get; set; }

public DbSet<Student> Students { get; set; }

public DbSet<Parent> Parents { get; set; }

// Assessment

public DbSet<Result> Results { get; set; }

public DbSet<Exam> Exams { get; set; }

public DbSet<ExamSubject> ExamSubjects { get; set; }

public DbSet<Assignment> Assignments { get; set; }

public DbSet<AssignmentQuestion> AssignmentQuestions { get; set; }

public DbSet<QuestionOption> QuestionOptions { get; set; }

public DbSet<AssignmentSubmission> AssignmentSubmissions { get; set; }

public DbSet<SubmissionAnswer> SubmissionAnswers { get; set; }

// Attendance & Timetable

public DbSet<Attendance> Attendances { get; set; }

public DbSet<Timetable> Timetables { get; set; }

public DbSet<TimetableSlot> TimetableSlots { get; set; }

public DbSet<StudentTimetable> StudentTimetables { get; set; }

// Handwriting Recognition

public DbSet<HandwritingSample> HandwritingSamples { get; set; }

public DbSet<HandwritingValidation> HandwritingValidations { get; set; }

public DbSet<PersonalizedModel> PersonalizedModels { get; set; }

// Online Tutoring

public DbSet<OnlineTutor> OnlineTutors { get; set; }

public DbSet<OnlineTutorSubject> OnlineTutorSubjects { get; set; }

public DbSet<TutorAvailability> TutorAvailabilities { get; set; }

public DbSet<OnlineTutoringSession> OnlineTutoringSessions { get; set; }

// Communication

public DbSet<Notification> Notifications { get; set; }

public DbSet<SocialMediaPost> SocialMediaPosts { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

// User inheritance configuration

modelBuilder.Entity<User>()

.HasDiscriminator<string>("UserType")

.HasValue<Teacher>("Teacher")

.HasValue<Student>("Student")

.HasValue<Parent>("Parent");

// Configure relationships

ConfigureGradingSystem(modelBuilder);

ConfigureSchoolStructure(modelBuilder);

ConfigureUserRelationships(modelBuilder);

ConfigureAssessmentSystem(modelBuilder);

ConfigureTimetableSystem(modelBuilder);

ConfigureHandwritingSystem(modelBuilder);

ConfigureTutoringSystem(modelBuilder);

ConfigureCommunicationSystem(modelBuilder);

ConfigureIndexes(modelBuilder);

ConfigureConstraints(modelBuilder);

}

private void ConfigureGradingSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeScale>()

.HasOne(gs => gs.GradingScheme)

.WithMany(gs => gs.GradeScales)

.HasForeignKey(gs => gs.GradingSchemeId);

modelBuilder.Entity<GradingScheme>()

.HasOne(gs => gs.GradingBody)

.WithMany(gb => gb.GradingSchemes)

.HasForeignKey(gs => gs.GradingBodyId);

}

private void ConfigureSchoolStructure(ModelBuilder modelBuilder)

{

modelBuilder.Entity<School>()

.HasOne(s => s.GradingBody)

.WithMany(gb => gb.Schools)

.HasForeignKey(s => s.GradingBodyId);

modelBuilder.Entity<Class>()

.HasOne(c => c.ClassTeacher)

.WithMany(t => t.ManagedClasses)

.HasForeignKey(c => c.ClassTeacherId)

.OnDelete(DeleteBehavior.SetNull);

modelBuilder.Entity<ClassSubject>()

.HasKey(cs => cs.Id);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Class)

.WithMany(c => c.ClassSubjects)

.HasForeignKey(cs => cs.ClassId);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Subject)

.WithMany(s => s.ClassSubjects)

.HasForeignKey(cs => cs.SubjectId);

modelBuilder.Entity<ClassSubject>()

.HasOne(cs => cs.Teacher)

.WithMany(t => t.ClassSubjects)

.HasForeignKey(cs => cs.TeacherId);

}

private void ConfigureUserRelationships(ModelBuilder modelBuilder)

{

// Student-Parent many-to-many

modelBuilder.Entity<Student>()

.HasMany(s => s.Parents)

.WithMany(p => p.Children)

.UsingEntity<Dictionary<string, object>>(

"StudentParent",

j => j.HasOne<Parent>().WithMany().HasForeignKey("ParentId"),

j => j.HasOne<Student>().WithMany().HasForeignKey("StudentId"));

}

private void ConfigureAssessmentSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Result>()

.Property(r => r.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<AssignmentQuestion>()

.Property(aq => aq.Marks)

.HasPrecision(5, 2);

modelBuilder.Entity<SubmissionAnswer>()

.Property(sa => sa.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<SubmissionAnswer>()

.Property(sa => sa.Confidence)

.HasPrecision(5, 4);

}

private void ConfigureTimetableSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<StudentTimetable>()

.HasKey(st => new { st.StudentId, st.TimetableSlotId });

}

private void ConfigureHandwritingSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<HandwritingSample>()

.Property(hs => hs.Confidence)

.HasPrecision(5, 4);

modelBuilder.Entity<PersonalizedModel>()

.Property(pm => pm.Accuracy)

.HasPrecision(5, 4);

}

private void ConfigureTutoringSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<OnlineTutorSubject>()

.HasKey(ots => new { ots.TutorId, ots.SubjectId });

modelBuilder.Entity<OnlineTutoringSession>()

.Property(ots => ots.Cost)

.HasPrecision(10, 2);

modelBuilder.Entity<OnlineTutor>()

.Property(ot => ot.HourlyRate)

.HasPrecision(10, 2);

modelBuilder.Entity<OnlineTutor>()

.Property(ot => ot.Rating)

.HasPrecision(3, 2);

}

private void ConfigureCommunicationSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Notification>()

.HasOne(n => n.Parent)

.WithMany(p => p.Notifications)

.HasForeignKey(n => n.ParentId)

.OnDelete(DeleteBehavior.Cascade);

}

private void ConfigureIndexes(ModelBuilder modelBuilder)

{

// Performance indexes

modelBuilder.Entity<Student>()

.HasIndex(s => s.StudentNumber)

.IsUnique();

modelBuilder.Entity<Teacher>()

.HasIndex(t => t.EmployeeId)

.IsUnique();

modelBuilder.Entity<User>()

.HasIndex(u => u.Email)

.IsUnique();

modelBuilder.Entity<Attendance>()

.HasIndex(a => new { a.StudentId, a.Date });

modelBuilder.Entity<r>()

.HasIndex(r => new { r.StudentId, r.SubjectId, r.TermId });

}

private void ConfigureConstraints(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeScale>()

.HasCheckConstraint("CK\_GradeScale\_Percentage",

"MinPercentage >= 0 AND MaxPercentage <= 100 AND MinPercentage <= MaxPercentage");

modelBuilder.Entity<TimetableSlot>()

.HasCheckConstraint("CK\_TimetableSlot\_Time", "StartTime < EndTime");

}

}

// =================== DTOS ===================

public class StudentResultDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public string ClassName { get; set; }

public string TermName { get; set; }

public List<SubjectResultDto> SubjectResults { get; set; } = new();

public decimal OverallAverage { get; set; }

public string OverallGrade { get; set; }

public int? OverallUnit { get; set; }

public int Position { get; set; }

public int TotalStudents { get; set; }

}

public class SubjectResultDto

{

public string SubjectName { get; set; }

public decimal Score { get; set; }

public string Grade { get; set; }

public int? Unit { get; set; }

public string Comments { get; set; }

public string TeacherName { get; set; }

}

public class TimetableDto

{

public int Id { get; set; }

public string ClassName { get; set; }

public List<TimetableSlotDto> Slots { get; set; } = new();

}

public class TimetableSlotDto

{

public DayOfWeek DayOfWeek { get; set; }

public string StartTime { get; set; }

public string EndTime { get; set; }

public string SubjectName { get; set; }

public string TeacherName { get; set; }

public string Room { get; set; }

public SlotType Type { get; set; }

}

public class AttendanceReportDto

{

public string StudentName { get; set; }

public string StudentNumber { get; set; }

public int TotalDays { get; set; }

public int PresentDays { get; set; }

public int AbsentDays { get; set; }

public int LateDays { get; set; }

public decimal AttendancePercentage { get; set; }

}

public class HandwritingRecognitionDto

{

public string ImageBase64 { get; set; }

public int StudentId { get; set; }

public HandwritingType Type { get; set; }

public string ExpectedText { get; set; }

}

public class TutoringSessionDto

{

public int TutorId { get; set; }

public int StudentId { get; set; }

public int? SubjectId { get; set; }

public DateTime ScheduledDateTime { get; set; }

public TimeSpan Duration { get; set; }

public SessionType Type { get; set; }

public PaymentFrequency PaymentFrequency { get; set; }

}

// =================== SERVICE INTERFACES ===================

public interface IGradingService

{

Task<GradingBody> CreateGradingBodyAsync(GradingBody gradingBody);

Task<GradingScheme> CreateGradingSchemeAsync(GradingScheme gradingScheme);

Task<GradeScale> AddGradeScaleAsync(GradeScale gradeScale);

Task<(string grade, int? unit)> CalculateGradeAsync(decimal score, int gradingSchemeId);

Task<List<GradingBody>> GetAllGradingBodiesAsync();

Task<List<GradingScheme>> GetGradingSchemesByBodyAsync(int gradingBodyId);

}

public interface IStudentService

{

Task<Student> CreateStudentAsync(Student student);

Task<Student> UpdateStudentAsync(Student student);

Task<List<Student>> GetStudentsByClassAsync(int classId);

Task<List<Student>> GetStudentsBySchoolAsync(int schoolId);

Task<Student> GetStudentByNumberAsync(string studentNumber, int schoolId);

Task<bool> AssignStudentToClassAsync(int studentId, int classId);

Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId);

Task<byte[]> GenerateStudentReportCardAsync(int studentId, int termId);

}

public interface ITeacherService

{

Task<Teacher> CreateTeacherAsync(Teacher teacher);

Task<List<Teacher>> GetTeachersBySchoolAsync(int schoolId);

Task<bool> AssignTeacherToSubjectAsync(int teacherId, int classId, int subjectId);

Task<List<ClassSubject>> GetTeacherAssignmentsAsync(int teacherId);

}

public interface IResultService

{

Task<r> RecordResultAsync(r result);

Task<List<r>> BulkRecordResultsAsync(List<r> results);

Task<List<StudentResultDto>> GetClassResultsAsync(int classId, int termId);

Task<byte[]> GenerateClassReportAsync(int classId, int termId);

Task<StudentResultDto> CalculateStudentPositionAsync(int studentId, int termId);

}

public interface IAttendanceService

{

Task<Attendance> RecordAttendanceAsync(Attendance attendance);

Task<List<Attendance>> BulkRecordAttendanceAsync(List<Attendance> attendances);

Task<List<AttendanceReportDto>> GetClassAttendanceReportAsync(int classId, DateTime startDate, DateTime endDate);

Task<AttendanceReportDto> GetStudentAttendanceReportAsync(int studentId, DateTime startDate, DateTime endDate);

}

public interface ITimetableService

{

Task<Timetable> GenerateTimetableAsync(int classId, int schoolYearId);

Task<TimetableDto> GetClassTimetableAsync(int classId);

Task<TimetableDto> GetStudentTimetableAsync(int studentId);

Task<TimetableSlot> UpdateTimetableSlotAsync(TimetableSlot slot);

Task<bool> ValidateTimetableRulesAsync(int timetableId);

Task<List<TimetableSlot>> GetTeacherScheduleAsync(int teacherId, DateTime date);

}

public interface INotificationService

{

Task<Notification> CreateNotificationAsync(Notification notification);

Task SendNotificationAsync(int notificationId);

Task SendBulkNotificationsAsync(List<int> notificationIds);

Task<List<Notification>> GetParentNotificationsAsync(int parentId);

Task<bool> MarkNotificationAsReadAsync(int notificationId);

Task SendWhatsAppNotificationAsync(string phoneNumber, string message);

Task SendEmailNotificationAsync(string email, string subject, string message);

}

public interface IAssignmentService

{

Task<Assignment> CreateAssignmentAsync(Assignment assignment);

Task<AssignmentSubmission> SubmitAssignmentAsync(AssignmentSubmission submission);

Task<AssignmentSubmission> AutoGradeAssignmentAsync(int submissionId);

Task<List<Assignment>> GetClassAssignmentsAsync(int classId);

Task<List<Assignment>> GetStudentAssignmentsAsync(int studentId);

Task<decimal> CalculateContinuousAssessmentMarkAsync(int studentId, int subjectId, int termId);

}

public interface IHandwritingRecognitionService

{

Task<string> RecognizeHandwritingAsync(string imagePath, int studentId);

Task<HandwritingSample> CreateHandwritingSampleAsync(HandwritingRecognitionDto dto);

Task<bool> ValidateHandwritingSampleAsync(int sampleId, string correctedText, int teacherId);

Task TrainPersonalizedModelAsync(int studentId);

Task<PersonalizedModel> DeployModelLocallyAsync(int studentId);

Task<string> GenerateTrainingContentAsync(HandwritingType type, string difficulty = "beginner");

Task<bool> ProcessHandwritingOfflineAsync(string imagePath, int studentId);

}

public interface IOnlineTutoringService

{

Task<OnlineTutor> RegisterTutorAsync(OnlineTutor tutor);

Task<OnlineTutoringSession> BookSessionAsync(TutoringSessionDto sessionDto);

Task<List<OnlineTutor>> SearchTutorsAsync(int? subjectId, string gradeLevel, decimal? maxRate);

Task<List<TutorAvailability>> GetTutorAvailabilityAsync(int tutorId, DateTime date);

Task<OnlineTutoringSession> StartSessionAsync(int sessionId);

Task<OnlineTutoringSession> EndSessionAsync(int sessionId, string notes, int? rating, string review);

Task<decimal> CalculateTutorEarningsAsync(int tutorId, DateTime startDate, DateTime endDate);

}

public interface IExamService

{

Task<Exam> CreateExamAsync(Exam exam);

Task<ExamSubject> AddExamSubjectAsync(ExamSubject examSubject);

Task<List<Exam>> GetSchoolExamsAsync(int schoolId, int termId);

Task<byte[]> GenerateExamTimetableAsync(int examId);

}

public interface ISocialMediaService

{

Task<SocialMediaPost> CreatePostAsync(SocialMediaPost post);

Task<bool> PublishToFacebookAsync(int postId);

Task<bool> PublishToWhatsAppAsync(int postId, List<string> phoneNumbers);

Task<bool> PublishToInstagramAsync(int postId);

Task<List<SocialMediaPost>> GetSchoolPostsAsync(int schoolId);

}

// =================== SERVICE IMPLEMENTATIONS ===================

public class GradingService : IGradingService

{

private readonly SchoolManagementContext \_context;

public GradingService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<GradingBody> CreateGradingBodyAsync(GradingBody gradingBody)

{

gradingBody.CreatedAt = DateTime.UtcNow;

\_context.GradingBodies.Add(gradingBody);

await \_context.SaveChangesAsync();

return gradingBody;

}

public async Task<GradingScheme> CreateGradingSchemeAsync(GradingScheme gradingScheme)

{

\_context.GradingSchemes.Add(gradingScheme);

await \_context.SaveChangesAsync();

return gradingScheme;

}

public async Task<GradeScale> AddGradeScaleAsync(GradeScale gradeScale)

{

\_context.GradeScales.Add(gradeScale);

await \_context.SaveChangesAsync();

return gradeScale;

}

public async Task<(string grade, int? unit)> CalculateGradeAsync(decimal score, int gradingSchemeId)

{

var gradeScale = await \_context.GradeScales

.Where(gs => gs.GradingSchemeId == gradingSchemeId

&& score >= gs.MinPercentage

&& score <= gs.MaxPercentage)

.FirstOrDefaultAsync();

return gradeScale != null ? (gradeScale.Symbol, gradeScale.Unit) : ("F", null);

}

public async Task<List<GradingBody>> GetAllGradingBodiesAsync()

{

return await \_context.GradingBodies

.Include(gb => gb.GradingSchemes)

.ThenInclude(gs => gs.GradeScales)

.ToListAsync();

}

public async Task<List<GradingScheme>> GetGradingSchemesByBodyAsync(int gradingBodyId)

{

return await \_context.GradingSchemes

.Where(gs => gs.GradingBodyId == gradingBodyId)

.Include(gs => gs.GradeScales)

.ToListAsync();

}

}

public class StudentService : IStudentService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

public StudentService(SchoolManagementContext context, IGradingService gradingService)

{

\_context = context;

\_gradingService = gradingService;

}

public async Task<Student> CreateStudentAsync(Student student)

{

student.CreatedAt = DateTime.UtcNow;

student.IsActive = true;

student.UserType = "Student";

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

return student;

}

public async Task<Student> UpdateStudentAsync(Student student)

{

\_context.Students.Update(student);

await \_context.SaveChangesAsync();

return student;

}

public async Task<List<Student>> GetStudentsByClassAsync(int classId)

{

return await \_context.Students

.Where(s => s.ClassId == classId && s.IsActive)

.Include(s => s.Parents)

.OrderBy(s => s.LastName)

.ThenBy(s => s.FirstName)

.ToListAsync();

}

public async Task<List<Student>> GetStudentsBySchoolAsync(int schoolId)

{

return await \_context.Students

.Where(s => s.SchoolId == schoolId && s.IsActive)

.Include(s => s.Class)

.Include(s => s.Parents)

.ToListAsync();

}

public async Task<Student> GetStudentByNumberAsync(string studentNumber, int schoolId)

{

return await \_context.Students

.Where(s => s.StudentNumber == studentNumber && s.SchoolId == schoolId)

.Include(s => s.Class)

.Include(s => s.Parents)

.FirstOrDefaultAsync();

}

public async Task<bool> AssignStudentToClassAsync(int studentId, int classId)

{

var student = await \_context.Students.FindAsync(studentId);

if (student == null) return false;

student.ClassId = classId;

await \_context.SaveChangesAsync();

return true;

}

public async Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId)

{

var student = await \_context.Students

.Include(s => s.Class)

.FirstOrDefaultAsync(s => s.Id == studentId);

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Subject)

.Include(r => r.Subject.GradingScheme)

.ThenInclude(gs => gs.GradeScales)

.Include(r => r.Teacher)

.ToListAsync();

var term = await \_context.Terms.FindAsync(termId);

var subjectResults = results.Select(r => new SubjectResultDto

{

SubjectName = r.Subject.Name,

Score = r.Score,

Grade = r.Grade,

Unit = r.Unit,

Comments = r.Comments,

TeacherName = $"{r.Teacher.FirstName} {r.Teacher.LastName}"

}).ToList();

var overallAverage = results.Any() ? results.Average(r => r.Score) : 0;

var gradingScheme = results.FirstOrDefault()?.Subject.GradingScheme;

var overallGrade = "";

int? overallUnit = null;

if (gradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(overallAverage, gradingScheme.Id);

overallGrade = gradeInfo.grade;

overallUnit = gradeInfo.unit;

}

// Calculate position in class

var classAverages = await \_context.Results

.Where(r => r.TermId == termId && r.Student.ClassId == student.ClassId)

.GroupBy(r => r.StudentId)

.Select(g => new { StudentId = g.Key, Average = g.Average(r => r.Score) })

.OrderByDescending(x => x.Average)

.ToListAsync();

var position = classAverages.FindIndex(x => x.StudentId == studentId) + 1;

return new StudentResultDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

ClassName = student.Class.Name,

TermName = term.Name,

SubjectResults = subjectResults,

OverallAverage = overallAverage,

OverallGrade = overallGrade,

OverallUnit = overallUnit,

Position = position,

TotalStudents = classAverages.Count

};

}

public async Task<byte[]> GenerateStudentReportCardAsync(int studentId, int termId)

{

var results = await GetStudentTermResultsAsync(studentId, termId);

// Here you would use a PDF library like iTextSharp or similar

// For now, returning placeholder

var reportContent = $"""

STUDENT REPORT CARD

Student: {results.StudentName}

Student Number: {results.StudentNumber}

Class: {results.ClassName}

Term: {results.TermName}

SUBJECT RESULTS:

{string.Join("\n", results.SubjectResults.Select(sr =>

$"{sr.SubjectName}: {sr.Score}% ({sr.Grade}{(sr.Unit.HasValue ? $" - Unit {sr.Unit}" : "")}) - {sr.Comments}"))}

OVERALL PERFORMANCE:

Average: {results.OverallAverage:F2}%

Grade: {results.OverallGrade}{(results.OverallUnit.HasValue ? $" - Unit {results.OverallUnit}" : "")}

Position: {results.Position} out of {results.TotalStudents}

""";

return System.Text.Encoding.UTF8.GetBytes(reportContent);

}

}

public class ResultService : IResultService

{

private readonly SchoolManagementContext \_context;

private readonly IGradingService \_gradingService;

private readonly INotificationService \_notificationService;

public ResultService(SchoolManagementContext context, IGradingService gradingService, INotificationService notificationService)

{

\_context = context;

\_gradingService = gradingService;

\_notificationService = notificationService;

}

public async Task<r> RecordResultAsync(r result)

{

// Calculate grade and unit based on score

var subject = await \_context.Subjects

.Include(s => s.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == result.SubjectId);

if (subject?.GradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(result.Score, subject.GradingScheme.Id);

result.Grade = gradeInfo.grade;

result.Unit = gradeInfo.unit;

}

result.DateRecorded = DateTime.UtcNow;

\_context.Results.Add(result);

await \_context.SaveChangesAsync();

// Send notification to parents

await SendResultNotificationToParentsAsync(result);

return result;

}

public async Task<List<r>> BulkRecordResultsAsync(List<r> results)

{

foreach (var result in results)

{

var subject = await \_context.Subjects

.Include(s => s.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == result.SubjectId);

if (subject?.GradingScheme != null)

{

var gradeInfo = await \_gradingService.CalculateGradeAsync(result.Score, subject.GradingScheme.Id);

result.Grade = gradeInfo.grade;

result.Unit = gradeInfo.unit;

}

result.DateRecorded = DateTime.UtcNow;

}

\_context.Results.AddRange(results);

await \_context.SaveChangesAsync();

// Send notifications

foreach (var result in results)

{

await SendResultNotificationToParentsAsync(result);

}

return results;

}

public async Task<List<StudentResultDto>> GetClassResultsAsync(int classId, int termId)

{

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.Include(s => s.Results.Where(r => r.TermId == termId))

.ThenInclude(r => r.Subject)

.ThenInclude(s => s.GradingScheme)

.ToListAsync();

var studentResults = new List<StudentResultDto>();

foreach (var student in students)

{

var subjectResults = student.Results.Select(r => new SubjectResultDto

{

SubjectName = r.Subject.Name,

Score = r.Score,

Grade = r.Grade,

Unit = r.Unit,

Comments = r.Comments

}).ToList();

var overallAverage = student.Results.Any() ? student.Results.Average(r => r.Score) : 0;

studentResults.Add(new StudentResultDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

SubjectResults = subjectResults,

OverallAverage = overallAverage

});

}

// Calculate positions

var sortedResults = studentResults.OrderByDescending(sr => sr.OverallAverage).ToList();

for (int i = 0; i < sortedResults.Count; i++)

{

sortedResults[i].Position = i + 1;

sortedResults[i].TotalStudents = sortedResults.Count;

}

return sortedResults;

}

public async Task<byte[]> GenerateClassReportAsync(int classId, int termId)

{

var classResults = await GetClassResultsAsync(classId, termId);

var classInfo = await \_context.Classes

.Include(c => c.School)

.FirstOrDefaultAsync(c => c.Id == classId);

var term = await \_context.Terms.FindAsync(termId);

var reportContent = $"""

CLASS PERFORMANCE REPORT

School: {classInfo.School.Name}

Class: {classInfo.Name}

Term: {term.Name}

STUDENT RESULTS:

{string.Join("\n", classResults.Select(sr =>

$"{sr.Position}. {sr.StudentName} ({sr.StudentNumber}) - Average: {sr.OverallAverage:F2}%"))}

CLASS STATISTICS:

Total Students: {classResults.Count}

Class Average: {(classResults.Any() ? classResults.Average(sr => sr.OverallAverage) : 0):F2}%

Highest Score: {(classResults.Any() ? classResults.Max(sr => sr.OverallAverage) : 0):F2}%

Lowest Score: {(classResults.Any() ? classResults.Min(sr => sr.OverallAverage) : 0):F2}%

""";

return System.Text.Encoding.UTF8.GetBytes(reportContent);

}

public async Task<StudentResultDto> CalculateStudentPositionAsync(int studentId, int termId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetStudentTermResultsAsync(studentId, termId);

}

private async Task SendResultNotificationToParentsAsync(r result)

{

var student = await \_context.Students

.Include(s => s.Parents)

.Include(s => s.Class)

.FirstOrDefaultAsync(s => s.Id == result.StudentId);

var subject = await \_context.Subjects.FindAsync(result.SubjectId);

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "New Result Posted",

Message = $"New {subject.Name} result for {student.FirstName}: {result.Score}% ({result.Grade})",

Type = NotificationType.Results,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

}

private async Task<StudentResultDto> GetStudentTermResultsAsync(int studentId, int termId)

{

// Implementation moved to StudentService for better organization

var studentService = new StudentService(\_context, \_gradingService);

return await studentService.GetStudentTermResultsAsync(studentId, termId);

}

}

public class AttendanceService : IAttendanceService

{

private readonly SchoolManagementContext \_context;

private readonly INotificationService \_notificationService;

public AttendanceService(SchoolManagementContext context, INotificationService notificationService)

{

\_context = context;

\_notificationService = notificationService;

}

public async Task<Attendance> RecordAttendanceAsync(Attendance attendance)

{

// Check if attendance already exists for this student and date

var existingAttendance = await \_context.Attendances

.FirstOrDefaultAsync(a => a.StudentId == attendance.StudentId

&& a.Date.Date == attendance.Date.Date);

if (existingAttendance != null)

{

existingAttendance.Status = attendance.Status;

existingAttendance.Remarks = attendance.Remarks;

existingAttendance.RecordedByTeacherId = attendance.RecordedByTeacherId;

}

else

{

\_context.Attendances.Add(attendance);

}

await \_context.SaveChangesAsync();

// Send notification if absent

if (attendance.Status == AttendanceStatus.Absent)

{

await SendAbsenteeNotificationAsync(attendance);

}

return existingAttendance ?? attendance;

}

public async Task<List<Attendance>> BulkRecordAttendanceAsync(List<Attendance> attendances)

{

var results = new List<Attendance>();

foreach (var attendance in attendances)

{

var result = await RecordAttendanceAsync(attendance);

results.Add(result);

}

return results;

}

public async Task<List<AttendanceReportDto>> GetClassAttendanceReportAsync(int classId, DateTime startDate, DateTime endDate)

{

var students = await \_context.Students

.Where(s => s.ClassId == classId)

.ToListAsync();

var attendanceData = await \_context.Attendances

.Where(a => a.ClassId == classId

&& a.Date >= startDate

&& a.Date <= endDate)

.GroupBy(a => a.StudentId)

.Select(g => new

{

StudentId = g.Key,

TotalDays = g.Count(),

PresentDays = g.Count(a => a.Status == AttendanceStatus.Present),

AbsentDays = g.Count(a => a.Status == AttendanceStatus.Absent),

LateDays = g.Count(a => a.Status == AttendanceStatus.Late)

})

.ToListAsync();

var totalSchoolDays = await CalculateSchoolDaysAsync(startDate, endDate);

return students.Select(s =>

{

var attendance = attendanceData.FirstOrDefault(a => a.StudentId == s.Id);

var presentDays = attendance?.PresentDays ?? 0;

var totalDays = Math.Max(attendance?.TotalDays ?? 0, totalSchoolDays);

return new AttendanceReportDto

{

StudentName = $"{s.FirstName} {s.LastName}",

StudentNumber = s.StudentNumber,

TotalDays = totalDays,

PresentDays = presentDays,

AbsentDays = attendance?.AbsentDays ?? 0,

LateDays = attendance?.LateDays ?? 0,

AttendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0

};

}).ToList();

}

public async Task<AttendanceReportDto> GetStudentAttendanceReportAsync(int studentId, DateTime startDate, DateTime endDate)

{

var student = await \_context.Students.FindAsync(studentId);

var attendances = await \_context.Attendances

.Where(a => a.StudentId == studentId

&& a.Date >= startDate

&& a.Date <= endDate)

.ToListAsync();

var totalDays = await CalculateSchoolDaysAsync(startDate, endDate);

var presentDays = attendances.Count(a => a.Status == AttendanceStatus.Present);

var absentDays = attendances.Count(a => a.Status == AttendanceStatus.Absent);

var lateDays = attendances.Count(a => a.Status == AttendanceStatus.Late);

return new AttendanceReportDto

{

StudentName = $"{student.FirstName} {student.LastName}",

StudentNumber = student.StudentNumber,

TotalDays = Math.Max(attendances.Count, totalDays),

PresentDays = presentDays,

AbsentDays = absentDays,

LateDays = lateDays,

AttendancePercentage = totalDays > 0 ? (decimal)presentDays / totalDays \* 100 : 0

};

}

private async Task<int> CalculateSchoolDaysAsync(DateTime startDate, DateTime endDate)

{

// Calculate weekdays between dates (excluding weekends)

var days = 0;

for (var date = startDate; date <= endDate; date = date.AddDays(1))

{

if (date.DayOfWeek != DayOfWeek.Saturday && date.DayOfWeek != DayOfWeek.Sunday)

days++;

}

return days;

}

private async Task SendAbsenteeNotificationAsync(Attendance attendance)

{

var student = await \_context.Students

.Include(s => s.Parents)

.FirstOrDefaultAsync(s => s.Id == attendance.StudentId);

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "Student Absent",

Message = $"{student.FirstName} was marked absent on {attendance.Date:yyyy-MM-dd}",

Type = NotificationType.Attendance,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

await \_notificationService.SendNotificationAsync(notification.Id);

}

}

}

public class TimetableService : ITimetableService

{

private readonly SchoolManagementContext \_context;

public TimetableService(SchoolManagementContext context)

{

\_context = context;

}

public async Task<Timetable> GenerateTimetableAsync(int classId, int schoolYearId)

{

var classEntity = await \_context.Classes

.Include(c => c.ClassSubjects)

.ThenInclude(cs => cs.Subject)

.Include(c => c.ClassSubjects)

.ThenInclude(cs => cs.Teacher)

.FirstOrDefaultAsync(c => c.Id == classId);

var timetable = new Timetable

{

Name = $"{classEntity.Name} Timetable {DateTime.Now.Year}",

ClassId = classId,

SchoolYearId = schoolYearId,

IsActive = true,

CreatedAt = DateTime.UtcNow

};

\_context.Timetables.Add(timetable);

await \_context.SaveChangesAsync();

// Generate basic timetable structure

await GenerateBasicTimetableStructureAsync(timetable, classEntity.ClassSubjects.ToList());

return timetable;

}

private async Task GenerateBasicTimetableStructureAsync(Timetable timetable, List<ClassSubject> classSubjects)

{

var timeSlots = new[]

{

(new TimeSpan(8, 0, 0), new TimeSpan(8, 45, 0)),

(new TimeSpan(8, 45, 0), new TimeSpan(9, 30, 0)),

(new TimeSpan(9, 30, 0), new TimeSpan(9, 45, 0)), // Break

(new TimeSpan(9, 45, 0), new TimeSpan(10, 30, 0)),

(new TimeSpan(10, 30, 0), new TimeSpan(11, 15, 0)),

(new TimeSpan(11, 15, 0), new TimeSpan(12, 0, 0)),

(new TimeSpan(12, 0, 0), new TimeSpan(13, 0, 0)), // Lunch

(new TimeSpan(13, 0, 0), new TimeSpan(13, 45, 0)),

(new TimeSpan(13, 45, 0), new TimeSpan(14, 30, 0))

};

var workingDays = new[] { DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday, DayOfWeek.Thursday, DayOfWeek.Friday };

var subjectRotation = 0;

foreach (var day in workingDays)

{

for (int i = 0; i < timeSlots.Length; i++)

{

var (startTime, endTime) = timeSlots[i];

TimetableSlot slot;

if (i == 2) // Break time

{

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Type = SlotType.Break,

Room = "Playground"

};

}

else if (i == 6) // Lunch time

{

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Type = SlotType.Lunch,

Room = "Cafeteria"

};

}

else if (classSubjects.Any())

{

var classSubject = classSubjects[subjectRotation % classSubjects.Count];

slot = new TimetableSlot

{

TimetableId = timetable.Id,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

ClassSubjectId = classSubject.Id,

Type = SlotType.Regular,

Room = $"Room {subjectRotation + 1}"

};

subjectRotation++;

}

else continue;

\_context.TimetableSlots.Add(slot);

}

}

await \_context.SaveChangesAsync();

}

public async Task<TimetableDto> GetClassTimetableAsync(int classId)

{

var timetable = await \_context.Timetables

.Where(t => t.ClassId == classId && t.IsActive)

.Include(t => t.Slots)

.ThenInclude(s => s.ClassSubject)

.ThenInclude(cs => cs.Subject)

.Include(t => t.Slots)

.ThenInclude(s => s.ClassSubject)

.ThenInclude(cs => cs.Teacher)

.Include(t => t.Class)

.FirstOrDefaultAsync();

if (timetable == null) return null;

var slots = timetable.Slots.Select(s => new TimetableSlotDto

{

DayOfWeek = s.DayOfWeek,

StartTime = s.StartTime.ToString(@"hh\:mm"),

EndTime = s.EndTime.ToString(@"hh\:mm"),

SubjectName = s.ClassSubject?.Subject?.Name ?? s.Type.ToString(),

TeacherName = s.ClassSubject?.Teacher != null

? $"{s.ClassSubject.Teacher.FirstName} {s.ClassSubject.Teacher.LastName}"

: "",

Room = s.Room,

Type = s.Type

}).ToList();

return new TimetableDto

{

Id = timetable.Id,

ClassName = timetable.Class.Name,

Slots = slots

};

}

public async Task<TimetableDto> GetStudentTimetableAsync(int studentId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetClassTimetableAsync(student.ClassId);

}

public async Task<TimetableSlot> UpdateTimetableSlotAsync(TimetableSlot slot)

{

\_context.TimetableSlots.Update(slot);

await \_context.SaveChangesAsync();

return slot;

}

public async Task<bool> ValidateTimetableRulesAsync(int timetableId)

{

var slots = await \_context.TimetableSlots

.Where(s => s.TimetableId == timetableId)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Teacher)

.ToListAsync();

// Check for teacher conflicts

var teacherConflicts = slots

.Where(s => s.ClassSubject?.Teacher != null)

.GroupBy(s => new { s.DayOfWeek, s.ClassSubject.TeacherId })

.Where(g => g.Any(s1 => g.Any(s2 => s1.Id != s2.Id

&& s1.StartTime < s2.EndTime

&& s2.StartTime < s1.EndTime)))

.Any();

// Check for room conflicts

var roomConflicts = slots

.Where(s => !string.IsNullOrEmpty(s.Room))

.GroupBy(s => new { s.DayOfWeek, s.Room })

.Where(g => g.Any(s1 => g.Any(s2 => s1.Id != s2.Id

&& s1.StartTime < s2.EndTime

&& s2.StartTime < s1.EndTime)))

.Any();

return !teacherConflicts && !roomConflicts;

}

public async Task<List<TimetableSlot>> GetTeacherScheduleAsync(int teacherId, DateTime date)

{

return await \_context.TimetableSlots

.Where(s => s.ClassSubject.TeacherId == teacherId)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Subject)

.Include(s => s.ClassSubject)

.ThenInclude(cs => cs.Class)

.ToListAsync();

}

}

public class AssignmentService : IAssignmentService

{

private readonly SchoolManagementContext \_context;

private readonly INotificationService \_notificationService;

private readonly IHandwritingRecognitionService \_handwritingService;

public AssignmentService(SchoolManagementContext context, INotificationService notificationService, IHandwritingRecognitionService handwritingService)

{

\_context = context;

\_notificationService = notificationService;

\_handwritingService = handwritingService;

}

public async Task<Assignment> CreateAssignmentAsync(Assignment assignment)

{

assignment.CreatedAt = DateTime.UtcNow;

\_context.Assignments.Add(assignment);

await \_context.SaveChangesAsync();

// Notify students/parents about new assignment

await SendAssignmentNotificationAsync(assignment);

return assignment;

}

public async Task<AssignmentSubmission> SubmitAssignmentAsync(AssignmentSubmission submission)

{

submission.SubmittedAt = DateTime.UtcNow;

submission.Status = submission.SubmittedAt <= await GetAssignmentDueDateAsync(submission.AssignmentId)

? SubmissionStatus.Submitted

: SubmissionStatus.Late;

\_context.AssignmentSubmissions.Add(submission);

await \_context.SaveChangesAsync();

// Process handwritten answers

foreach (var answer in submission.Answers.Where(a => !string.IsNullOrEmpty(a.HandwrittenImagePath)))

{

var recognizedText = await \_handwritingService.RecognizeHandwritingAsync(answer.HandwrittenImagePath, submission.StudentId);

answer.ProcessedText = recognizedText;

}

// Auto-grade if it's an online platform assignment

var assignment = await \_context.Assignments.FindAsync(submission.AssignmentId);

if (assignment.IsOnlinePlatformWork)

{

await AutoGradeAssignmentAsync(submission.Id);

}

return submission;

}

public async Task<AssignmentSubmission> AutoGradeAssignmentAsync(int submissionId)

{

var submission = await \_context.AssignmentSubmissions

.Include(s => s.Assignment)

.ThenInclude(a => a.Questions)

.ThenInclude(q => q.Options)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == submissionId);

decimal totalScore = 0;

decimal maxScore = submission.Assignment.Questions.Sum(q => q.Marks);

foreach (var answer in submission.Answers)

{

var question = submission.Assignment.Questions.First(q => q.Id == answer.QuestionId);

switch (question.Type)

{

case QuestionType.MultipleChoice:

var correctOption = question.Options.FirstOrDefault(o => o.IsCorrect);

if (correctOption != null && answer.Answer == correctOption.OptionLetter.ToString())

{

answer.IsCorrect = true;

answer.Score = question.Marks;

totalScore += question.Marks;

}

break;

case QuestionType.TrueFalse:

if (string.Equals(answer.Answer, question.CorrectAnswer, StringComparison.OrdinalIgnoreCase))

{

answer.IsCorrect = true;

answer.Score = question.Marks;

totalScore += question.Marks;

}

break;

case QuestionType.ShortAnswer:

// Simple string matching - could be enhanced with fuzzy matching

var similarity = CalculateStringSimilarity(answer.Answer, question.CorrectAnswer);

if (similarity > 0.8m)

{

answer.IsCorrect = true;

answer.Score = question.Marks \* similarity;

totalScore += answer.Score.Value;

}

break;

case QuestionType.Handwritten:

// Use processed handwritten text for comparison

var handwritingSimilarity = CalculateStringSimilarity(answer.ProcessedText, question.CorrectAnswer);

if (handwritingSimilarity > 0.7m) // Lower threshold for handwriting

{

answer.IsCorrect = true;

answer.Score = question.Marks \* handwritingSimilarity;

totalScore += answer.Score.Value;

}

break;

}

}

submission.Score = maxScore > 0 ? (totalScore / maxScore) \* 100 : 0;

submission.Status = SubmissionStatus.Graded;

submission.IsAutoGraded = true;

await \_context.SaveChangesAsync();

return submission;

}

public async Task<List<Assignment>> GetClassAssignmentsAsync(int classId)

{

return await \_context.Assignments

.Where(a => a.ClassId == classId)

.Include(a => a.Subject)

.Include(a => a.Teacher)

.Include(a => a.Questions)

.OrderByDescending(a => a.CreatedAt)

.ToListAsync();

}

public async Task<List<Assignment>> GetStudentAssignmentsAsync(int studentId)

{

var student = await \_context.Students.FindAsync(studentId);

return await GetClassAssignmentsAsync(student.ClassId);

}

public async Task<decimal> CalculateContinuousAssessmentMarkAsync(int studentId, int subjectId, int termId)

{

var submissions = await \_context.AssignmentSubmissions

.Where(s => s.StudentId == studentId

&& s.Assignment.SubjectId == subjectId

&& s.Status == SubmissionStatus.Graded)

.Include(s => s.Assignment)

.ToListAsync();

if (!submissions.Any()) return 0;

// Weight different assignment types

var weightedScores = submissions.Select(s => new

{

Score = s.Score ?? 0,

Weight = s.Assignment.Type switch

{

AssignmentType.Homework => 0.2m,

AssignmentType.Classwork => 0.3m,

AssignmentType.Assignment => 0.3m,

AssignmentType.Project => 0.4m,

AssignmentType.Quiz => 0.25m,

\_ => 0.25m

}

});

var totalWeightedScore = weightedScores.Sum(ws => ws.Score \* ws.Weight);

var totalWeight = weightedScores.Sum(ws => ws.Weight);

return totalWeight > 0 ? totalWeightedScore / totalWeight : 0;

}

private decimal CalculateStringSimilarity(string str1, string str2)

{

if (string.IsNullOrEmpty(str1) || string.IsNullOrEmpty(str2)) return 0;

// Simple Levenshtein distance-based similarity

var distance = LevenshteinDistance(str1.ToLower(), str2.ToLower());

var maxLength = Math.Max(str1.Length, str2.Length);

return maxLength > 0 ? 1m - (decimal)distance / maxLength : 0;

}

private int LevenshteinDistance(string s1, string s2)

{

if (s1.Length == 0) return s2.Length;

if (s2.Length == 0) return s1.Length;

int[,] d = new int[s1.Length + 1, s2.Length + 1];

for (int i = 0; i <= s1.Length; i++) d[i, 0] = i;

for (int j = 0; j <= s2.Length; j++) d[0, j] = j;

for (int i = 1; i <= s1.Length; i++)

{

for (int j = 1; j <= s2.Length; j++)

{

int cost = s1[i - 1] == s2[j - 1] ? 0 : 1;

d[i, j] = Math.Min(Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1), d[i - 1, j - 1] + cost);

}

}

return d[s1.Length, s2.Length];

}

private async Task<DateTime> GetAssignmentDueDateAsync(int assignmentId)

{

var assignment = await \_context.Assignments.FindAsync(assignmentId);

return assignment.DueDate;

}

private async Task SendAssignmentNotificationAsync(Assignment assignment)

{

var students = await \_context.Students

.Where(s => s.ClassId == assignment.ClassId)

.Include(s => s.Parents)

.ToListAsync();

foreach (var student in students)

{

foreach (var parent in student.Parents.Where(p => p.ReceiveNotifications))

{

var notification = new Notification

{

Title = "New Assignment",

Message = $"New {assignment.Type} assigned: {assignment.Title}. Due: {assignment.DueDate:yyyy-MM-dd}",

Type = NotificationType.Assignment,

ParentId = parent.Id,

StudentId = student.Id,

CreatedAt = DateTime.UtcNow,

Channel = NotificationChannel.InApp

};

await \_notificationService.CreateNotificationAsync(notification);

}

}

}

}

public class HandwritingRecognitionService : IHandwritingRecognitionService

{

private readonly SchoolManagementContext \_context;

private readonly IConfiguration \_configuration;

private readonly HttpClient \_httpClient;

public HandwritingRecognitionService(SchoolManagementContext context, IConfiguration configuration, HttpClient httpClient)

{

\_context = context;

\_configuration = configuration;

\_httpClient = httpClient;

}

public async Task<string> RecognizeHandwritingAsync(string imagePath, int studentId)

{

// First try local model if available

var localResult = await ProcessHandwritingOfflineAsync(imagePath, studentId);

if (localResult) return await GetLocalRecognitionResultAsync(imagePath);

// Fallback to cloud-based recognition

return await ProcessHandwritingCloudAsync(imagePath, studentId);

}

public async Task<HandwritingSample> CreateHandwritingSampleAsync(HandwritingRecognitionDto dto)

{

var imageBytes = Convert.FromBase64String(dto.ImageBase64);

var imagePath = await SaveImageAsync(imageBytes, dto.StudentId);

var recognizedText = await RecognizeHandwritingAsync(imagePath, dto.StudentId);

var sample = new HandwritingSample

{

StudentId = dto.StudentId,

ImagePath = imagePath,

ExpectedText = dto.ExpectedText,

RecognizedText = recognizedText,

Type = dto.Type,

CreatedAt = DateTime.UtcNow,

IsTrainingData = true,

Confidence = CalculateConfidence(dto.ExpectedText, recognizedText)

};

\_context.HandwritingSamples.Add(sample);

await \_context.SaveChangesAsync();

return sample;

}

public async Task<bool> ValidateHandwritingSampleAsync(int sampleId, string correctedText, int teacherId)

{

var sample = await \_context.HandwritingSamples.FindAsync(sampleId);

if (sample == null) return false;

var validation = new HandwritingValidation

{

HandwritingSampleId = sampleId,

TeacherId = teacherId,

CorrectedText = correctedText,

ValidatedAt = DateTime.UtcNow,

Status = ValidationStatus.Approved

};

\_context.HandwritingValidations.Add(validation);

sample.IsValidated = true;

sample.RecognizedText = correctedText;

await \_context.SaveChangesAsync();

// Trigger model retraining if enough samples

await CheckAndTriggerModelRetrainingAsync(sample.StudentId);

return true;

}

public async Task TrainPersonalizedModelAsync(int studentId)

{

var trainingSamples = await \_context.HandwritingSamples

.Where(s => s.StudentId == studentId && s.IsValidated)

.ToListAsync();

if (trainingSamples.Count < 50) // Minimum samples for training

{

throw new InvalidOperationException("Insufficient training samples. Minimum 50 validated samples required.");

}

// Prepare training data

var trainingData = trainingSamples.Select(s => new

{

ImagePath = s.ImagePath,

GroundTruth = s.RecognizedText, // Use validated text

Type = s.Type

}).ToList();

// Call ML training service (this would be implemented with ML.NET or similar)

var modelPath = await TrainModelAsync(studentId, trainingData);

var existingModel = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId);

if (existingModel != null)

{

existingModel.ModelPath = modelPath;

existingModel.LastTrainingDate = DateTime.UtcNow;

existingModel.SampleCount = trainingSamples.Count;

existingModel.Accuracy = await CalculateModelAccuracyAsync(modelPath, studentId);

}

else

{

var newModel = new PersonalizedModel

{

StudentId = studentId,

ModelPath = modelPath,

LastTrainingDate = DateTime.UtcNow,

SampleCount = trainingSamples.Count,

Accuracy = await CalculateModelAccuracyAsync(modelPath, studentId),

IsDeployedLocally = false,

IsDeployedOnCloud = true

};

\_context.PersonalizedModels.Add(newModel);

}

await \_context.SaveChangesAsync();

}

public async Task<PersonalizedModel> DeployModelLocallyAsync(int studentId)

{

var model = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId);

if (model == null) return null;

// Deploy model to local device (implementation depends on your mobile/desktop app architecture)

var localPath = await DeployToLocalDeviceAsync(model.CloudModelPath, studentId);

model.ModelPath = localPath;

model.IsDeployedLocally = true;

await \_context.SaveChangesAsync();

return model;

}

public async Task<string> GenerateTrainingContentAsync(HandwritingType type, string difficulty = "beginner")

{

return type switch

{

HandwritingType.Alphabet => GenerateAlphabetContent(difficulty),

HandwritingType.Number => GenerateNumberContent(difficulty),

HandwritingType.Word => GenerateWordContent(difficulty),

HandwritingType.Sentence => GenerateSentenceContent(difficulty),

\_ => "Practice writing: Hello World"

};

}

public async Task<bool> ProcessHandwritingOfflineAsync(string imagePath, int studentId)

{

var model = await \_context.PersonalizedModels

.FirstOrDefaultAsync(m => m.StudentId == studentId && m.IsDeployedLocally);

if (model == null) return false;

// Process using local model (implementation depends on your ML framework)

try

{

await ProcessWithLocalModelAsync(imagePath, model.ModelPath);

return true;

}

catch

{

return false;

}

}

private async Task<string> ProcessHandwritingCloudAsync(string imagePath, int studentId)

{

// Call cloud-based handwriting recognition API

var cloudApiUrl = \_configuration["HandwritingRecognition:CloudApiUrl"];

using var content = new MultipartFormDataContent();

var imageBytes = await File.ReadAllBytesAsync(imagePath);

content.Add(new ByteArrayContent(imageBytes), "image", "handwriting.jpg");

content.Add(new StringContent(studentId.ToString()), "studentId");

var response = await \_httpClient.PostAsync(cloudApiUrl, content);

var result = await response.Content.ReadAsStringAsync();

return result; // Assume API returns recognized text

}

private decimal CalculateConfidence(string expected, string recognized)

{

if (string.IsNullOrEmpty(expected) || string.IsNullOrEmpty(recognized)) return 0;

var similarity = 1m - (decimal)LevenshteinDistance(expected.ToLower(), recognized.ToLower()) / Math.Max(expected.Length, recognized.Length);

return Math.Max(0, Math.Min(1, similarity));

}

private int LevenshteinDistance(string s1, string s2)

{

// Same implementation as in AssignmentService

if (s1.Length == 0) return s2.Length;

if (s2.Length == 0) return s1.Length;

int[,] d = new int[s1.Length + 1, s2.Length + 1];

for (int i = 0; i <= s1.Length; i++) d[i, 0] = i;

for (int j = 0; j <= s2.Length; j++) d[0, j] = j;

for (int i = 1; i <= s1.Length; i++)

{

for (int j = 1; j <= s2.Length; j++)

{

int cost = s1[i - 1] == s2[j - 1] ? 0 : 1;

d[i, j] = Math.Min(Math.Min(d[i - 1, j] + 1, d[i, j - 1] + 1), d[i - 1, j - 1] + cost);

}

}

return d[s1.Length, s2.Length];

}

private string GenerateAlphabetContent(string difficulty)

{

return difficulty switch

{

"beginner" => "A B C D E F G H I J K L M N O P Q R S T U V W X Y Z",

"intermediate" => "a b c d e f g h i j k l m n o p q r s t u v w x y z",

"advanced" => "Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj Kk Ll Mm Nn Oo Pp Qq Rr Ss Tt Uu Vv Ww Xx Yy Zz",

\_ => "A B C D E"

};

}

private string GenerateNumberContent(string difficulty)

{

return difficulty switch

{

"beginner" => "1 2 3 4 5 6 7 8 9 0",

"intermediate" => "12 34 56 78 90 123 456 789",

"advanced" => "1,234 5,678 9,012 3,456 7,890",

\_ => "1 2 3 4 5"

};

}

private string GenerateWordContent(string difficulty)

{

return difficulty switch

{

"beginner" => "cat dog sun fun run",

"intermediate" => "school book pencil teacher student",

"advanced" => "education mathematics science literature",

\_ => "cat dog"

};

}

private string GenerateSentenceContent(string difficulty)

{

return difficulty switch

{

"beginner" => "The cat sat on the mat.",

"intermediate" => "I love going to school every day.",

"advanced" => "Education is the most powerful weapon which you can use to change the world.",

\_ => "Hello world."

};

}

private async Task<string> SaveImageAsync(byte[] imageBytes, int studentId)

{

var uploadsPath = Path.Combine("uploads", "handwriting", studentId.ToString());

Directory.CreateDirectory(uploadsPath);

var fileName = $"{Guid.NewGuid()}.jpg";

var filePath = Path.Combine(uploadsPath, fileName);

await File.WriteAllBytesAsync(filePath, imageBytes);

return filePath;

}

private async Task CheckAndTriggerModelRetrainingAsync(int studentId)

{

var validatedSamples = await \_context.HandwritingSamples

.CountAsync(s => s.StudentId == studentId && s.IsValidated);

if (validatedSamples >= 50 && validatedSamples % 25 == 0) // Retrain every 25 new samples

{

await TrainPersonalizedModelAsync(studentId);

}

}

private async Task<string> TrainModelAsync(int studentId, object trainingData)

{

// Implementation would use ML.NET or similar framework

var modelPath = $"models/student\_{studentId}\_{DateTime.UtcNow:yyyyMMdd}.model";

// Training logic here...

return modelPath;

}

private async Task<decimal> CalculateModelAccuracyAsync(string modelPath, int studentId)

{

// Test model accuracy on validation set

return 0.85m; // Placeholder

}

private async Task<string> DeployToLocalDeviceAsync(string cloudPath, int studentId)

{

// Deploy to local device storage

return $"local/models/student\_{studentId}.model";

}

private async Task<string> GetLocalRecognitionResultAsync(string imagePath)

{

// Get result from local processing

return "Sample recognized text";

}

private async Task ProcessWithLocalModelAsync(string imagePath, string modelPath)

{

// Process image with local model

await Task.Delay(100); // Placeholder

}

}

// =================== AI ASSESSMENT & VERIFICATION SYSTEM ===================

public interface IAIAssessmentService

{

Task<AIAssessmentResult> AssessSubmissionAsync(int submissionId);

Task<List<AIAssessmentResult>> BatchAssessSubmissionsAsync(List<int> submissionIds);

Task<AIAssessmentResult> ProcessHandwrittenAnswerAsync(int submissionAnswerId);

Task<bool> FlagForTeacherReviewAsync(int assessmentResultId, string reason);

Task<List<AIAssessmentResult>> GetPendingReviewsAsync(int teacherId);

Task<AIAssessmentResult> TeacherVerifyAssessmentAsync(int assessmentResultId, int teacherId, TeacherVerificationDto verification);

}

public class AIAssessmentResult

{

public int Id { get; set; }

public int? SubmissionId { get; set; }

public AssignmentSubmission Submission { get; set; }

public int? SubmissionAnswerId { get; set; }

public SubmissionAnswer SubmissionAnswer { get; set; }

public decimal AIScore { get; set; }

public decimal ConfidenceLevel { get; set; }

public string AIFeedback { get; set; }

public AIAssessmentStatus Status { get; set; }

public bool RequiresTeacherReview { get; set; }

public string ReviewReason { get; set; }

public DateTime ProcessedAt { get; set; }

// Teacher verification

public int? VerifiedByTeacherId { get; set; }

public Teacher VerifiedByTeacher { get; set; }

public decimal? TeacherScore { get; set; }

public string TeacherFeedback { get; set; }

public DateTime? VerifiedAt { get; set; }

public VerificationStatus? VerificationStatus { get; set; }

// Detailed AI analysis

public string HandwritingRecognitionText { get; set; }

public decimal HandwritingConfidence { get; set; }

public List<AIScoreBreakdown> ScoreBreakdowns { get; set; } = new();

}

public enum AIAssessmentStatus

{

Processing,

Completed,

Failed,

PendingReview,

Verified,

Disputed

}

public enum VerificationStatus

{

Approved,

Modified,

Rejected,

NeedsReprocessing

}

public class AIScoreBreakdown

{

public int Id { get; set; }

public int AIAssessmentResultId { get; set; }

public AIAssessmentResult AIAssessmentResult { get; set; }

public string Criterion { get; set; } // "Accuracy", "Completeness", "Clarity", "Grammar"

public decimal Score { get; set; }

public decimal MaxScore { get; set; }

public string Explanation { get; set; }

}

public class TeacherVerificationDto

{

public decimal? OverrideScore { get; set; }

public string TeacherFeedback { get; set; }

public VerificationStatus VerificationStatus { get; set; }

public List<CriterionVerification> CriterionVerifications { get; set; } = new();

}

public class CriterionVerification

{

public string Criterion { get; set; }

public decimal TeacherScore { get; set; }

public string TeacherComment { get; set; }

}

// Add to DbContext

public class SchoolManagementContext : DbContext

{

// ... existing DbSets ...

public DbSet<AIAssessmentResult> AIAssessmentResults { get; set; }

public DbSet<AIScoreBreakdown> AIScoreBreakdowns { get; set; }

// ... existing configuration methods ...

private void ConfigureAIAssessmentSystem(ModelBuilder modelBuilder)

{

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.AIScore)

.HasPrecision(5, 2);

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.ConfidenceLevel)

.HasPrecision(5, 4);

modelBuilder.Entity<AIAssessmentResult>()

.Property(ar => ar.HandwritingConfidence)

.HasPrecision(5, 4);

modelBuilder.Entity<AIScoreBreakdown>()

.Property(sb => sb.Score)

.HasPrecision(5, 2);

modelBuilder.Entity<AIScoreBreakdown>()

.Property(sb => sb.MaxScore)

.HasPrecision(5, 2);

}

}

public class AIAssessmentService : IAIAssessmentService

{

private readonly SchoolManagementContext \_context;

private readonly IHandwritingRecognitionService \_handwritingService;

private readonly INotificationService \_notificationService;

private readonly HttpClient \_httpClient;

private readonly IConfiguration \_configuration;

public AIAssessmentService(

SchoolManagementContext context,

IHandwritingRecognitionService handwritingService,

INotificationService notificationService,

HttpClient httpClient,

IConfiguration configuration)

{

\_context = context;

\_handwritingService = handwritingService;

\_notificationService = notificationService;

\_httpClient = httpClient;

\_configuration = configuration;

}

public async Task<AIAssessmentResult> AssessSubmissionAsync(int submissionId)

{

var submission = await \_context.AssignmentSubmissions

.Include(s => s.Assignment)

.ThenInclude(a => a.Questions)

.ThenInclude(q => q.Options)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == submissionId);

if (submission == null) return null;

var assessmentResult = new AIAssessmentResult

{

SubmissionId = submissionId,

Status = AIAssessmentStatus.Processing,

ProcessedAt = DateTime.UtcNow

};

\_context.AIAssessmentResults.Add(assessmentResult);

await \_context.SaveChangesAsync();

try

{

decimal totalScore = 0;

decimal maxPossibleScore = submission.Assignment.Questions.Sum(q => q.Marks);

var allBreakdowns = new List<AIScoreBreakdown>();

foreach (var answer in submission.Answers)

{

var question = submission.Assignment.Questions.First(q => q.Id == answer.QuestionId);

var answerAssessment = await AssessIndividualAnswerAsync(answer, question);

totalScore += answerAssessment.Score;

allBreakdowns.AddRange(answerAssessment.Breakdowns);

// Update the answer with AI results

answer.Score = answerAssessment.Score;

answer.IsCorrect = answerAssessment.Score >= (question.Marks \* 0.7m); // 70% threshold

answer.ProcessedText = answerAssessment.ProcessedText;

answer.Confidence = answerAssessment.Confidence;

}

// Calculate final score and confidence

assessmentResult.AIScore = maxPossibleScore > 0 ? (totalScore / maxPossibleScore) \* 100 : 0;

assessmentResult.ConfidenceLevel = allBreakdowns.Any() ? allBreakdowns.Average(b => b.Score / b.MaxScore) : 0;

assessmentResult.AIFeedback = GenerateOverallFeedback(allBreakdowns, assessmentResult.AIScore);

assessmentResult.Status = AIAssessmentStatus.Completed;

assessmentResult.ScoreBreakdowns = allBreakdowns;

// Determine if teacher review is needed

var needsReview = DetermineIfTeacherReviewNeeded(assessmentResult, allBreakdowns);

if (needsReview.needed)

{

await FlagForTeacherReviewAsync(assessmentResult.Id, needsReview.reason);

}

// Update submission

submission.Score = assessmentResult.AIScore;

submission.Status = needsReview.needed ? SubmissionStatus.Submitted : SubmissionStatus.Graded;

submission.IsAutoGraded = true;

await \_context.SaveChangesAsync();

// Notify if teacher review is needed

if (needsReview.needed)

{

await NotifyTeacherForReviewAsync(submission.Assignment.TeacherId, assessmentResult.Id);

}

return assessmentResult;

}

catch (Exception ex)

{

assessmentResult.Status = AIAssessmentStatus.Failed;

assessmentResult.AIFeedback = $"Assessment failed: {ex.Message}";

await \_context.SaveChangesAsync();

return assessmentResult;

}

}

public async Task<List<AIAssessmentResult>> BatchAssessSubmissionsAsync(List<int> submissionIds)

{

var results = new List<AIAssessmentResult>();

// Process in batches to avoid overwhelming the system

const int batchSize = 10;

for (int i = 0; i < submissionIds.Count; i += batchSize)

{

var batch = submissionIds.Skip(i).Take(batchSize);

var batchTasks = batch.Select(AssessSubmissionAsync);

var batchResults = await Task.WhenAll(batchTasks);

results.AddRange(batchResults.Where(r => r != null));

}

return results;

}

public async Task<AIAssessmentResult> ProcessHandwrittenAnswerAsync(int submissionAnswerId)

{

var answer = await \_context.SubmissionAnswers

.Include(a => a.Question)

.Include(a => a.Submission)

.ThenInclude(s => s.Student)

.FirstOrDefaultAsync(a => a.Id == submissionAnswerId);

if (answer == null || string.IsNullOrEmpty(answer.HandwrittenImagePath))

return null;

var assessmentResult = new AIAssessmentResult

{

SubmissionAnswerId = submissionAnswerId,

Status = AIAssessmentStatus.Processing,

ProcessedAt = DateTime.UtcNow

};

\_context.AIAssessmentResults.Add(assessmentResult);

await \_context.SaveChangesAsync();

try

{

// Step 1: Handwriting Recognition

var recognizedText = await \_handwritingService.RecognizeHandwritingAsync(

answer.HandwrittenImagePath,

answer.Submission.StudentId);

assessmentResult.HandwritingRecognitionText = recognizedText;

// Calculate handwriting confidence

var handwritingConfidence = await CalculateHandwritingConfidenceAsync(

answer.HandwrittenImagePath, recognizedText);

assessmentResult.HandwritingConfidence = handwritingConfidence;

// Step 2: Content Assessment

var contentAssessment = await AssessAnswerContentAsync(

recognizedText,

answer.Question.CorrectAnswer,

answer.Question.Type,

answer.Question.Marks);

assessmentResult.AIScore = contentAssessment.Score;

assessmentResult.ConfidenceLevel = Math.Min(handwritingConfidence, contentAssessment.Confidence);

assessmentResult.AIFeedback = contentAssessment.Feedback;

assessmentResult.ScoreBreakdowns = contentAssessment.Breakdowns;

// Update the original answer

answer.ProcessedText = recognizedText;

answer.Score = contentAssessment.Score;

answer.Confidence = assessmentResult.ConfidenceLevel;

answer.IsCorrect = contentAssessment.Score >= (answer.Question.Marks \* 0.7m);

// Determine if manual review is needed

var needsReview = handwritingConfidence < 0.8m || contentAssessment.Confidence < 0.8m;

if (needsReview)

{

await FlagForTeacherReviewAsync(assessmentResult.Id,

$"Low confidence: Handwriting={handwritingConfidence:P}, Content={contentAssessment.Confidence:P}");

}

assessmentResult.Status = AIAssessmentStatus.Completed;

await \_context.SaveChangesAsync();

return assessmentResult;

}

catch (Exception ex)

{

assessmentResult.Status = AIAssessmentStatus.Failed;

assessmentResult.AIFeedback = $"Processing failed: {ex.Message}";

await \_context.SaveChangesAsync();

return assessmentResult;

}

}

public async Task<bool> FlagForTeacherReviewAsync(int assessmentResultId, string reason)

{

var assessmentResult = await \_context.AIAssessmentResults.FindAsync(assessmentResultId);

if (assessmentResult == null) return false;

assessmentResult.RequiresTeacherReview = true;

assessmentResult.ReviewReason = reason;

assessmentResult.Status = AIAssessmentStatus.PendingReview;

await \_context.SaveChangesAsync();

return true;

}

public async Task<List<AIAssessmentResult>> GetPendingReviewsAsync(int teacherId)

{

return await \_context.AIAssessmentResults

.Where(ar => ar.RequiresTeacherReview

&& ar.Status == AIAssessmentStatus.PendingReview

&& (ar.Submission.Assignment.TeacherId == teacherId ||

ar.SubmissionAnswer.Submission.Assignment.TeacherId == teacherId))

.Include(ar => ar.Submission)

.ThenInclude(s => s.Student)

.Include(ar => ar.Submission)

.ThenInclude(s => s.Assignment)

.Include(ar => ar.SubmissionAnswer)

.ThenInclude(sa => sa.Question)

.Include(ar => ar.ScoreBreakdowns)

.OrderByDescending(ar => ar.ProcessedAt)

.ToListAsync();

}

public async Task<AIAssessmentResult> TeacherVerifyAssessmentAsync(int assessmentResultId, int teacherId, TeacherVerificationDto verification)

{

var assessmentResult = await \_context.AIAssessmentResults

.Include(ar => ar.ScoreBreakdowns)

.Include(ar => ar.Submission)

.ThenInclude(s => s.Assignment)

.FirstOrDefaultAsync(ar => ar.Id == assessmentResultId);

if (assessmentResult == null) return null;

// Record teacher verification

assessmentResult.VerifiedByTeacherId = teacherId;

assessmentResult.TeacherScore = verification.OverrideScore ?? assessmentResult.AIScore;

assessmentResult.TeacherFeedback = verification.TeacherFeedback;

assessmentResult.VerifiedAt = DateTime.UtcNow;

assessmentResult.VerificationStatus = verification.VerificationStatus;

assessmentResult.Status = AIAssessmentStatus.Verified;

assessmentResult.RequiresTeacherReview = false;

// Update criterion scores if provided

foreach (var criterionVerification in verification.CriterionVerifications)

{

var breakdown = assessmentResult.ScoreBreakdowns

.FirstOrDefault(b => b.Criterion == criterionVerification.Criterion);

if (breakdown != null)

{

breakdown.Score = criterionVerification.TeacherScore;

breakdown.Explanation = criterionVerification.TeacherComment;

}

}

// Update the associated submission/answer

if (assessmentResult.SubmissionId.HasValue)

{

var submission = assessmentResult.Submission;

submission.Score = assessmentResult.TeacherScore;

submission.Status = SubmissionStatus.Graded;

submission.Feedback = verification.TeacherFeedback;

}

else if (assessmentResult.SubmissionAnswerId.HasValue)

{

var answer = await \_context.SubmissionAnswers.FindAsync(assessmentResult.SubmissionAnswerId);

if (answer != null)

{

answer.Score = assessmentResult.TeacherScore;

answer.IsCorrect = assessmentResult.TeacherScore >= (answer.Question.Marks \* 0.7m);

}

}

await \_context.SaveChangesAsync();

// Learn from teacher corrections for future AI improvements

await RecordTeacherCorrectionForLearningAsync(assessmentResult);

return assessmentResult;

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessIndividualAnswerAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

switch (question.Type)

{

case QuestionType.MultipleChoice:

return await AssessMultipleChoiceAsync(answer, question);

case QuestionType.TrueFalse:

return await AssessTrueFalseAsync(answer, question);

case QuestionType.ShortAnswer:

return await AssessShortAnswerAsync(answer, question);

case QuestionType.Essay:

return await AssessEssayAsync(answer, question);

case QuestionType.Handwritten:

if (!string.IsNullOrEmpty(answer.HandwrittenImagePath))

{

var handwrittenResult = await ProcessHandwrittenAnswerAsync(answer.Id);

return (handwrittenResult?.AIScore ?? 0,

handwrittenResult?.ConfidenceLevel ?? 0,

handwrittenResult?.AIFeedback ?? "Processing failed",

handwrittenResult?.ScoreBreakdowns ?? new List<AIScoreBreakdown>());

}

return await AssessShortAnswerAsync(answer, question);

default:

return (0, 0, "Unknown question type", breakdowns);

}

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessMultipleChoiceAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var correctOption = question.Options.FirstOrDefault(o => o.IsCorrect);

var isCorrect = correctOption != null &&

string.Equals(answer.Answer, correctOption.OptionLetter.ToString(), StringComparison.OrdinalIgnoreCase);

var score = isCorrect ? question.Marks : 0;

var feedback = isCorrect ? "Correct answer" : $"Incorrect. The correct answer is {correctOption?.OptionLetter}";

var breakdown = new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = score,

MaxScore = question.Marks,

Explanation = feedback

};

return (score, 1.0m, feedback, new List<AIScoreBreakdown> { breakdown });

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessTrueFalseAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var isCorrect = string.Equals(answer.Answer, question.CorrectAnswer, StringComparison.OrdinalIgnoreCase);

var score = isCorrect ? question.Marks : 0;

var feedback = isCorrect ? "Correct answer" : $"Incorrect. The correct answer is {question.CorrectAnswer}";

var breakdown = new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = score,

MaxScore = question.Marks,

Explanation = feedback

};

return (score, 1.0m, feedback, new List<AIScoreBreakdown> { breakdown });

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessShortAnswerAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

// Assess accuracy

var accuracy = CalculateTextSimilarity(answer.Answer, question.CorrectAnswer);

var accuracyScore = question.Marks \* accuracy;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Accuracy",

Score = accuracyScore,

MaxScore = question.Marks,

Explanation = $"Answer similarity to expected response: {accuracy:P}"

});

var totalScore = accuracyScore;

var confidence = accuracy > 0.6m ? 0.9m : 0.7m; // Lower confidence for low similarity

var feedback = accuracy switch

{

>= 0.9m => "Excellent answer, very close to expected response",

>= 0.7m => "Good answer, mostly correct",

>= 0.5m => "Partially correct, but missing some key points",

\_ => "Answer needs improvement, significantly different from expected response"

};

return (totalScore, confidence, feedback, breakdowns);

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessEssayAsync(

SubmissionAnswer answer, AssignmentQuestion question)

{

var breakdowns = new List<AIScoreBreakdown>();

// This would ideally use advanced NLP/AI services like OpenAI GPT or Azure Cognitive Services

// For now, implementing basic assessment criteria

// Content relevance (40% of marks)

var contentScore = await AssessContentRelevanceAsync(answer.Answer, question.CorrectAnswer);

var contentMarks = question.Marks \* 0.4m \* contentScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Content Relevance",

Score = contentMarks,

MaxScore = question.Marks \* 0.4m,

Explanation = $"Content relevance score: {contentScore:P}"

});

// Grammar and language (30% of marks)

var grammarScore = await AssessGrammarAsync(answer.Answer);

var grammarMarks = question.Marks \* 0.3m \* grammarScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Grammar & Language",

Score = grammarMarks,

MaxScore = question.Marks \* 0.3m,

Explanation = $"Grammar and language quality: {grammarScore:P}"

});

// Structure and organization (30% of marks)

var structureScore = await AssessStructureAsync(answer.Answer);

var structureMarks = question.Marks \* 0.3m \* structureScore;

breakdowns.Add(new AIScoreBreakdown

{

Criterion = "Structure & Organization",

Score = structureMarks,

MaxScore = question.Marks \* 0.3m,

Explanation = $"Structure and organization: {structureScore:P}"

});

var totalScore = contentMarks + grammarMarks + structureMarks;

var averageScore = (contentScore + grammarScore + structureScore) / 3;

// Lower confidence for essays as they're more subjective

var confidence = averageScore > 0.7m ? 0.75m : 0.6m;

var feedback = GenerateEssayFeedback(contentScore, grammarScore, structureScore);

return (totalScore, confidence, feedback, breakdowns);

}

private async Task<(decimal Score, decimal Confidence, string Feedback, List<AIScoreBreakdown> Breakdowns)> AssessAnswerContentAsync(

string recognizedText, string correctAnswer, QuestionType questionType, decimal maxMarks)

{

// Use the appropriate assessment method based on question type

var dummyAnswer = new SubmissionAnswer { Answer = recognizedText };

var dummyQuestion = new AssignmentQuestion

{

CorrectAnswer = correctAnswer,

Type = questionType,

Marks = maxMarks

};

return questionType switch

{

QuestionType.ShortAnswer => await AssessShortAnswerAsync(dummyAnswer, dummyQuestion),

QuestionType.Essay => await AssessEssayAsync(dummyAnswer, dummyQuestion),

\_ => await AssessShortAnswerAsync(dummyAnswer, dummyQuestion)

};

}

private decimal CalculateTextSimilarity(string text1, string text2)

{

if (string.IsNullOrEmpty(text1) || string.IsNullOrEmpty(text2)) return 0;

// Normalize texts

text1 = text1.ToLower().Trim();

text2 = text2.ToLower().Trim();

// Simple word-based similarity

var words1 = text1.Split(' ', StringSplitOptions.RemoveEmptyEntries);

var words2 = text2.Split(' ', StringSplitOptions.RemoveEmptyEntries);

var commonWords = words1.Intersect(words2).Count();

var totalWords = Math.Max(words1.Length, words2.Length);

return totalWords > 0 ? (decimal)commonWords / totalWords : 0;

}

private async Task<decimal> CalculateHandwritingConfidenceAsync(string imagePath, string recognizedText)

{

// This would use image quality metrics and OCR confidence scores

// For now, return a simulated confidence based on text length and clarity

if (string.IsNullOrEmpty(recognizedText)) return 0;

// Simulate confidence calculation

var baseConfidence = 0.8m;

var lengthFactor = Math.Min(recognizedText.Length / 50m, 1m); // Longer text = higher confidence

var clarityFactor = recognizedText.Count(char.IsLetter) / (decimal)recognizedText.Length;

return Math.Min(baseConfidence \* lengthFactor \* clarityFactor, 1m);

}

private async Task<decimal> AssessContentRelevanceAsync(string studentAnswer, string expectedAnswer)

{

return CalculateTextSimilarity(studentAnswer, expectedAnswer);

}

private async Task<decimal> AssessGrammarAsync(string text)

{

if (string.IsNullOrEmpty(text)) return 0;

// Basic grammar assessment

var sentences = text.Split('.', '!', '?').Where(s => !string.IsNullOrWhiteSpace(s)).ToArray();

if (sentences.Length == 0) return 0;

var grammarScore = 0.8m; // Base score

// Simple checks

var hasCapitalizedSentences = sentences.Count(s => char.IsUpper(s.Trim().FirstOrDefault())) / (decimal)sentences.Length;

var hasProperPunctuation = (text.Count(c => ".!?".Contains(c)) >= sentences.Length) ? 1m : 0.7m;

return (grammarScore + hasCapitalizedSentences + hasProperPunctuation) / 3;

}

private async Task<decimal> AssessStructureAsync(string text)

{

if (string.IsNullOrEmpty(text)) return 0;

var sentences = text.Split('.', '!', '?').Where(s => !string.IsNullOrWhiteSpace(s)).Count();

var paragraphs = text.Split('\n').Where(p => !string.IsNullOrWhiteSpace(p)).Count();

// Basic structure scoring

var structureScore = 0.7m; // Base score

if (sentences >= 3) structureScore += 0.2m; // Has multiple sentences

if (paragraphs >= 2) structureScore += 0.1m; // Has multiple paragraphs

return Math.Min(structureScore, 1m);

}

private string GenerateEssayFeedback(decimal contentScore, decimal grammarScore, decimal structureScore)

{

var feedback = new List<string>();

if (contentScore >= 0.8m) feedback.Add("Excellent content relevance and understanding");

else if (contentScore >= 0.6m) feedback.Add("Good content but could be more detailed");

else feedback.Add("Content needs improvement - ensure you address all key points");

if (grammarScore >= 0.8m) feedback.Add("Good grammar and language use");

else feedback.Add("Pay attention to grammar, spelling, and sentence structure");

if (structureScore >= 0.8m) feedback.Add("Well-organized response");

else feedback.Add("Work on organizing your thoughts into clear paragraphs");

return string.Join(". ", feedback);

}

private string GenerateOverallFeedback(List<AIScoreBreakdown> breakdowns, decimal overallScore)

{

if (overallScore >= 80) return "Excellent work! You have demonstrated strong understanding.";

if (overallScore >= 70) return "Good work! You're on the right track with room for improvement.";

if (overallScore >= 60) return "Fair performance. Focus on improving weaker areas.";

if (overallScore >= 50) return "Below average performance. Consider reviewing the material.";

return "Needs significant improvement. Please seek additional help.";

}

private (bool needed, string reason) DetermineIfTeacherReviewNeeded(AIAssessmentResult assessment, List<AIScoreBreakdown> breakdowns)

{

// Flag for review if confidence is low

if (assessment.ConfidenceLevel < 0.7m)

return (true, $"Low AI confidence: {assessment.Conf

Second Version

// Models/Entities/BaseEntity.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.Entities

{

public abstract class BaseEntity

{

[Key]

public Guid Id { get; set; } = Guid.NewGuid();

public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;

public string CreatedBy { get; set; } = string.Empty;

public string UpdatedBy { get; set; } = string.Empty;

public bool IsDeleted { get; set; } = false;

}

}

// Models/Entities/School.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.Entities

{

public class School : BaseEntity

{

[Required]

[StringLength(200)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Address { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(100)]

public string Website { get; set; } = string.Empty;

[StringLength(50)]

public string RegistrationNumber { get; set; } = string.Empty;

public string Logo { get; set; } = string.Empty;

// Navigation Properties

public virtual ICollection<Teacher> Teachers { get; set; } = new List<Teacher>();

public virtual ICollection<Student> Students { get; set; } = new List<Student>();

public virtual ICollection<Grade> Grades { get; set; } = new List<Grade>();

public virtual ICollection<Subject> Subjects { get; set; } = new List<Subject>();

public virtual ICollection<SchoolYear> SchoolYears { get; set; } = new List<SchoolYear>();

public virtual ICollection<Timetable> Timetables { get; set; } = new List<Timetable>();

// Grading Scheme Reference

public Guid? GradingSchemeId { get; set; }

public virtual GradingScheme? GradingScheme { get; set; }

}

}

// Models/Entities/GradingScheme.cs

namespace SchoolManagement.Core.Entities

{

public class GradingScheme : BaseEntity

{

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Description { get; set; } = string.Empty;

[StringLength(50)]

public string Country { get; set; } = string.Empty;

public bool IsActive { get; set; } = true;

// Navigation Properties

public virtual ICollection<GradingScale> GradingScales { get; set; } = new List<GradingScale>();

public virtual ICollection<School> Schools { get; set; } = new List<School>();

}

}

// Models/Entities/GradingScale.cs

namespace SchoolManagement.Core.Entities

{

public class GradingScale : BaseEntity

{

public Guid GradingSchemeId { get; set; }

public virtual GradingScheme GradingScheme { get; set; } = null!;

[StringLength(10)]

public string Symbol { get; set; } = string.Empty; // A, B, C, D, etc.

public int Unit { get; set; } // 1, 2, 3, 4, etc.

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

[StringLength(100)]

public string Description { get; set; } = string.Empty; // Excellent, Good, Fair, etc.

public int SortOrder { get; set; }

}

}

// Models/Entities/Teacher.cs

namespace SchoolManagement.Core.Entities

{

public class Teacher : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[StringLength(50)]

public string EmployeeId { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public DateTime HireDate { get; set; }

[StringLength(100)]

public string Qualification { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

// Navigation Properties

public virtual ICollection<SubjectTeacher> SubjectTeachers { get; set; } = new List<SubjectTeacher>();

public virtual ICollection<ClassTeacher> ClassTeachers { get; set; } = new List<ClassTeacher>();

public virtual ICollection<Attendance> AttendanceRecords { get; set; } = new List<Attendance>();

public virtual ICollection<Result> Results { get; set; } = new List<Result>();

public virtual ICollection<TimetableSlot> TimetableSlots { get; set; } = new List<TimetableSlot>();

}

}

// Models/Entities/Student.cs

namespace SchoolManagement.Core.Entities

{

public class Student : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[StringLength(50)]

public string StudentNumber { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

[StringLength(10)]

public string Gender { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

// Current Grade

public Guid CurrentGradeId { get; set; }

public virtual Grade CurrentGrade { get; set; } = null!;

// Navigation Properties

public virtual ICollection<Parent> Parents { get; set; } = new List<Parent>();

public virtual ICollection<StudentSubject> StudentSubjects { get; set; } = new List<StudentSubject>();

public virtual ICollection<Attendance> AttendanceRecords { get; set; } = new List<Attendance>();

public virtual ICollection<Result> Results { get; set; } = new List<Result>();

public virtual ICollection<HandwritingTrainingData> HandwritingTrainingData { get; set; } = new List<HandwritingTrainingData>();

public virtual ICollection<StudentTimetable> StudentTimetables { get; set; } = new List<StudentTimetable>();

}

}

// Models/Entities/Parent.cs

namespace SchoolManagement.Core.Entities

{

public class Parent : BaseEntity

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(20)]

public string WhatsAppNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[StringLength(50)]

public string Relationship { get; set; } = string.Empty; // Father, Mother, Guardian

[StringLength(100)]

public string Occupation { get; set; } = string.Empty;

// Navigation Properties

public virtual ICollection<Student> Students { get; set; } = new List<Student>();

public virtual ICollection<Notification> Notifications { get; set; } = new List<Notification>();

}

}

// Models/Entities/Grade.cs

namespace SchoolManagement.Core.Entities

{

public class Grade : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

[Required]

[StringLength(50)]

public string Name { get; set; } = string.Empty; // Grade 1, Form 1, Year 7, etc.

[StringLength(200)]

public string Description { get; set; } = string.Empty;

public int Level { get; set; } // 1, 2, 3, etc. for ordering

public int MaxStudents { get; set; } = 50;

// Navigation Properties

public virtual ICollection<Student> Students { get; set; } = new List<Student>();

public virtual ICollection<GradeSubject> GradeSubjects { get; set; } = new List<GradeSubject>();

public virtual ICollection<ClassTeacher> ClassTeachers { get; set; } = new List<ClassTeacher>();

public virtual ICollection<Timetable> Timetables { get; set; } = new List<Timetable>();

}

}

// Models/Entities/Subject.cs

namespace SchoolManagement.Core.Entities

{

public class Subject : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

[StringLength(10)]

public string Code { get; set; } = string.Empty;

[StringLength(500)]

public string Description { get; set; } = string.Empty;

public int Credits { get; set; } = 1;

public bool IsActive { get; set; } = true;

// Navigation Properties

public virtual ICollection<GradeSubject> GradeSubjects { get; set; } = new List<GradeSubject>();

public virtual ICollection<SubjectTeacher> SubjectTeachers { get; set; } = new List<SubjectTeacher>();

public virtual ICollection<StudentSubject> StudentSubjects { get; set; } = new List<StudentSubject>();

public virtual ICollection<Result> Results { get; set; } = new List<Result>();

public virtual ICollection<TimetableSlot> TimetableSlots { get; set; } = new List<TimetableSlot>();

}

}

// Models/Entities/GradeSubject.cs

namespace SchoolManagement.Core.Entities

{

public class GradeSubject : BaseEntity

{

public Guid GradeId { get; set; }

public virtual Grade Grade { get; set; } = null!;

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public bool IsCompulsory { get; set; } = true;

public int SortOrder { get; set; }

}

}

// Models/Entities/SubjectTeacher.cs

namespace SchoolManagement.Core.Entities

{

public class SubjectTeacher : BaseEntity

{

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public Guid TeacherId { get; set; }

public virtual Teacher Teacher { get; set; } = null!;

public Guid GradeId { get; set; }

public virtual Grade Grade { get; set; } = null!;

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

public bool IsActive { get; set; } = true;

}

}

// Models/Entities/StudentSubject.cs

namespace SchoolManagement.Core.Entities

{

public class StudentSubject : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

public DateTime EnrollmentDate { get; set; } = DateTime.UtcNow;

public bool IsActive { get; set; } = true;

}

}

// Models/Entities/ClassTeacher.cs

namespace SchoolManagement.Core.Entities

{

public class ClassTeacher : BaseEntity

{

public Guid TeacherId { get; set; }

public virtual Teacher Teacher { get; set; } = null!;

public Guid GradeId { get; set; }

public virtual Grade Grade { get; set; } = null!;

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

public bool IsActive { get; set; } = true;

}

}

// Models/Entities/SchoolYear.cs

namespace SchoolManagement.Core.Entities

{

public class SchoolYear : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

[Required]

[StringLength(50)]

public string Name { get; set; } = string.Empty; // 2024/2025

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; } = true;

public bool IsCurrent { get; set; } = false;

// Navigation Properties

public virtual ICollection<Term> Terms { get; set; } = new List<Term>();

public virtual ICollection<SubjectTeacher> SubjectTeachers { get; set; } = new List<SubjectTeacher>();

public virtual ICollection<StudentSubject> StudentSubjects { get; set; } = new List<StudentSubject>();

public virtual ICollection<ClassTeacher> ClassTeachers { get; set; } = new List<ClassTeacher>();

public virtual ICollection<Result> Results { get; set; } = new List<Result>();

}

}

// Models/Entities/Term.cs

namespace SchoolManagement.Core.Entities

{

public class Term : BaseEntity

{

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

[Required]

[StringLength(50)]

public string Name { get; set; } = string.Empty; // Term 1, Semester 1, etc.

public int TermNumber { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; } = true;

public bool IsCurrent { get; set; } = false;

// Navigation Properties

public virtual ICollection<Result> Results { get; set; } = new List<Result>();

public virtual ICollection<Attendance> AttendanceRecords { get; set; } = new List<Attendance>();

}

}

// Models/Entities/Attendance.cs

namespace SchoolManagement.Core.Entities

{

public class Attendance : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public Guid TeacherId { get; set; }

public virtual Teacher Teacher { get; set; } = null!;

public Guid TermId { get; set; }

public virtual Term Term { get; set; } = null!;

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

[StringLength(200)]

public string Remarks { get; set; } = string.Empty;

}

public enum AttendanceStatus

{

Present,

Absent,

Late,

Excused,

Sick

}

}

// Models/Entities/Result.cs

namespace SchoolManagement.Core.Entities

{

public class Result : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public Guid TeacherId { get; set; }

public virtual Teacher Teacher { get; set; } = null!;

public Guid TermId { get; set; }

public virtual Term Term { get; set; } = null!;

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

public decimal Score { get; set; }

public decimal TotalMarks { get; set; }

public decimal Percentage { get; set; }

[StringLength(10)]

public string Grade { get; set; } = string.Empty; // A, B, C, etc.

public int GradeUnit { get; set; } // 1, 2, 3, etc.

[StringLength(50)]

public string AssessmentType { get; set; } = string.Empty; // Test, Exam, Assignment

[StringLength(500)]

public string Comments { get; set; } = string.Empty;

public DateTime AssessmentDate { get; set; }

// AI Answer Verification

public bool IsAiVerified { get; set; } = false;

public decimal AiConfidenceScore { get; set; }

public string AiProcessedAnswers { get; set; } = string.Empty; // JSON

}

}

// Models/Entities/Notification.cs

namespace SchoolManagement.Core.Entities

{

public class Notification : BaseEntity

{

public Guid? ParentId { get; set; }

public virtual Parent? Parent { get; set; }

public Guid? StudentId { get; set; }

public virtual Student? Student { get; set; }

public Guid? TeacherId { get; set; }

public virtual Teacher? Teacher { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[Required]

public string Message { get; set; } = string.Empty;

public NotificationType Type { get; set; }

public NotificationChannel Channel { get; set; }

public bool IsRead { get; set; } = false;

public bool IsSent { get; set; } = false;

public DateTime? SentAt { get; set; }

public DateTime? ReadAt { get; set; }

[StringLength(100)]

public string ExternalId { get; set; } = string.Empty; // For WhatsApp, SMS tracking

public string Metadata { get; set; } = string.Empty; // JSON for additional data

}

public enum NotificationType

{

General,

Attendance,

Results,

Fee,

Event,

Emergency,

Assignment

}

public enum NotificationChannel

{

InApp,

Email,

SMS,

WhatsApp,

Push

}

}

// Models/Entities/Timetable.cs

namespace SchoolManagement.Core.Entities

{

public class Timetable : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

public Guid GradeId { get; set; }

public virtual Grade Grade { get; set; } = null!;

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

public Guid TermId { get; set; }

public virtual Term Term { get; set; } = null!;

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

public bool IsActive { get; set; } = true;

public bool IsGenerated { get; set; } = false;

public DateTime GeneratedAt { get; set; }

// Navigation Properties

public virtual ICollection<TimetableSlot> TimetableSlots { get; set; } = new List<TimetableSlot>();

}

}

// Models/Entities/TimetableSlot.cs

namespace SchoolManagement.Core.Entities

{

public class TimetableSlot : BaseEntity

{

public Guid TimetableId { get; set; }

public virtual Timetable Timetable { get; set; } = null!;

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public Guid TeacherId { get; set; }

public virtual Teacher Teacher { get; set; } = null!;

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

[StringLength(50)]

public string Room { get; set; } = string.Empty;

public int PeriodNumber { get; set; }

[StringLength(200)]

public string Notes { get; set; } = string.Empty;

}

}

// Models/Entities/StudentTimetable.cs

namespace SchoolManagement.Core.Entities

{

public class StudentTimetable : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid TimetableSlotId { get; set; }

public virtual TimetableSlot TimetableSlot { get; set; } = null!;

public bool IsActive { get; set; } = true;

// For individual student customizations

[StringLength(200)]

public string CustomNotes { get; set; } = string.Empty;

}

}// Models/Entities/AI/HandwritingTrainingData.cs

namespace SchoolManagement.Core.Entities

{

public class HandwritingTrainingData : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid? TeacherId { get; set; }

public virtual Teacher? Teacher { get; set; }

public TrainingDataType DataType { get; set; }

[Required]

public string OriginalContent { get; set; } = string.Empty; // What the student was supposed to write

[Required]

public string ImagePath { get; set; } = string.Empty; // Path to handwritten image

public string ProcessedText { get; set; } = string.Empty; // AI interpreted text

public string VerifiedText { get; set; } = string.Empty; // Teacher verified text

public decimal ConfidenceScore { get; set; }

public TrainingStatus Status { get; set; } = TrainingStatus.Pending;

public bool IsVerified { get; set; } = false;

public bool IsTrainingComplete { get; set; } = false;

public DateTime CapturedAt { get; set; } = DateTime.UtcNow;

public DateTime? VerifiedAt { get; set; }

public DateTime? TrainedAt { get; set; }

// Metadata for training

public string BoundingBoxes { get; set; } = string.Empty; // JSON array of character bounding boxes

public string CharacterData { get; set; } = string.Empty; // JSON array of individual character data

public string ModelVersion { get; set; } = string.Empty;

// Navigation Properties

public virtual ICollection<HandwritingCharacterData> CharacterData { get; set; } = new List<HandwritingCharacterData>();

}

public enum TrainingDataType

{

Alphabet,

Numbers,

Words,

Sentences,

MathExpressions,

Answers

}

public enum TrainingStatus

{

Pending,

Processing,

NeedsVerification,

Verified,

TrainingQueued,

Training,

Complete,

Failed

}

}

// Models/Entities/AI/HandwritingCharacterData.cs

namespace SchoolManagement.Core.Entities

{

public class HandwritingCharacterData : BaseEntity

{

public Guid HandwritingTrainingDataId { get; set; }

public virtual HandwritingTrainingData HandwritingTrainingData { get; set; } = null!;

[Required]

public string Character { get; set; } = string.Empty; // The actual character (A, B, 1, 2, etc.)

[Required]

public string ImageSegmentPath { get; set; } = string.Empty; // Path to individual character image

// Bounding box coordinates

public int X { get; set; }

public int Y { get; set; }

public int Width { get; set; }

public int Height { get; set; }

public decimal ConfidenceScore { get; set; }

public bool IsVerified { get; set; } = false;

// Feature vectors for ML model

public string FeatureVector { get; set; } = string.Empty; // JSON array of features

public int SequenceOrder { get; set; } // Order in the original text

}

}

// Models/Entities/AI/HandwritingModel.cs

namespace SchoolManagement.Core.Entities

{

public class HandwritingModel : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

[Required]

[StringLength(100)]

public string ModelName { get; set; } = string.Empty;

[Required]

[StringLength(50)]

public string Version { get; set; } = string.Empty;

public string ModelFilePath { get; set; } = string.Empty; // Local device model path

public string CloudModelPath { get; set; } = string.Empty; // Cloud backup path

public ModelType Type { get; set; }

public decimal Accuracy { get; set; }

public int TrainingDataCount { get; set; }

public bool IsActive { get; set; } = true;

public bool IsDeployedLocally { get; set; } = false;

public bool IsDeployedCloud { get; set; } = false;

public DateTime TrainingStarted { get; set; }

public DateTime? TrainingCompleted { get; set; }

public DateTime? LastUsed { get; set; }

// Model metadata

public string TrainingParameters { get; set; } = string.Empty; // JSON

public string PerformanceMetrics { get; set; } = string.Empty; // JSON

// Navigation Properties

public virtual ICollection<HandwritingRecognitionResult> RecognitionResults { get; set; } = new List<HandwritingRecognitionResult>();

}

public enum ModelType

{

Personal, // Individual student model

Grade, // Grade-level model

School, // School-wide model

Global // System-wide model

}

}

// Models/Entities/AI/HandwritingRecognitionResult.cs

namespace SchoolManagement.Core.Entities

{

public class HandwritingRecognitionResult : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid? SubjectId { get; set; }

public virtual Subject? Subject { get; set; }

public Guid? ResultId { get; set; }

public virtual Result? Result { get; set; }

public Guid HandwritingModelId { get; set; }

public virtual HandwritingModel HandwritingModel { get; set; } = null!;

[Required]

public string OriginalImagePath { get; set; } = string.Empty;

[Required]

public string RecognizedText { get; set; } = string.Empty;

public decimal OverallConfidence { get; set; }

public bool ProcessedLocally { get; set; } = true;

public bool ProcessedInCloud { get; set; } = false;

public TimeSpan ProcessingTime { get; set; }

public string CharacterConfidences { get; set; } = string.Empty; // JSON array

public string AlternativeTexts { get; set; } = string.Empty; // JSON array of alternatives

// For answer verification

public string ExpectedAnswer { get; set; } = string.Empty;

public bool IsCorrect { get; set; }

public decimal PartialCreditScore { get; set; }

public DateTime ProcessedAt { get; set; } = DateTime.UtcNow;

}

}

// Models/Entities/AI/TrainingExercise.cs

namespace SchoolManagement.Core.Entities

{

public class TrainingExercise : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

public Guid? GradeId { get; set; }

public virtual Grade? Grade { get; set; }

public Guid? SubjectId { get; set; }

public virtual Subject? Subject { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

public ExerciseType Type { get; set; }

public DifficultyLevel Difficulty { get; set; }

[Required]

public string Content { get; set; } = string.Empty; // JSON containing exercise data

public bool IsActive { get; set; } = true;

public bool IsSystemGenerated { get; set; } = false;

public int EstimatedMinutes { get; set; } = 10;

// Navigation Properties

public virtual ICollection<StudentTrainingSession> StudentSessions { get; set; } = new List<StudentTrainingSession>();

}

public enum ExerciseType

{

Alphabet,

Numbers,

Words,

Sentences,

MathProblems,

CopyText,

Dictation,

FreeWriting

}

public enum DifficultyLevel

{

Beginner,

Elementary,

Intermediate,

Advanced,

Expert

}

}

// Models/Entities/AI/StudentTrainingSession.cs

namespace SchoolManagement.Core.Entities

{

public class StudentTrainingSession : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid TrainingExerciseId { get; set; }

public virtual TrainingExercise TrainingExercise { get; set; } = null!;

public DateTime StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public SessionStatus Status { get; set; } = SessionStatus.InProgress;

public int TotalItems { get; set; }

public int CompletedItems { get; set; }

public int CorrectItems { get; set; }

public decimal OverallAccuracy { get; set; }

public TimeSpan TotalTime { get; set; }

// Navigation Properties

public virtual ICollection<HandwritingTrainingData> TrainingData { get; set; } = new List<HandwritingTrainingData>();

}

public enum SessionStatus

{

InProgress,

Completed,

Paused,

Abandoned,

NeedsReview

}

}

// Models/Entities/AI/ModelTrainingJob.cs

namespace SchoolManagement.Core.Entities

{

public class ModelTrainingJob : BaseEntity

{

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

[Required]

[StringLength(100)]

public string JobName { get; set; } = string.Empty;

public JobType Type { get; set; }

public JobStatus Status { get; set; } = JobStatus.Queued;

public int TotalDataPoints { get; set; }

public int ProcessedDataPoints { get; set; }

public decimal Progress { get; set; }

public DateTime QueuedAt { get; set; } = DateTime.UtcNow;

public DateTime? StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public string ErrorMessage { get; set; } = string.Empty;

public string ResultPath { get; set; } = string.Empty;

// Training parameters

public string TrainingParameters { get; set; } = string.Empty; // JSON

// Navigation Properties

public virtual HandwritingModel? ResultingModel { get; set; }

}

public enum JobType

{

InitialTraining,

IncrementalTraining,

ModelUpdate,

ModelValidation

}

public enum JobStatus

{

Queued,

Running,

Completed,

Failed,

Cancelled

}

}// Data/SchoolManagementDbContext.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Data

{

public class SchoolManagementDbContext : DbContext

{

public SchoolManagementDbContext(DbContextOptions<SchoolManagementDbContext> options)

: base(options)

{

}

// Core Entities

public DbSet<School> Schools { get; set; }

public DbSet<GradingScheme> GradingSchemes { get; set; }

public DbSet<GradingScale> GradingScales { get; set; }

public DbSet<Teacher> Teachers { get; set; }

public DbSet<Student> Students { get; set; }

public DbSet<Parent> Parents { get; set; }

public DbSet<Grade> Grades { get; set; }

public DbSet<Subject> Subjects { get; set; }

public DbSet<SchoolYear> SchoolYears { get; set; }

public DbSet<Term> Terms { get; set; }

// Relationship Entities

public DbSet<GradeSubject> GradeSubjects { get; set; }

public DbSet<SubjectTeacher> SubjectTeachers { get; set; }

public DbSet<StudentSubject> StudentSubjects { get; set; }

public DbSet<ClassTeacher> ClassTeachers { get; set; }

// Academic Records

public DbSet<Attendance> Attendances { get; set; }

public DbSet<Result> Results { get; set; }

public DbSet<Notification> Notifications { get; set; }

// Timetable

public DbSet<Timetable> Timetables { get; set; }

public DbSet<TimetableSlot> TimetableSlots { get; set; }

public DbSet<StudentTimetable> StudentTimetables { get; set; }

// AI Handwriting Recognition

public DbSet<HandwritingTrainingData> HandwritingTrainingData { get; set; }

public DbSet<HandwritingCharacterData> HandwritingCharacterData { get; set; }

public DbSet<HandwritingModel> HandwritingModels { get; set; }

public DbSet<HandwritingRecognitionResult> HandwritingRecognitionResults { get; set; }

public DbSet<TrainingExercise> TrainingExercises { get; set; }

public DbSet<StudentTrainingSession> StudentTrainingSessions { get; set; }

public DbSet<ModelTrainingJob> ModelTrainingJobs { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

// Configure entity relationships and constraints

ConfigureSchoolEntities(modelBuilder);

ConfigureGradingEntities(modelBuilder);

ConfigureUserEntities(modelBuilder);

ConfigureAcademicEntities(modelBuilder);

ConfigureTimetableEntities(modelBuilder);

ConfigureAIEntities(modelBuilder);

ConfigureIndexes(modelBuilder);

ConfigureSoftDelete(modelBuilder);

}

private void ConfigureSchoolEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<School>(entity =>

{

entity.HasIndex(e => e.RegistrationNumber).IsUnique();

entity.HasIndex(e => e.Email).IsUnique();

entity.Property(e => e.Name).IsRequired();

});

modelBuilder.Entity<SchoolYear>(entity =>

{

entity.HasOne(sy => sy.School)

.WithMany(s => s.SchoolYears)

.HasForeignKey(sy => sy.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Name }).IsUnique();

entity.HasIndex(e => new { e.SchoolId, e.IsCurrent });

});

modelBuilder.Entity<Term>(entity =>

{

entity.HasOne(t => t.SchoolYear)

.WithMany(sy => sy.Terms)

.HasForeignKey(t => t.SchoolYearId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolYearId, e.TermNumber }).IsUnique();

});

}

private void ConfigureGradingEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradingScheme>(entity =>

{

entity.HasIndex(e => new { e.Name, e.Country }).IsUnique();

entity.Property(e => e.Name).IsRequired();

});

modelBuilder.Entity<GradingScale>(entity =>

{

entity.HasOne(gs => gs.GradingScheme)

.WithMany(g => g.GradingScales)

.HasForeignKey(gs => gs.GradingSchemeId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.GradingSchemeId, e.Symbol }).IsUnique();

entity.HasIndex(e => new { e.GradingSchemeId, e.Unit }).IsUnique();

entity.Property(e => e.MinPercentage).HasColumnType("decimal(5,2)");

entity.Property(e => e.MaxPercentage).HasColumnType("decimal(5,2)");

});

modelBuilder.Entity<School>()

.HasOne(s => s.GradingScheme)

.WithMany(gs => gs.Schools)

.HasForeignKey(s => s.GradingSchemeId)

.OnDelete(DeleteBehavior.SetNull);

}

private void ConfigureUserEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Teacher>(entity =>

{

entity.HasOne(t => t.School)

.WithMany(s => s.Teachers)

.HasForeignKey(t => t.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Email }).IsUnique();

entity.HasIndex(e => new { e.SchoolId, e.EmployeeId }).IsUnique();

entity.Property(e => e.FirstName).IsRequired();

entity.Property(e => e.LastName).IsRequired();

});

modelBuilder.Entity<Student>(entity =>

{

entity.HasOne(s => s.School)

.WithMany(sch => sch.Students)

.HasForeignKey(s => s.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(s => s.CurrentGrade)

.WithMany(g => g.Students)

.HasForeignKey(s => s.CurrentGradeId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.SchoolId, e.StudentNumber }).IsUnique();

entity.Property(e => e.FirstName).IsRequired();

entity.Property(e => e.LastName).IsRequired();

// Many-to-many relationship with Parents

entity.HasMany(s => s.Parents)

.WithMany(p => p.Students)

.UsingEntity(j => j.ToTable("StudentParents"));

});

modelBuilder.Entity<Parent>(entity =>

{

entity.HasIndex(e => e.Email).IsUnique();

entity.Property(e => e.FirstName).IsRequired();

entity.Property(e => e.LastName).IsRequired();

});

}

private void ConfigureAcademicEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Grade>(entity =>

{

entity.HasOne(g => g.School)

.WithMany(s => s.Grades)

.HasForeignKey(g => g.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Name }).IsUnique();

entity.HasIndex(e => new { e.SchoolId, e.Level });

entity.Property(e => e.Name).IsRequired();

});

modelBuilder.Entity<Subject>(entity =>

{

entity.HasOne(s => s.School)

.WithMany(sch => sch.Subjects)

.HasForeignKey(s => s.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Code }).IsUnique();

entity.Property(e => e.Name).IsRequired();

});

// Configure relationship entities

ConfigureRelationshipEntities(modelBuilder);

// Configure attendance

modelBuilder.Entity<Attendance>(entity =>

{

entity.HasOne(a => a.Student)

.WithMany(s => s.AttendanceRecords)

.HasForeignKey(a => a.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(a => a.Subject)

.WithMany()

.HasForeignKey(a => a.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(a => a.Teacher)

.WithMany(t => t.AttendanceRecords)

.HasForeignKey(a => a.TeacherId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(a => a.Term)

.WithMany(t => t.AttendanceRecords)

.HasForeignKey(a => a.TermId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.StudentId, e.SubjectId, e.Date }).IsUnique();

});

// Configure results

modelBuilder.Entity<Result>(entity =>

{

entity.HasOne(r => r.Student)

.WithMany(s => s.Results)

.HasForeignKey(r => r.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(r => r.Subject)

.WithMany(s => s.Results)

.HasForeignKey(r => r.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(r => r.Teacher)

.WithMany(t => t.Results)

.HasForeignKey(r => r.TeacherId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(r => r.Term)

.WithMany(t => t.Results)

.HasForeignKey(r => r.TermId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(r => r.SchoolYear)

.WithMany(sy => sy.Results)

.HasForeignKey(r => r.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.Property(e => e.Score).HasColumnType("decimal(6,2)");

entity.Property(e => e.TotalMarks).HasColumnType("decimal(6,2)");

entity.Property(e => e.Percentage).HasColumnType("decimal(5,2)");

entity.Property(e => e.AiConfidenceScore).HasColumnType("decimal(5,4)");

entity.Property(e => e.PartialCreditScore).HasColumnType("decimal(5,2)");

});

}

private void ConfigureRelationshipEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeSubject>(entity =>

{

entity.HasOne(gs => gs.Grade)

.WithMany(g => g.GradeSubjects)

.HasForeignKey(gs => gs.GradeId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(gs => gs.Subject)

.WithMany(s => s.GradeSubjects)

.HasForeignKey(gs => gs.SubjectId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.GradeId, e.SubjectId }).IsUnique();

});

modelBuilder.Entity<SubjectTeacher>(entity =>

{

entity.HasOne(st => st.Subject)

.WithMany(s => s.SubjectTeachers)

.HasForeignKey(st => st.SubjectId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(st => st.Teacher)

.WithMany(t => t.SubjectTeachers)

.HasForeignKey(st => st.TeacherId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(st => st.Grade)

.WithMany()

.HasForeignKey(st => st.GradeId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(st => st.SchoolYear)

.WithMany(sy => sy.SubjectTeachers)

.HasForeignKey(st => st.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.SubjectId, e.TeacherId, e.GradeId, e.SchoolYearId }).IsUnique();

});

modelBuilder.Entity<StudentSubject>(entity =>

{

entity.HasOne(ss => ss.Student)

.WithMany(s => s.StudentSubjects)

.HasForeignKey(ss => ss.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(ss => ss.Subject)

.WithMany(s => s.StudentSubjects)

.HasForeignKey(ss => ss.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(ss => ss.SchoolYear)

.WithMany(sy => sy.StudentSubjects)

.HasForeignKey(ss => ss.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.StudentId, e.SubjectId, e.SchoolYearId }).IsUnique();

});

modelBuilder.Entity<ClassTeacher>(entity =>

{

entity.HasOne(ct => ct.Teacher)

.WithMany(t => t.ClassTeachers)

.HasForeignKey(ct => ct.TeacherId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(ct => ct.Grade)

.WithMany(g => g.ClassTeachers)

.HasForeignKey(ct => ct.GradeId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(ct => ct.SchoolYear)

.WithMany(sy => sy.ClassTeachers)

.HasForeignKey(ct => ct.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.GradeId, e.SchoolYearId }).IsUnique();

});

}

private void ConfigureTimetableEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Timetable>(entity =>

{

entity.HasOne(t => t.School)

.WithMany(s => s.Timetables)

.HasForeignKey(t => t.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(t => t.Grade)

.WithMany(g => g.Timetables)

.HasForeignKey(t => t.GradeId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(t => t.SchoolYear)

.WithMany()

.HasForeignKey(t => t.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(t => t.Term)

.WithMany()

.HasForeignKey(t => t.TermId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.GradeId, e.SchoolYearId, e.TermId }).IsUnique();

});

modelBuilder.Entity<TimetableSlot>(entity =>

{

entity.HasOne(ts => ts.Timetable)

.WithMany(t => t.TimetableSlots)

.HasForeignKey(ts => ts.TimetableId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(ts => ts.Subject)

.WithMany(s => s.TimetableSlots)

.HasForeignKey(ts => ts.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(ts => ts.Teacher)

.WithMany(t => t.TimetableSlots)

.HasForeignKey(ts => ts.TeacherId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.TimetableId, e.DayOfWeek, e.PeriodNumber }).IsUnique();

entity.HasIndex(e => new { e.TeacherId, e.DayOfWeek, e.StartTime, e.EndTime });

});

modelBuilder.Entity<StudentTimetable>(entity =>

{

entity.HasOne(st => st.Student)

.WithMany(s => s.StudentTimetables)

.HasForeignKey(st => st.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(st => st.TimetableSlot)

.WithMany()

.HasForeignKey(st => st.TimetableSlotId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.StudentId, e.TimetableSlotId }).IsUnique();

});

}

private void ConfigureAIEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<HandwritingTrainingData>(entity =>

{

entity.HasOne(htd => htd.Student)

.WithMany(s => s.HandwritingTrainingData)

.HasForeignKey(htd => htd.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(htd => htd.Teacher)

.WithMany()

.HasForeignKey(htd => htd.TeacherId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasIndex(e => new { e.StudentId, e.Status });

entity.HasIndex(e => e.IsVerified);

entity.Property(e => e.ConfidenceScore).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<HandwritingCharacterData>(entity =>

{

entity.HasOne(hcd => hcd.HandwritingTrainingData)

.WithMany(htd => htd.CharacterData)

.HasForeignKey(hcd => hcd.HandwritingTrainingDataId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.HandwritingTrainingDataId, e.SequenceOrder });

entity.Property(e => e.ConfidenceScore).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<HandwritingModel>(entity =>

{

entity.HasOne(hm => hm.Student)

.WithMany()

.HasForeignKey(hm => hm.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.StudentId, e.Version }).IsUnique();

entity.HasIndex(e => new { e.StudentId, e.IsActive });

entity.Property(e => e.Accuracy).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<HandwritingRecognitionResult>(entity =>

{

entity.HasOne(hrr => hrr.Student)

.WithMany()

.HasForeignKey(hrr => hrr.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(hrr => hrr.Subject)

.WithMany()

.HasForeignKey(hrr => hrr.SubjectId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasOne(hrr => hrr.Result)

.WithMany()

.HasForeignKey(hrr => hrr.ResultId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasOne(hrr => hrr.HandwritingModel)

.WithMany(hm => hm.RecognitionResults)

.HasForeignKey(hrr => hrr.HandwritingModelId)

.OnDelete(DeleteBehavior.Restrict);

entity.Property(e => e.OverallConfidence).HasColumnType("decimal(5,4)");

entity.Property(e => e.PartialCreditScore).HasColumnType("decimal(5,2)");

});

modelBuilder.Entity<TrainingExercise>(entity =>

{

entity.HasOne(te => te.School)

.WithMany()

.HasForeignKey(te => te.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(te => te.Grade)

.WithMany()

.HasForeignKey(te => te.GradeId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasOne(te => te.Subject)

.WithMany()

.HasForeignKey(te => te.SubjectId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasIndex(e => new { e.SchoolId, e.Type, e.Difficulty });

});

modelBuilder.Entity<StudentTrainingSession>(entity =>

{

entity.HasOne(sts => sts.Student)

.WithMany()

.HasForeignKey(sts => sts.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(sts => sts.TrainingExercise)

.WithMany(te => te.StudentSessions)

.HasForeignKey(sts => sts.TrainingExerciseId)

.OnDelete(DeleteBehavior.Cascade);

entity.Property(e => e.OverallAccuracy).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<ModelTrainingJob>(entity =>

{

entity.HasOne(mtj => mtj.Student)

.WithMany()

.HasForeignKey(mtj => mtj.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.Status, e.QueuedAt });

entity.Property(e => e.Progress).HasColumnType("decimal(5,2)");

});

}

private void ConfigureIndexes(ModelBuilder modelBuilder)

{

// Additional performance indexes

modelBuilder.Entity<Result>()

.HasIndex(e => new { e.StudentId, e.TermId, e.SubjectId });

modelBuilder.Entity<Attendance>()

.HasIndex(e => new { e.Date, e.Status });

modelBuilder.Entity<Notification>()

.HasIndex(e => new { e.Type, e.IsSent, e.CreatedAt });

}

private void ConfigureSoftDelete(ModelBuilder modelBuilder)

{

// Configure soft delete for all entities inheriting from BaseEntity

foreach (var entityType in modelBuilder.Model.GetEntityTypes())

{

if (typeof(BaseEntity).IsAssignableFrom(entityType.ClrType))

{

modelBuilder.Entity(entityType.ClrType)

.HasQueryFilter(e => !((BaseEntity)e).IsDeleted);

}

}

}

public override int SaveChanges()

{

UpdateTimestamps();

return base.SaveChanges();

}

public override Task<int> SaveChangesAsync(CancellationToken cancellationToken = default)

{

UpdateTimestamps();

return base.SaveChangesAsync(cancellationToken);

}

private void UpdateTimestamps()

{

var entries = ChangeTracker.Entries<BaseEntity>();

foreach (var entry in entries)

{

switch (entry.State)

{

case EntityState.Added:

entry.Entity.CreatedAt = DateTime.UtcNow;

entry.Entity.UpdatedAt = DateTime.UtcNow;

break;

case EntityState.Modified:

entry.Entity.UpdatedAt = DateTime.UtcNow;

break;

}

}

}

}

}// Services/Interfaces/ISchoolService.cs

using SchoolManagement.Core.DTOs;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Services.Interfaces

{

public interface ISchoolService

{

Task<IEnumerable<SchoolDto>> GetAllSchoolsAsync();

Task<SchoolDto> GetSchoolByIdAsync(Guid schoolId);

Task<SchoolDto> CreateSchoolAsync(CreateSchoolDto createSchoolDto);

Task<SchoolDto> UpdateSchoolAsync(Guid schoolId, UpdateSchoolDto updateSchoolDto);

Task<bool> DeleteSchoolAsync(Guid schoolId);

Task<IEnumerable<SchoolDto>> GetSchoolsByGradingSchemeAsync(Guid gradingSchemeId);

}

}

// Services/Interfaces/IStudentService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IStudentService

{

Task<IEnumerable<StudentDto>> GetStudentsBySchoolAsync(Guid schoolId);

Task<IEnumerable<StudentDto>> GetStudentsByGradeAsync(Guid gradeId);

Task<StudentDto> GetStudentByIdAsync(Guid studentId);

Task<StudentDto> CreateStudentAsync(CreateStudentDto createStudentDto);

Task<StudentDto> UpdateStudentAsync(Guid studentId, UpdateStudentDto updateStudentDto);

Task<bool> DeleteStudentAsync(Guid studentId);

Task<bool> AssignStudentToSubjectsAsync(Guid studentId, List<Guid> subjectIds);

Task<IEnumerable<SubjectDto>> GetStudentSubjectsAsync(Guid studentId, Guid schoolYearId);

Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId);

}

}

// Services/Interfaces/ITeacherService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface ITeacherService

{

Task<IEnumerable<TeacherDto>> GetTeachersBySchoolAsync(Guid schoolId);

Task<TeacherDto> GetTeacherByIdAsync(Guid teacherId);

Task<TeacherDto> CreateTeacherAsync(CreateTeacherDto createTeacherDto);

Task<TeacherDto> UpdateTeacherAsync(Guid teacherId, UpdateTeacherDto updateTeacherDto);

Task<bool> DeleteTeacherAsync(Guid teacherId);

Task<bool> AssignTeacherToSubjectAsync(AssignTeacherSubjectDto assignDto);

Task<bool> AssignClassTeacherAsync(AssignClassTeacherDto assignDto);

Task<IEnumerable<SubjectDto>> GetTeacherSubjectsAsync(Guid teacherId, Guid schoolYearId);

Task<IEnumerable<GradeDto>> GetTeacherClassesAsync(Guid teacherId, Guid schoolYearId);

}

}

// Services/Interfaces/IAttendanceService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IAttendanceService

{

Task<AttendanceDto> MarkAttendanceAsync(CreateAttendanceDto createAttendanceDto);

Task<IEnumerable<AttendanceDto>> GetAttendanceByClassAsync(Guid gradeId, DateTime date);

Task<IEnumerable<AttendanceDto>> GetStudentAttendanceAsync(Guid studentId, Guid termId);

Task<AttendanceReportDto> GetAttendanceReportAsync(Guid studentId, Guid termId);

Task<bool> BulkMarkAttendanceAsync(List<CreateAttendanceDto> attendanceList);

Task<AttendanceDto> UpdateAttendanceAsync(Guid attendanceId, UpdateAttendanceDto updateDto);

}

}

// Services/Interfaces/IResultService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IResultService

{

Task<ResultDto> CreateResultAsync(CreateResultDto createResultDto);

Task<ResultDto> UpdateResultAsync(Guid resultId, UpdateResultDto updateResultDto);

Task<IEnumerable<ResultDto>> GetStudentResultsAsync(Guid studentId, Guid termId);

Task<IEnumerable<ResultDto>> GetClassResultsAsync(Guid gradeId, Guid subjectId, Guid termId);

Task<StudentReportCardDto> GenerateReportCardAsync(Guid studentId, Guid termId);

Task<bool> UploadResultsAsync(Guid teacherId, List<CreateResultDto> results);

Task<ClassResultSummaryDto> GetClassResultSummaryAsync(Guid gradeId, Guid termId);

Task<byte[]> PrintTermResultsAsync(Guid studentId, Guid termId);

Task<byte[]> PrintClassResultsAsync(Guid gradeId, Guid termId);

}

}

// Services/Interfaces/IGradingService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IGradingService

{

Task<IEnumerable<GradingSchemeDto>> GetAllGradingSchemesAsync();

Task<GradingSchemeDto> CreateGradingSchemeAsync(CreateGradingSchemeDto createDto);

Task<GradingSchemeDto> UpdateGradingSchemeAsync(Guid schemeId, UpdateGradingSchemeDto updateDto);

Task<bool> DeleteGradingSchemeAsync(Guid schemeId);

Task<GradingScaleDto> AddGradingScaleAsync(Guid schemeId, CreateGradingScaleDto createDto);

Task<GradingScaleDto> UpdateGradingScaleAsync(Guid scaleId, UpdateGradingScaleDto updateDto);

Task<bool> DeleteGradingScaleAsync(Guid scaleId);

Task<GradeCalculationDto> CalculateGradeAsync(Guid gradingSchemeId, decimal percentage);

Task<IEnumerable<GradingScaleDto>> GetGradingScalesAsync(Guid schemeId);

}

}

// Services/Interfaces/ITimetableService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface ITimetableService

{

Task<TimetableDto> GenerateTimetableAsync(GenerateTimetableDto generateDto);

Task<TimetableDto> GetGradeTimetableAsync(Guid gradeId, Guid termId);

Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId);

Task<TeacherTimetableDto> GetTeacherTimetableAsync(Guid teacherId, Guid termId);

Task<TimetableSlotDto> UpdateTimetableSlotAsync(Guid slotId, UpdateTimetableSlotDto updateDto);

Task<bool> ValidateTimetableAsync(Guid timetableId);

Task<TimetableConflictDto> CheckTimetableConflictsAsync(Guid timetableId);

Task<byte[]> PrintTimetableAsync(Guid timetableId);

}

}

// Services/Interfaces/INotificationService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface INotificationService

{

Task<NotificationDto> CreateNotificationAsync(CreateNotificationDto createDto);

Task<bool> SendNotificationAsync(Guid notificationId);

Task<bool> SendBulkNotificationAsync(List<Guid> recipientIds, CreateNotificationDto notificationDto);

Task<IEnumerable<NotificationDto>> GetUserNotificationsAsync(Guid userId, string userType);

Task<bool> MarkAsReadAsync(Guid notificationId);

Task<bool> SendWhatsAppMessageAsync(string phoneNumber, string message);

Task<bool> SendEmailNotificationAsync(string email, string subject, string message);

Task<NotificationStatsDto> GetNotificationStatsAsync(Guid schoolId);

}

}

// Services/Interfaces/IHandwritingRecognitionService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IHandwritingRecognitionService

{

Task<HandwritingRecognitionResultDto> ProcessHandwritingAsync(ProcessHandwritingDto processDto);

Task<TrainingExerciseDto> GenerateTrainingExerciseAsync(GenerateTrainingExerciseDto generateDto);

Task<StudentTrainingSessionDto> StartTrainingSessionAsync(Guid studentId, Guid exerciseId);

Task<StudentTrainingSessionDto> SubmitTrainingDataAsync(SubmitTrainingDataDto submitDto);

Task<bool> VerifyTrainingDataAsync(Guid trainingDataId, string verifiedText);

Task<ModelTrainingJobDto> StartModelTrainingAsync(Guid studentId);

Task<ModelTrainingJobDto> GetTrainingJobStatusAsync(Guid jobId);

Task<HandwritingModelDto> GetStudentModelAsync(Guid studentId);

Task<bool> DeployModelLocallyAsync(Guid modelId);

Task<TrainingProgressDto> GetStudentTrainingProgressAsync(Guid studentId);

Task<IEnumerable<HandwritingTrainingDataDto>> GetUnverifiedTrainingDataAsync(Guid teacherId);

}

}

// Services/Implementations/SchoolService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class SchoolService : ISchoolService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

public SchoolService(SchoolManagementDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

public async Task<IEnumerable<SchoolDto>> GetAllSchoolsAsync()

{

var schools = await \_context.Schools

.Include(s => s.GradingScheme)

.ToListAsync();

return \_mapper.Map<IEnumerable<SchoolDto>>(schools);

}

public async Task<SchoolDto> GetSchoolByIdAsync(Guid schoolId)

{

var school = await \_context.Schools

.Include(s => s.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == schoolId);

if (school == null)

throw new NotFoundException($"School with ID {schoolId} not found");

return \_mapper.Map<SchoolDto>(school);

}

public async Task<SchoolDto> CreateSchoolAsync(CreateSchoolDto createSchoolDto)

{

var school = \_mapper.Map<School>(createSchoolDto);

\_context.Schools.Add(school);

await \_context.SaveChangesAsync();

return \_mapper.Map<SchoolDto>(school);

}

public async Task<SchoolDto> UpdateSchoolAsync(Guid schoolId, UpdateSchoolDto updateSchoolDto)

{

var school = await \_context.Schools.FindAsync(schoolId);

if (school == null)

throw new NotFoundException($"School with ID {schoolId} not found");

\_mapper.Map(updateSchoolDto, school);

await \_context.SaveChangesAsync();

return \_mapper.Map<SchoolDto>(school);

}

public async Task<bool> DeleteSchoolAsync(Guid schoolId)

{

var school = await \_context.Schools.FindAsync(schoolId);

if (school == null)

return false;

school.IsDeleted = true;

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<SchoolDto>> GetSchoolsByGradingSchemeAsync(Guid gradingSchemeId)

{

var schools = await \_context.Schools

.Where(s => s.GradingSchemeId == gradingSchemeId)

.Include(s => s.GradingScheme)

.ToListAsync();

return \_mapper.Map<IEnumerable<SchoolDto>>(schools);

}

}

}

// Services/Implementations/StudentService.cs

namespace SchoolManagement.Services.Implementations

{

public class StudentService : IStudentService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

public StudentService(SchoolManagementDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

public async Task<IEnumerable<StudentDto>> GetStudentsBySchoolAsync(Guid schoolId)

{

var students = await \_context.Students

.Where(s => s.SchoolId == schoolId)

.Include(s => s.CurrentGrade)

.Include(s => s.Parents)

.ToListAsync();

return \_mapper.Map<IEnumerable<StudentDto>>(students);

}

public async Task<IEnumerable<StudentDto>> GetStudentsByGradeAsync(Guid gradeId)

{

var students = await \_context.Students

.Where(s => s.CurrentGradeId == gradeId)

.Include(s => s.CurrentGrade)

.Include(s => s.Parents)

.ToListAsync();

return \_mapper.Map<IEnumerable<StudentDto>>(students);

}

public async Task<StudentDto> GetStudentByIdAsync(Guid studentId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.Parents)

.Include(s => s.School)

.FirstOrDefaultAsync(s => s.Id == studentId);

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

return \_mapper.Map<StudentDto>(student);

}

public async Task<StudentDto> CreateStudentAsync(CreateStudentDto createStudentDto)

{

var student = \_mapper.Map<Student>(createStudentDto);

// Generate student number if not provided

if (string.IsNullOrEmpty(student.StudentNumber))

{

student.StudentNumber = await GenerateStudentNumberAsync(student.SchoolId);

}

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

return \_mapper.Map<StudentDto>(student);

}

public async Task<StudentDto> UpdateStudentAsync(Guid studentId, UpdateStudentDto updateStudentDto)

{

var student = await \_context.Students.FindAsync(studentId);

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

\_mapper.Map(updateStudentDto, student);

await \_context.SaveChangesAsync();

return \_mapper.Map<StudentDto>(student);

}

public async Task<bool> DeleteStudentAsync(Guid studentId)

{

var student = await \_context.Students.FindAsync(studentId);

if (student == null)

return false;

student.IsDeleted = true;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> AssignStudentToSubjectsAsync(Guid studentId, List<Guid> subjectIds)

{

var student = await \_context.Students.FindAsync(studentId);

if (student == null)

return false;

var currentSchoolYear = await \_context.SchoolYears

.Where(sy => sy.SchoolId == student.SchoolId && sy.IsCurrent)

.FirstOrDefaultAsync();

if (currentSchoolYear == null)

return false;

// Remove existing assignments for current school year

var existingAssignments = await \_context.StudentSubjects

.Where(ss => ss.StudentId == studentId && ss.SchoolYearId == currentSchoolYear.Id)

.ToListAsync();

\_context.StudentSubjects.RemoveRange(existingAssignments);

// Add new assignments

var newAssignments = subjectIds.Select(subjectId => new StudentSubject

{

StudentId = studentId,

SubjectId = subjectId,

SchoolYearId = currentSchoolYear.Id,

EnrollmentDate = DateTime.UtcNow

});

\_context.StudentSubjects.AddRange(newAssignments);

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<SubjectDto>> GetStudentSubjectsAsync(Guid studentId, Guid schoolYearId)

{

var subjects = await \_context.StudentSubjects

.Where(ss => ss.StudentId == studentId && ss.SchoolYearId == schoolYearId && ss.IsActive)

.Include(ss => ss.Subject)

.Select(ss => ss.Subject)

.ToListAsync();

return \_mapper.Map<IEnumerable<SubjectDto>>(subjects);

}

public async Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.FirstOrDefaultAsync(s => s.Id == studentId);

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

var currentTerm = await \_context.Terms

.Where(t => t.SchoolYear.SchoolId == student.SchoolId && t.IsCurrent)

.FirstOrDefaultAsync();

if (currentTerm == null)

throw new NotFoundException("No current term found");

var timetable = await \_context.Timetables

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Subject)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Teacher)

.FirstOrDefaultAsync(t => t.GradeId == student.CurrentGradeId && t.TermId == currentTerm.Id);

return \_mapper.Map<StudentTimetableDto>(timetable);

}

private async Task<string> GenerateStudentNumberAsync(Guid schoolId)

{

var year = DateTime.Now.Year;

var lastStudent = await \_context.Students

.Where(s => s.SchoolId == schoolId && s.StudentNumber.StartsWith(year.ToString()))

.OrderByDescending(s => s.StudentNumber)

.FirstOrDefaultAsync();

var sequence = 1;

if (lastStudent != null && int.TryParse(lastStudent.StudentNumber.Substring(4), out var lastSequence))

{

sequence = lastSequence + 1;

}

return $"{year}{sequence:D4}";

}

}

}

// Services/Implementations/ResultService.cs

namespace SchoolManagement.Services.Implementations

{

public class ResultService : IResultService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly IGradingService \_gradingService;

private readonly IHandwritingRecognitionService \_handwritingService;

public ResultService(

SchoolManagementDbContext context,

IMapper mapper,

IGradingService gradingService,

IHandwritingRecognitionService handwritingService)

{

\_context = context;

\_mapper = mapper;

\_gradingService = gradingService;

\_handwritingService = handwritingService;

}

public async Task<ResultDto> CreateResultAsync(CreateResultDto createResultDto)

{

var result = \_mapper.Map<r>(createResultDto);

// Calculate percentage

result.Percentage = (result.Score / result.TotalMarks) \* 100;

// Get student's school grading scheme

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == createResultDto.StudentId);

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

result.Percentage);

result.Grade = gradeCalculation.Symbol;

result.GradeUnit = gradeCalculation.Unit;

}

\_context.Results.Add(result);

await \_context.SaveChangesAsync();

return \_mapper.Map<ResultDto>(result);

}

public async Task<ResultDto> UpdateResultAsync(Guid resultId, UpdateResultDto updateResultDto)

{

var result = await \_context.Results.FindAsync(resultId);

if (result == null)

throw new NotFoundException($"Result with ID {resultId} not found");

\_mapper.Map(updateResultDto, result);

// Recalculate grade if score changed

if (updateResultDto.Score.HasValue || updateResultDto.TotalMarks.HasValue)

{

result.Percentage = (result.Score / result.TotalMarks) \* 100;

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == result.StudentId);

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

result.Percentage);

result.Grade = gradeCalculation.Symbol;

result.GradeUnit = gradeCalculation.Unit;

}

}

await \_context.SaveChangesAsync();

return \_mapper.Map<ResultDto>(result);

}

public async Task<IEnumerable<ResultDto>> GetStudentResultsAsync(Guid studentId, Guid termId)

{

var results = await \_context.Results

.Where(r => r.StudentId == studentId && r.TermId == termId)

.Include(r => r.Subject)

.Include(r => r.Teacher)

.ToListAsync();

return \_mapper.Map<IEnumerable<ResultDto>>(results);

}

public async Task<IEnumerable<ResultDto>> GetClassResultsAsync(Guid gradeId, Guid subjectId, Guid termId)

{

var results = await \_context.Results

.Where(r => r.Student.CurrentGradeId == gradeId &&

r.SubjectId == subjectId &&

r.TermId == termId)

.Include(r => r.Student)

.Include(r => r.Subject)

.ToListAsync();

return \_mapper.Map<IEnumerable<ResultDto>>(results);

}

public async Task<StudentReportCardDto> GenerateReportCardAsync(Guid studentId, Guid termId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.School)

.Include(s => s.Parents)

.FirstOrDefaultAsync(s => s.Id == studentId);

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

var term = await \_context.Terms

.Include(t => t.SchoolYear)

.FirstOrDefaultAsync(t => t.Id == termId);

var results = await GetStudentResultsAsync(studentId, termId);

var attendance = await \_context.Attendances

.Where(a => a.StudentId == studentId && a.TermId == termId)

.GroupBy(a => a.SubjectId)

.Select(g => new

{

SubjectId = g.Key,

TotalClasses = g.Count(),

PresentClasses = g.Count(a => a.Status == AttendanceStatus.Present)

})

.ToListAsync();

var reportCard = new StudentReportCardDto

{

Student = \_mapper.Map<StudentDto>(student),

Term = \_mapper.Map<TermDto>(term),

Results = results,

AttendanceSummary = attendance.Select(a => new AttendanceSummaryDto

{

SubjectId = a.SubjectId,

TotalClasses = a.TotalClasses,

PresentClasses = a.PresentClasses,

AttendancePercentage = (decimal)a.PresentClasses / a.TotalClasses \* 100

}).ToList(),

GeneratedAt = DateTime.UtcNow

};

return reportCard;

}

public async Task<bool> UploadResultsAsync(Guid teacherId, List<CreateResultDto> results)

{

var teacher = await \_context.Teachers.FindAsync(teacherId);

if (teacher == null)

return false;

var resultEntities = new List<r>();

foreach (var createResultDto in results)

{

var result = \_mapper.Map<r>(createResultDto);

result.TeacherId = teacherId;

result.Percentage = (result.Score / result.TotalMarks) \* 100;

// Calculate grade

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == createResultDto.StudentId);

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

result.Percentage);

result.Grade = gradeCalculation.Symbol;

result.GradeUnit = gradeCalculation.Unit;

}

resultEntities.Add(result);

}

\_context.Results.AddRange(resultEntities);

await \_context.SaveChangesAsync();

return true;

}

public async Task<ClassResultSummaryDto> GetClassResultSummaryAsync(Guid gradeId, Guid termId)

{

var results = await \_context.Results

.Where(r => r.Student.CurrentGradeId == gradeId && r.TermId == termId)

.Include(r => r.Subject)

.Include(r => r.Student)

.ToListAsync();

var subjectSummaries = results

.GroupBy(r => r.Subject)

.Select(g => new SubjectResultSummaryDto

{

Subject = \_mapper.Map<SubjectDto>(g.Key),

TotalStudents = g.Count(),

AverageScore = g.Average(r => r.Percentage),

HighestScore = g.Max(r => r.Percentage),

LowestScore = g.Min(r => r.Percentage),

PassCount = g.Count(r => r.Percentage >= 50), // Assuming 50% is pass mark

FailCount = g.Count(r => r.Percentage < 50)

}).ToList();

var grade = await \_context.Grades.FindAsync(gradeId);

var term = await \_context.Terms.FindAsync(termId);

return new ClassResultSummaryDto

{

Grade = \_mapper.Map<GradeDto>(grade),

Term = \_mapper.Map<TermDto>(term),

SubjectSummaries = subjectSummaries,

TotalStudents = results.Select(r => r.StudentId).Distinct().Count(),

OverallAverage = results.Average(r => r.Percentage)

};

}

public async Task<byte[]> PrintTermResultsAsync(Guid studentId, Guid termId)

{

var reportCard = await GenerateReportCardAsync(studentId, termId);

// Implementation would generate PDF using a library like iTextSharp or similar

// This is a placeholder for PDF generation logic

var pdfService = new PdfGenerationService();

return await pdfService.GenerateReportCardPdfAsync(reportCard);

}

public async Task<byte[]> PrintClassResultsAsync(Guid gradeId, Guid termId)

{

var classSummary = await GetClassResultSummaryAsync(gradeId, termId);

// Implementation would generate PDF

var pdfService = new PdfGenerationService();

return await pdfService.GenerateClassResultsPdfAsync(classSummary);

}

}

}// Services/Implementations/HandwritingRecognitionService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Services.Interfaces;

using System.Drawing;

using System.Drawing.Imaging;

namespace SchoolManagement.Services.Implementations

{

public class HandwritingRecognitionService : IHandwritingRecognitionService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly IImageProcessingService \_imageProcessingService;

private readonly IMachineLearningService \_mlService;

private readonly IFileStorageService \_fileStorageService;

private readonly ILogger<HandwritingRecognitionService> \_logger;

public HandwritingRecognitionService(

SchoolManagementDbContext context,

IMapper mapper,

IImageProcessingService imageProcessingService,

IMachineLearningService mlService,

IFileStorageService fileStorageService,

ILogger<HandwritingRecognitionService> logger)

{

\_context = context;

\_mapper = mapper;

\_imageProcessingService = imageProcessingService;

\_mlService = mlService;

\_fileStorageService = fileStorageService;

\_logger = logger;

}

public async Task<HandwritingRecognitionResultDto> ProcessHandwritingAsync(ProcessHandwritingDto processDto)

{

try

{

var startTime = DateTime.UtcNow;

// Get student's personal model if available

var personalModel = await GetActiveModelForStudentAsync(processDto.StudentId);

// Process image and extract text

var imageProcessingResult = await \_imageProcessingService.ProcessImageAsync(processDto.ImagePath);

string recognizedText;

decimal confidence;

bool processedLocally = false;

if (personalModel != null && personalModel.IsDeployedLocally)

{

// Use local personal model for recognition

var localResult = await \_mlService.RecognizeTextLocallyAsync(

personalModel.ModelFilePath,

processDto.ImagePath);

recognizedText = localResult.Text;

confidence = localResult.Confidence;

processedLocally = true;

}

else

{

// Fallback to cloud-based recognition

var cloudResult = await \_mlService.RecognizeTextInCloudAsync(processDto.ImagePath);

recognizedText = cloudResult.Text;

confidence = cloudResult.Confidence;

processedLocally = false;

}

var processingTime = DateTime.UtcNow - startTime;

// Create recognition result record

var recognitionResult = new HandwritingRecognitionResult

{

StudentId = processDto.StudentId,

SubjectId = processDto.SubjectId,

ResultId = processDto.ResultId,

HandwritingModelId = personalModel?.Id ?? Guid.Empty,

OriginalImagePath = processDto.ImagePath,

RecognizedText = recognizedText,

OverallConfidence = confidence,

ProcessedLocally = processedLocally,

ProcessedInCloud = !processedLocally,

ProcessingTime = processingTime,

CharacterConfidences = imageProcessingResult.CharacterConfidences,

AlternativeTexts = imageProcessingResult.AlternativeTexts,

ExpectedAnswer = processDto.ExpectedAnswer ?? "",

ProcessedAt = DateTime.UtcNow

};

// Calculate correctness if expected answer provided

if (!string.IsNullOrEmpty(processDto.ExpectedAnswer))

{

var answerComparison = await CompareAnswersAsync(recognizedText, processDto.ExpectedAnswer);

recognitionResult.IsCorrect = answerComparison.IsCorrect;

recognitionResult.PartialCreditScore = answerComparison.PartialCredit;

}

\_context.HandwritingRecognitionResults.Add(recognitionResult);

await \_context.SaveChangesAsync();

return \_mapper.Map<HandwritingRecognitionResultDto>(recognitionResult);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error processing handwriting for student {StudentId}", processDto.StudentId);

throw new HandwritingProcessingException("Failed to process handwriting", ex);

}

}

public async Task<TrainingExerciseDto> GenerateTrainingExerciseAsync(GenerateTrainingExerciseDto generateDto)

{

var exercise = new TrainingExercise

{

SchoolId = generateDto.SchoolId,

GradeId = generateDto.GradeId,

SubjectId = generateDto.SubjectId,

Title = generateDto.Title,

Description = generateDto.Description,

Type = generateDto.Type,

Difficulty = generateDto.Difficulty,

IsSystemGenerated = true,

EstimatedMinutes = generateDto.EstimatedMinutes

};

// Generate exercise content based on type and difficulty

var content = await GenerateExerciseContentAsync(generateDto);

exercise.Content = System.Text.Json.JsonSerializer.Serialize(content);

\_context.TrainingExercises.Add(exercise);

await \_context.SaveChangesAsync();

return \_mapper.Map<TrainingExerciseDto>(exercise);

}

public async Task<StudentTrainingSessionDto> StartTrainingSessionAsync(Guid studentId, Guid exerciseId)

{

var exercise = await \_context.TrainingExercises.FindAsync(exerciseId);

if (exercise == null)

throw new NotFoundException($"Training exercise {exerciseId} not found");

var session = new StudentTrainingSession

{

StudentId = studentId,

TrainingExerciseId = exerciseId,

StartedAt = DateTime.UtcNow,

Status = SessionStatus.InProgress

};

// Parse exercise content to get total items

var exerciseContent = System.Text.Json.JsonSerializer.Deserialize<ExerciseContentDto>(exercise.Content);

session.TotalItems = exerciseContent?.Items?.Count ?? 0;

\_context.StudentTrainingSessions.Add(session);

await \_context.SaveChangesAsync();

return \_mapper.Map<StudentTrainingSessionDto>(session);

}

public async Task<StudentTrainingSessionDto> SubmitTrainingDataAsync(SubmitTrainingDataDto submitDto)

{

var session = await \_context.StudentTrainingSessions

.Include(s => s.TrainingExercise)

.FirstOrDefaultAsync(s => s.Id == submitDto.SessionId);

if (session == null)

throw new NotFoundException($"Training session {submitDto.SessionId} not found");

// Store the uploaded image

var imagePath = await \_fileStorageService.SaveImageAsync(

submitDto.HandwritingImage,

$"training/{session.StudentId}/{session.Id}");

// Process the handwritten text

var processedText = await \_imageProcessingService.ExtractTextAsync(imagePath);

// Create training data record

var trainingData = new HandwritingTrainingData

{

StudentId = session.StudentId,

DataType = DetermineDataType(submitDto.OriginalContent),

OriginalContent = submitDto.OriginalContent,

ImagePath = imagePath,

ProcessedText = processedText.Text,

ConfidenceScore = processedText.Confidence,

Status = TrainingStatus.Processing,

CapturedAt = DateTime.UtcNow,

BoundingBoxes = processedText.BoundingBoxes,

ModelVersion = await GetCurrentModelVersionAsync()

};

// Extract individual characters for detailed training

var characterData = await ExtractCharacterDataAsync(trainingData);

trainingData.CharacterData = characterData;

\_context.HandwritingTrainingData.Add(trainingData);

// Update session progress

session.CompletedItems++;

session.OverallAccuracy = CalculateSessionAccuracy(session);

if (session.CompletedItems >= session.TotalItems)

{

session.Status = SessionStatus.Completed;

session.CompletedAt = DateTime.UtcNow;

session.TotalTime = session.CompletedAt.Value - session.StartedAt;

}

await \_context.SaveChangesAsync();

// Queue for verification if confidence is low

if (trainingData.ConfidenceScore < 0.8m) // 80% threshold

{

trainingData.Status = TrainingStatus.NeedsVerification;

await \_context.SaveChangesAsync();

}

return \_mapper.Map<StudentTrainingSessionDto>(session);

}

public async Task<bool> VerifyTrainingDataAsync(Guid trainingDataId, string verifiedText)

{

var trainingData = await \_context.HandwritingTrainingData

.Include(td => td.CharacterData)

.FirstOrDefaultAsync(td => td.Id == trainingDataId);

if (trainingData == null)

return false;

trainingData.VerifiedText = verifiedText;

trainingData.IsVerified = true;

trainingData.VerifiedAt = DateTime.UtcNow;

trainingData.Status = TrainingStatus.Verified;

// Update character-level verification if needed

await UpdateCharacterVerificationAsync(trainingData, verifiedText);

await \_context.SaveChangesAsync();

// Check if student has enough verified data to start training

await CheckAndQueueModelTrainingAsync(trainingData.StudentId);

return true;

}

public async Task<ModelTrainingJobDto> StartModelTrainingAsync(Guid studentId)

{

// Check if student has sufficient training data

var verifiedDataCount = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.IsVerified);

if (verifiedDataCount < 100) // Minimum threshold

{

throw new InvalidOperationException($"Insufficient training data. Need at least 100 verified samples, have {verifiedDataCount}");

}

var job = new ModelTrainingJob

{

StudentId = studentId,

JobName = $"Personal Model Training - {DateTime.UtcNow:yyyy-MM-dd HH:mm}",

Type = JobType.InitialTraining,

Status = JobStatus.Queued,

TotalDataPoints = verifiedDataCount,

TrainingParameters = System.Text.Json.JsonSerializer.Serialize(GetDefaultTrainingParameters())

};

\_context.ModelTrainingJobs.Add(job);

await \_context.SaveChangesAsync();

// Start training process asynchronously

\_ = Task.Run(() => ProcessModelTrainingAsync(job.Id));

return \_mapper.Map<ModelTrainingJobDto>(job);

}

public async Task<ModelTrainingJobDto> GetTrainingJobStatusAsync(Guid jobId)

{

var job = await \_context.ModelTrainingJobs.FindAsync(jobId);

if (job == null)

throw new NotFoundException($"Training job {jobId} not found");

return \_mapper.Map<ModelTrainingJobDto>(job);

}

public async Task<HandwritingModelDto> GetStudentModelAsync(Guid studentId)

{

var model = await \_context.HandwritingModels

.Where(m => m.StudentId == studentId && m.IsActive)

.OrderByDescending(m => m.TrainingCompleted)

.FirstOrDefaultAsync();

if (model == null)

throw new NotFoundException($"No active model found for student {studentId}");

return \_mapper.Map<HandwritingModelDto>(model);

}

public async Task<bool> DeployModelLocallyAsync(Guid modelId)

{

var model = await \_context.HandwritingModels.FindAsync(modelId);

if (model == null)

return false;

try

{

// Download model from cloud storage

var localModelPath = await \_fileStorageService.DownloadModelAsync(model.CloudModelPath, modelId);

// Validate model

var isValid = await \_mlService.ValidateModelAsync(localModelPath);

if (!isValid)

{

\_logger.LogError("Model validation failed for model {ModelId}", modelId);

return false;

}

model.ModelFilePath = localModelPath;

model.IsDeployedLocally = true;

model.LastUsed = DateTime.UtcNow;

await \_context.SaveChangesAsync();

\_logger.LogInformation("Model {ModelId} successfully deployed locally", modelId);

return true;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to deploy model {ModelId} locally", modelId);

return false;

}

}

public async Task<TrainingProgressDto> GetStudentTrainingProgressAsync(Guid studentId)

{

var totalData = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId);

var verifiedData = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.IsVerified);

var pendingVerification = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.Status == TrainingStatus.NeedsVerification);

var currentModel = await \_context.HandwritingModels

.Where(m => m.StudentId == studentId && m.IsActive)

.OrderByDescending(m => m.TrainingCompleted)

.FirstOrDefaultAsync();

var recentSessions = await \_context.StudentTrainingSessions

.Where(s => s.StudentId == studentId)

.OrderByDescending(s => s.StartedAt)

.Take(5)

.ToListAsync();

return new TrainingProgressDto

{

StudentId = studentId,

TotalTrainingData = totalData,

VerifiedData = verifiedData,

PendingVerification = pendingVerification,

CurrentModelAccuracy = currentModel?.Accuracy ?? 0,

ModelIsDeployedLocally = currentModel?.IsDeployedLocally ?? false,

RecentSessions = \_mapper.Map<List<StudentTrainingSessionDto>>(recentSessions),

ReadyForTraining = verifiedData >= 100,

CompletionPercentage = Math.Min(100, (verifiedData / 100.0m) \* 100)

};

}

public async Task<IEnumerable<HandwritingTrainingDataDto>> GetUnverifiedTrainingDataAsync(Guid teacherId)

{

var teacher = await \_context.Teachers

.Include(t => t.School)

.FirstOrDefaultAsync(t => t.Id == teacherId);

if (teacher == null)

throw new NotFoundException($"Teacher {teacherId} not found");

var unverifiedData = await \_context.HandwritingTrainingData

.Where(td => td.Student.SchoolId == teacher.SchoolId &&

td.Status == TrainingStatus.NeedsVerification)

.Include(td => td.Student)

.OrderBy(td => td.CapturedAt)

.Take(50) // Limit for performance

.ToListAsync();

return \_mapper.Map<IEnumerable<HandwritingTrainingDataDto>>(unverifiedData);

}

// Private helper methods

private async Task<HandwritingModel> GetActiveModelForStudentAsync(Guid studentId)

{

return await \_context.HandwritingModels

.Where(m => m.StudentId == studentId && m.IsActive)

.OrderByDescending(m => m.TrainingCompleted)

.FirstOrDefaultAsync();

}

private async Task<AnswerComparisonResult> CompareAnswersAsync(string recognizedText, string expectedAnswer)

{

// Implement fuzzy matching logic for answer comparison

var similarity = CalculateStringSimilarity(recognizedText.Trim().ToLower(), expectedAnswer.Trim().ToLower());

return new AnswerComparisonResult

{

IsCorrect = similarity >= 0.9m, // 90% similarity threshold

PartialCredit = similarity \* 100,

Similarity = similarity

};

}

private decimal CalculateStringSimilarity(string text1, string text2)

{

if (string.IsNullOrEmpty(text1) || string.IsNullOrEmpty(text2))

return 0;

if (text1 == text2)

return 1;

// Levenshtein distance implementation

var distance = CalculateLevenshteinDistance(text1, text2);

var maxLength = Math.Max(text1.Length, text2.Length);

return 1 - (decimal)distance / maxLength;

}

private int CalculateLevenshteinDistance(string source, string target)

{

if (string.IsNullOrEmpty(source)) return target?.Length ?? 0;

if (string.IsNullOrEmpty(target)) return source.Length;

var matrix = new int[source.Length + 1, target.Length + 1];

for (int i = 0; i <= source.Length; i++)

matrix[i, 0] = i;

for (int j = 0; j <= target.Length; j++)

matrix[0, j] = j;

for (int i = 1; i <= source.Length; i++)

{

for (int j = 1; j <= target.Length; j++)

{

var cost = source[i - 1] == target[j - 1] ? 0 : 1;

matrix[i, j] = Math.Min(

Math.Min(matrix[i - 1, j] + 1, matrix[i, j - 1] + 1),

matrix[i - 1, j - 1] + cost);

}

}

return matrix[source.Length, target.Length];

}

private async Task<ExerciseContentDto> GenerateExerciseContentAsync(GenerateTrainingExerciseDto generateDto)

{

var content = new ExerciseContentDto

{

Items = new List<ExerciseItemDto>()

};

switch (generateDto.Type)

{

case ExerciseType.Alphabet:

content.Items = GenerateAlphabetItems(generateDto.Difficulty);

break;

case ExerciseType.Numbers:

content.Items = GenerateNumberItems(generateDto.Difficulty);

break;

case ExerciseType.Words:

content.Items = await GenerateWordItems(generateDto.Difficulty, generateDto.SubjectId);

break;

case ExerciseType.Sentences:

content.Items = await GenerateSentenceItems(generateDto.Difficulty, generateDto.SubjectId);

break;

case ExerciseType.MathProblems:

content.Items = GenerateMathItems(generateDto.Difficulty);

break;

}

return content;

}

private List<ExerciseItemDto> GenerateAlphabetItems(DifficultyLevel difficulty)

{

var items = new List<ExerciseItemDto>();

var letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

switch (difficulty)

{

case DifficultyLevel.Beginner:

// First 10 letters

for (int i = 0; i < 10; i++)

{

items.Add(new ExerciseItemDto

{

Content = letters[i].ToString(),

ExpectedResponse = letters[i].ToString(),

ItemType = "Letter"

});

}

break;

case DifficultyLevel.Elementary:

// All uppercase letters

foreach (char letter in letters)

{

items.Add(new ExerciseItemDto

{

Content = letter.ToString(),

ExpectedResponse = letter.ToString(),

ItemType = "Letter"

});

}

break;

case DifficultyLevel.Intermediate:

// Mixed case

foreach (char letter in letters)

{

items.Add(new ExerciseItemDto

{

Content = letter.ToString(),

ExpectedResponse = letter.ToString(),

ItemType = "Letter"

});

items.Add(new ExerciseItemDto

{

Content = letter.ToString().ToLower(),

ExpectedResponse = letter.ToString().ToLower(),

ItemType = "Letter"

});

}

break;

}

return items;

}

private List<ExerciseItemDto> GenerateNumberItems(DifficultyLevel difficulty)

{

var items = new List<ExerciseItemDto>();

switch (difficulty)

{

case DifficultyLevel.Beginner:

for (int i = 0; i <= 20; i++)

{

items.Add(new ExerciseItemDto

{

Content = i.ToString(),

ExpectedResponse = i.ToString(),

ItemType = "Number"

});

}

break;

case DifficultyLevel.Elementary:

for (int i = 0; i <= 100; i += 5)

{

items.Add(new ExerciseItemDto

{

Content = i.ToString(),

ExpectedResponse = i.ToString(),

ItemType = "Number"

});

}

break;

case DifficultyLevel.Intermediate:

var random = new Random();

for (int i = 0; i < 50; i++)

{

var number = random.Next(1, 1000);

items.Add(new ExerciseItemDto

{

Content = number.ToString(),

ExpectedResponse = number.ToString(),

ItemType = "Number"

});

}

break;

}

return items;

}

private async Task<List<ExerciseItemDto>> GenerateWordItems(DifficultyLevel difficulty, Guid? subjectId)

{

var items = new List<ExerciseItemDto>();

// This would typically load from a word database

var basicWords = new[] { "cat", "dog", "sun", "book", "tree", "house", "car", "ball" };

var intermediateWords = new[] { "computer", "elephant", "butterfly", "mountain", "rainbow", "chocolate" };

var advancedWords = new[] { "responsibility", "environment", "democracy", "photosynthesis", "constellation" };

string[] wordsToUse = difficulty switch

{

DifficultyLevel.Beginner => basicWords,

DifficultyLevel.Elementary => basicWords.Concat(intermediateWords).ToArray(),

DifficultyLevel.Intermediate => intermediateWords.Concat(advancedWords).ToArray(),

\_ => advancedWords

};

foreach (var word in wordsToUse.Take(20))

{

items.Add(new ExerciseItemDto

{

Content = word,

ExpectedResponse = word,

ItemType = "Word"

});

}

return items;

}

private async Task<List<ExerciseItemDto>> GenerateSentenceItems(DifficultyLevel difficulty, Guid? subjectId)

{

var items = new List<ExerciseItemDto>();

var beginnerSentences = new[]

{

"The cat is sleeping.",

"I like to read books.",

"The sun is bright today."

};

var intermediateSentences = new[]

{

"Students should always complete their homework on time.",

"The weather forecast predicts rain for tomorrow.",

"Science helps us understand the world around us."

};

string[] sentencesToUse = difficulty switch

{

DifficultyLevel.Beginner => beginnerSentences,

\_ => intermediateSentences

};

foreach (var sentence in sentencesToUse)

{

items.Add(new ExerciseItemDto

{

Content = sentence,

ExpectedResponse = sentence,

ItemType = "Sentence"

});

}

return items;

}

private List<ExerciseItemDto> GenerateMathItems(DifficultyLevel difficulty)

{

var items = new List<ExerciseItemDto>();

var random = new Random();

for (int i = 0; i < 10; i++)

{

int a, b;

string operation, expression, answer;

switch (difficulty)

{

case DifficultyLevel.Beginner:

a = random.Next(1, 10);

b = random.Next(1, 10);

operation = random.Next(2) == 0 ? "+" : "-";

expression = $"{a} {operation} {b} = ?";

answer = operation == "+" ? (a + b).ToString() : Math.Max(a - b, 0).ToString();

break;

default:

a = random.Next(10, 100);

b = random.Next(10, 100);

operation = new[] { "+", "-", "×", "÷" }[random.Next(4)];

expression = $"{a} {operation} {b} = ?";

answer = operation switch

{

"+" => (a + b).ToString(),

"-" => (a - b).ToString(),

"×" => (a \* b).ToString(),

"÷" => (a / b).ToString(),

\_ => "0"

};

break;

}

items.Add(new ExerciseItemDto

{

Content = expression,

ExpectedResponse = answer,

ItemType = "MathProblem"

});

}

return items;

}

private TrainingDataType DetermineDataType(string content)

{

if (content.All(char.IsLetter) && content.Length == 1)

return TrainingDataType.Alphabet;

if (content.All(char.IsDigit))

return TrainingDataType.Numbers;

if (content.Contains('+') || content.Contains('-') || content.Contains('×') || content.Contains('÷'))

return TrainingDataType.MathExpressions;

if (content.Split(' ').Length > 5)

return TrainingDataType.Sentences;

return TrainingDataType.Words;

}

private async Task<List<HandwritingCharacterData>> ExtractCharacterDataAsync(HandwritingTrainingData trainingData)

{

var characterData = new List<HandwritingCharacterData>();

// This would use computer vision to segment the image into individual characters

var characterSegments = await \_imageProcessingService.SegmentCharactersAsync(trainingData.ImagePath);

for (int i = 0; i < characterSegments.Count && i < trainingData.OriginalContent.Length; i++)

{

var segment = characterSegments[i];

var character = trainingData.OriginalContent[i].ToString();

characterData.Add(new HandwritingCharacterData

{

HandwritingTrainingDataId = trainingData.Id,

Character = character,

ImageSegmentPath = segment.ImagePath,

X = segment.BoundingBox.X,

Y = segment.BoundingBox.Y,

Width = segment.BoundingBox.Width,

Height = segment.BoundingBox.Height,

ConfidenceScore = segment.Confidence,

SequenceOrder = i,

FeatureVector = segment.FeatureVector

});

}

return characterData;

}

private decimal CalculateSessionAccuracy(StudentTrainingSession session)

{

if (session.CompletedItems == 0) return 0;

return (decimal)session.CorrectItems / session.CompletedItems \* 100;

}

private async Task UpdateCharacterVerificationAsync(HandwritingTrainingData trainingData, string verifiedText)

{

for (int i = 0; i < Math.Min(trainingData.CharacterData.Count, verifiedText.Length); i++)

{

var characterData = trainingData.CharacterData.Skip(i).First();

characterData.Character = verifiedText[i].ToString();

characterData.IsVerified = true;

}

}

private async Task CheckAndQueueModelTrainingAsync(Guid studentId)

{

var verifiedCount = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.IsVerified);

var existingJob = await \_context.ModelTrainingJobs

.Where(j => j.StudentId == studentId &&

(j.Status == JobStatus.Queued || j.Status == JobStatus.Running))

.FirstOrDefaultAsync();

if (verifiedCount >= 100 && existingJob == null)

{

await StartModelTrainingAsync(studentId);

}

}

private async Task ProcessModelTrainingAsync(Guid jobId)

{

var job = await \_context.ModelTrainingJobs.FindAsync(jobId);

if (job == null) return;

try

{

job.Status = JobStatus.Running;

job.StartedAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

// Get training data

var trainingData = await \_context.HandwritingTrainingData

.Where(td => td.StudentId == job.StudentId && td.IsVerified)

.Include(td => td.CharacterData)

.ToListAsync();

// Train the model

var modelResult = await \_mlService.TrainPersonalModelAsync(trainingData);

// Create model record

var model = new HandwritingModel

{

StudentId = job.StudentId,

ModelName = $"Personal Model v{DateTime.UtcNow:yyyyMMdd}",

Version = "1.0",

Type = ModelType.Personal,

Accuracy = modelResult.Accuracy,

TrainingDataCount = trainingData.Count,

CloudModelPath = modelResult.CloudPath,

TrainingStarted = job.StartedAt.Value,

TrainingCompleted = DateTime.UtcNow,

TrainingParameters = job.TrainingParameters,

PerformanceMetrics = System.Text.Json.JsonSerializer.Serialize(modelResult.Metrics),

IsActive = true,

IsDeployedCloud = true

};

\_context.HandwritingModels.Add(model);

job.Status = JobStatus.Completed;

job.CompletedAt = DateTime.UtcNow;

job.Progress = 100;

job.ResultPath = modelResult.CloudPath;

await \_context.SaveChangesAsync();

// Optionally deploy locally immediately

await DeployModelLocallyAsync(model.Id);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Model training failed for job {JobId}", jobId);

job.Status = JobStatus.Failed;

job.ErrorMessage = ex.Message;

await \_context.SaveChangesAsync();

}

}

private object GetDefaultTrainingParameters()

{

return new

{

LearningRate = 0.001,

BatchSize = 32,

Epochs = 100,

ValidationSplit = 0.2,

EarlyStopping = true,

Patience = 10

};

}

private async Task<string> GetCurrentModelVersionAsync()

{

return $"v{DateTime.UtcNow:yyyyMMdd}";

}

}

// Supporting DTOs and classes

public class AnswerComparisonResult

{

public bool IsCorrect { get; set; }

public decimal PartialCredit { get; set; }

public decimal Similarity { get; set; }

}

public class ExerciseContentDto

{

public List<ExerciseItemDto> Items { get; set; } = new();

}

public class ExerciseItemDto

{

public string Content { get; set; } = "";

public string ExpectedResponse { get; set; } = "";

public string ItemType { get; set; } = "";

}

public class HandwritingProcessingException : Exception

{

public HandwritingProcessingException(string message) : base(message) { }

public HandwritingProcessingException(string message, Exception innerException) : base(message, innerException) { }

}

}// DTOs/SchoolDtos.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.DTOs

{

public class SchoolDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string Website { get; set; } = string.Empty;

public string RegistrationNumber { get; set; } = string.Empty;

public string Logo { get; set; } = string.Empty;

public GradingSchemeDto? GradingScheme { get; set; }

public DateTime CreatedAt { get; set; }

}

public class CreateSchoolDto

{

[Required]

[StringLength(200)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Address { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(100)]

public string Website { get; set; } = string.Empty;

[StringLength(50)]

public string RegistrationNumber { get; set; } = string.Empty;

public Guid? GradingSchemeId { get; set; }

}

public class UpdateSchoolDto

{

[StringLength(200)]

public string? Name { get; set; }

[StringLength(500)]

public string? Address { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(100)]

public string? Website { get; set; }

public Guid? GradingSchemeId { get; set; }

}

}

// DTOs/StudentDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class StudentDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string StudentNumber { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public GradeDto CurrentGrade { get; set; } = null!;

public SchoolDto School { get; set; } = null!;

public List<ParentDto> Parents { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class CreateStudentDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[StringLength(50)]

public string? StudentNumber { get; set; }

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

[StringLength(10)]

public string Gender { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid CurrentGradeId { get; set; }

public List<CreateParentDto> Parents { get; set; } = new();

}

public class UpdateStudentDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[StringLength(200)]

public string? Address { get; set; }

public Guid? CurrentGradeId { get; set; }

}

public class StudentTimetableDto

{

public StudentDto Student { get; set; } = null!;

public List<TimetableSlotDto> TimetableSlots { get; set; } = new();

public string TimetableName { get; set; } = string.Empty;

}

}

// DTOs/TeacherDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TeacherDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string EmployeeId { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public DateTime HireDate { get; set; }

public string Qualification { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public SchoolDto School { get; set; } = null!;

}

public class CreateTeacherDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[Required]

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[StringLength(50)]

public string EmployeeId { get; set; } = string.Empty;

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

public DateTime HireDate { get; set; }

[StringLength(100)]

public string Qualification { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

}

public class UpdateTeacherDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[StringLength(200)]

public string? Address { get; set; }

[StringLength(100)]

public string? Qualification { get; set; }

}

public class AssignTeacherSubjectDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class AssignClassTeacherDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class TeacherTimetableDto

{

public TeacherDto Teacher { get; set; } = null!;

public List<TimetableSlotDto> TimetableSlots { get; set; } = new();

public TermDto Term { get; set; } = null!;

}

}

// DTOs/GradingDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class GradingSchemeDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public string Country { get; set; } = string.Empty;

public bool IsActive { get; set; }

public List<GradingScaleDto> GradingScales { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class CreateGradingSchemeDto

{

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Description { get; set; } = string.Empty;

[StringLength(50)]

public string Country { get; set; } = string.Empty;

public List<CreateGradingScaleDto> GradingScales { get; set; } = new();

}

public class UpdateGradingSchemeDto

{

[StringLength(100)]

public string? Name { get; set; }

[StringLength(500)]

public string? Description { get; set; }

[StringLength(50)]

public string? Country { get; set; }

public bool? IsActive { get; set; }

}

public class GradingScaleDto

{

public Guid Id { get; set; }

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class CreateGradingScaleDto

{

[Required]

[StringLength(10)]

public string Symbol { get; set; } = string.Empty;

[Required]

public int Unit { get; set; }

[Required]

[Range(0, 100)]

public decimal MinPercentage { get; set; }

[Required]

[Range(0, 100)]

public decimal MaxPercentage { get; set; }

[StringLength(100)]

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class UpdateGradingScaleDto

{

[StringLength(10)]

public string? Symbol { get; set; }

public int? Unit { get; set; }

[Range(0, 100)]

public decimal? MinPercentage { get; set; }

[Range(0, 100)]

public decimal? MaxPercentage { get; set; }

[StringLength(100)]

public string? Description { get; set; }

public int? SortOrder { get; set; }

}

public class GradeCalculationDto

{

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public string Description { get; set; } = string.Empty;

public decimal Percentage { get; set; }

}

}

// DTOs/ResultDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ResultDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public decimal Score { get; set; }

public decimal TotalMarks { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public int GradeUnit { get; set; }

public string AssessmentType { get; set; } = string.Empty;

public string Comments { get; set; } = string.Empty;

public DateTime AssessmentDate { get; set; }

public bool IsAiVerified { get; set; }

public decimal AiConfidenceScore { get; set; }

}

public class CreateResultDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

[Range(0, double.MaxValue)]

public decimal Score { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal TotalMarks { get; set; }

[Required]

[StringLength(50)]

public string AssessmentType { get; set; } = string.Empty;

[StringLength(500)]

public string Comments { get; set; } = string.Empty;

[Required]

public DateTime AssessmentDate { get; set; }

}

public class UpdateResultDto

{

[Range(0, double.MaxValue)]

public decimal? Score { get; set; }

[Range(0.01, double.MaxValue)]

public decimal? TotalMarks { get; set; }

[StringLength(50)]

public string? AssessmentType { get; set; }

[StringLength(500)]

public string? Comments { get; set; }

public DateTime? AssessmentDate { get; set; }

}

public class StudentReportCardDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public IEnumerable<ResultDto> Results { get; set; } = new List<ResultDto>();

public List<AttendanceSummaryDto> AttendanceSummary { get; set; } = new();

public DateTime GeneratedAt { get; set; }

public decimal OverallAverage { get; set; }

public string OverallGrade { get; set; } = string.Empty;

public int TotalSubjects { get; set; }

public int PassedSubjects { get; set; }

}

public class ClassResultSummaryDto

{

public GradeDto Grade { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<SubjectResultSummaryDto> SubjectSummaries { get; set; } = new();

public int TotalStudents { get; set; }

public decimal OverallAverage { get; set; }

}

public class SubjectResultSummaryDto

{

public SubjectDto Subject { get; set; } = null!;

public int TotalStudents { get; set; }

public decimal AverageScore { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public int PassCount { get; set; }

public int FailCount { get; set; }

public decimal PassRate { get; set; }

}

}

// DTOs/AttendanceDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class AttendanceDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

public string Remarks { get; set; } = string.Empty;

}

public class CreateAttendanceDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public DateTime Date { get; set; }

[Required]

public AttendanceStatus Status { get; set; }

[StringLength(200)]

public string Remarks { get; set; } = string.Empty;

}

public class UpdateAttendanceDto

{

public AttendanceStatus? Status { get; set; }

[StringLength(200)]

public string? Remarks { get; set; }

}

public class AttendanceReportDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<AttendanceSummaryDto> SubjectAttendance { get; set; } = new();

public int TotalClassesTaken { get; set; }

public int TotalClassesAttended { get; set; }

public decimal OverallAttendancePercentage { get; set; }

}

public class AttendanceSummaryDto

{

public Guid SubjectId { get; set; }

public string SubjectName { get; set; } = string.Empty;

public int TotalClasses { get; set; }

public int PresentClasses { get; set; }

public int AbsentClasses { get; set; }

public int LateClasses { get; set; }

public decimal AttendancePercentage { get; set; }

}

}

// DTOs/TimetableDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TimetableDto

{

public Guid Id { get; set; }

public SchoolDto School { get; set; } = null!;

public GradeDto Grade { get; set; } = null!;

public SchoolYearDto SchoolYear { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public string Name { get; set; } = string.Empty;

public bool IsActive { get; set; }

public bool IsGenerated { get; set; }

public DateTime GeneratedAt { get; set; }

public List<TimetableSlotDto> TimetableSlots { get; set; } = new();

}

public class TimetableSlotDto

{

public Guid Id { get; set; }

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public string Room { get; set; } = string.Empty;

public int PeriodNumber { get; set; }

public string Notes { get; set; } = string.Empty;

}

public class GenerateTimetableDto

{

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

public TimetableGenerationRulesDto Rules { get; set; } = new();

}

public class TimetableGenerationRulesDto

{

public List<DayOfWeek> WorkingDays { get; set; } = new()

{

DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday,

DayOfWeek.Thursday, DayOfWeek.Friday

};

public TimeSpan SchoolStartTime { get; set; } = new(8, 0, 0);

public TimeSpan SchoolEndTime { get; set; } = new(15, 0, 0);

public TimeSpan PeriodDuration { get; set; } = new(0, 40, 0);

public TimeSpan BreakDuration { get; set; } = new(0, 15, 0);

public TimeSpan LunchDuration { get; set; } = new(1, 0, 0);

public int MaxPeriodsPerDay { get; set; } = 8;

public int MaxConsecutivePeriods { get; set; } = 3;

// Subject-specific rules

public Dictionary<Guid, int> SubjectPeriodsPerWeek { get; set; } = new();

public Dictionary<Guid, List<DayOfWeek>> SubjectPreferredDays { get; set; } = new();

public Dictionary<Guid, List<int>> SubjectPreferredPeriods { get; set; } = new();

// Teacher constraints

public Dictionary<Guid, List<TimeSlotDto>> TeacherUnavailableSlots { get; set; } = new();

// Room assignments

public Dictionary<Guid, string> SubjectRoomPreferences { get; set; } = new();

}

public class TimeSlotDto

{

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

}

public class UpdateTimetableSlotDto

{

public Guid? SubjectId { get; set; }

public Guid? TeacherId { get; set; }

public TimeSpan? StartTime { get; set; }

public TimeSpan? EndTime { get; set; }

public string? Room { get; set; }

public string? Notes { get; set; }

}

public class TimetableConflictDto

{

public bool HasConflicts { get; set; }

public List<ConflictDto> Conflicts { get; set; } = new();

}

public class ConflictDto

{

public ConflictType Type { get; set; }

public string Description { get; set; } = string.Empty;

public List<Guid> AffectedSlotIds { get; set; } = new();

public string Severity { get; set; } = string.Empty;

}

public enum ConflictType

{

TeacherDoubleBooking,

RoomDoubleBooking,

SubjectOverallocation,

InvalidTimeSlot,

ConsecutivePeriodViolation

}

}

// DTOs/NotificationDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class NotificationDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Message { get; set; } = string.Empty;

public NotificationType Type { get; set; }

public NotificationChannel Channel { get; set; }

public bool IsRead { get; set; }

public bool IsSent { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? SentAt { get; set; }

public DateTime? ReadAt { get; set; }

public string ExternalId { get; set; } = string.Empty;

public ParentDto? Parent { get; set; }

public StudentDto? Student { get; set; }

public TeacherDto? Teacher { get; set; }

}

public class CreateNotificationDto

{

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[Required]

public string Message { get; set; } = string.Empty;

[Required]

public NotificationType Type { get; set; }

[Required]

public NotificationChannel Channel { get; set; }

public Guid? ParentId { get; set; }

public Guid? StudentId { get; set; }

public Guid? TeacherId { get; set; }

public string Metadata { get; set; } = string.Empty;

}

public class NotificationStatsDto

{

public int TotalNotifications { get; set; }

public int SentNotifications { get; set; }

public int PendingNotifications { get; set; }

public int ReadNotifications { get; set; }

public Dictionary<NotificationType, int> NotificationsByType { get; set; } = new();

public Dictionary<NotificationChannel, int> NotificationsByChannel { get; set; } = new();

}

}

// DTOs/HandwritingAIDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ProcessHandwritingDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public string ImagePath { get; set; } = string.Empty;

public Guid? SubjectId { get; set; }

public Guid? ResultId { get; set; }

public string? ExpectedAnswer { get; set; }

}

public class HandwritingRecognitionResultDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string RecognizedText { get; set; } = string.Empty;

public decimal OverallConfidence { get; set; }

public bool ProcessedLocally { get; set; }

public bool ProcessedInCloud { get; set; }

public TimeSpan ProcessingTime { get; set; }

public string ExpectedAnswer { get; set; } = string.Empty;

public bool IsCorrect { get; set; }

public decimal PartialCreditScore { get; set; }

public DateTime ProcessedAt { get; set; }

public List<string> AlternativeTexts { get; set; } = new();

public List<decimal> CharacterConfidences { get; set; } = new();

}

public class GenerateTrainingExerciseDto

{

[Required]

public Guid SchoolId { get; set; }

public Guid? GradeId { get; set; }

public Guid? SubjectId { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

[Required]

public ExerciseType Type { get; set; }

[Required]

public DifficultyLevel Difficulty { get; set; }

public int EstimatedMinutes { get; set; } = 10;

}

public class TrainingExerciseDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public ExerciseType Type { get; set; }

public DifficultyLevel Difficulty { get; set; }

public bool IsActive { get; set; }

public bool IsSystemGenerated { get; set; }

public int EstimatedMinutes { get; set; }

public ExerciseContentDto Content { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class StudentTrainingSessionDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingExerciseDto TrainingExercise { get; set; } = null!;

public DateTime StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public SessionStatus Status { get; set; }

public int TotalItems { get; set; }

public int CompletedItems { get; set; }

public int CorrectItems { get; set; }

public decimal OverallAccuracy { get; set; }

public TimeSpan TotalTime { get; set; }

}

public class SubmitTrainingDataDto

{

[Required]

public Guid SessionId { get; set; }

[Required]

public string OriginalContent { get; set; } = string.Empty;

[Required]

public byte[] HandwritingImage { get; set; } = Array.Empty<byte>();

}

public class HandwritingTrainingDataDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingDataType DataType { get; set; }

public string OriginalContent { get; set; } = string.Empty;

public string ImagePath { get; set; } = string.Empty;

public string ProcessedText { get; set; } = string.Empty;

public string VerifiedText { get; set; } = string.Empty;

public decimal ConfidenceScore { get; set; }

public TrainingStatus Status { get; set; }

public bool IsVerified { get; set; }

public bool IsTrainingComplete { get; set; }

public DateTime CapturedAt { get; set; }

public DateTime? VerifiedAt { get; set; }

public DateTime? TrainedAt { get; set; }

public StudentDto Student { get; set; } = null!;

}

public class ModelTrainingJobDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string JobName { get; set; } = string.Empty;

public JobType Type { get; set; }

public JobStatus Status { get; set; }

public int TotalDataPoints { get; set; }

public int ProcessedDataPoints { get; set; }

public decimal Progress { get; set; }

public DateTime QueuedAt { get; set; }

public DateTime? StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public string ErrorMessage { get; set; } = string.Empty;

public string ResultPath { get; set; } = string.Empty;

}

public class HandwritingModelDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string ModelName { get; set; } = string.Empty;

public string Version { get; set; } = string.Empty;

public ModelType Type { get; set; }

public decimal Accuracy { get; set; }

public int TrainingDataCount { get; set; }

public bool IsActive { get; set; }}// DTOs/SchoolDtos.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.DTOs

{

public class SchoolDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string Website { get; set; } = string.Empty;

public string RegistrationNumber { get; set; } = string.Empty;

public string Logo { get; set; } = string.Empty;

public GradingSchemeDto? GradingScheme { get; set; }

public DateTime CreatedAt { get; set; }

}

public class CreateSchoolDto

{

[Required]

[StringLength(200)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Address { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(100)]

public string Website { get; set; } = string.Empty;

[StringLength(50)]

public string RegistrationNumber { get; set; } = string.Empty;

public Guid? GradingSchemeId { get; set; }

}

public class UpdateSchoolDto

{

[StringLength(200)]

public string? Name { get; set; }

[StringLength(500)]

public string? Address { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(100)]

public string? Website { get; set; }

public Guid? GradingSchemeId { get; set; }

}

}

// DTOs/StudentDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class StudentDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string StudentNumber { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public GradeDto CurrentGrade { get; set; } = null!;

public SchoolDto School { get; set; } = null!;

public List<ParentDto> Parents { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class CreateStudentDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[StringLength(50)]

public string? StudentNumber { get; set; }

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

[StringLength(10)]

public string Gender { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid CurrentGradeId { get; set; }

public List<CreateParentDto> Parents { get; set; } = new();

}

public class UpdateStudentDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[StringLength(200)]

public string? Address { get; set; }

public Guid? CurrentGradeId { get; set; }

}

public class StudentTimetableDto

{

public StudentDto Student { get; set; } = null!;

public List<TimetableSlotDto> TimetableSlots { get; set; } = new();

public string TimetableName { get; set; } = string.Empty;

}

}

// DTOs/TeacherDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TeacherDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string EmployeeId { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public DateTime HireDate { get; set; }

public string Qualification { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public SchoolDto School { get; set; } = null!;

}

public class CreateTeacherDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[Required]

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[StringLength(50)]

public string EmployeeId { get; set; } = string.Empty;

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

public DateTime HireDate { get; set; }

[StringLength(100)]

public string Qualification { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

}

public class UpdateTeacherDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[StringLength(200)]

public string? Address { get; set; }

[StringLength(100)]

public string? Qualification { get; set; }

}

public class AssignTeacherSubjectDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class AssignClassTeacherDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class TeacherTimetableDto

{

public TeacherDto Teacher { get; set; } = null!;

public List<TimetableSlotDto> TimetableSlots { get; set; } = new();

public TermDto Term { get; set; } = null!;

}

}

// DTOs/GradingDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class GradingSchemeDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public string Country { get; set; } = string.Empty;

public bool IsActive { get; set; }

public bool IsDeployedLocally { get; set; }

public bool IsDeployedCloud { get; set; }

public DateTime TrainingStarted { get; set; }

public DateTime? TrainingCompleted { get; set; }

public DateTime? LastUsed { get; set; }

public string ModelFilePath { get; set; } = string.Empty;

public string CloudModelPath { get; set; } = string.Empty;

public Dictionary<string, object> PerformanceMetrics { get; set; } = new();

}

public class TrainingProgressDto

{

public Guid StudentId { get; set; }

public int TotalTrainingData { get; set; }

public int VerifiedData { get; set; }

public int PendingVerification { get; set; }

public decimal CurrentModelAccuracy { get; set; }

public bool ModelIsDeployedLocally { get; set; }

public List<StudentTrainingSessionDto> RecentSessions { get; set; } = new();

public bool ReadyForTraining { get; set; }

public decimal CompletionPercentage { get; set; }

}

}

// DTOs/CoreDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class GradeDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public int Level { get; set; }

public int MaxStudents { get; set; }

public SchoolDto School { get; set; } = null!;

}

public class SubjectDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Code { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public int Credits { get; set; }

public bool IsActive { get; set; }

public SchoolDto School { get; set; } = null!;

}

public class TermDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public int TermNumber { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public bool IsCurrent { get; set; }

public SchoolYearDto SchoolYear { get; set; } = null!;

}

public class SchoolYearDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public bool IsCurrent { get; set; }

public SchoolDto School { get; set; } = null!;

public List<TermDto> Terms { get; set; } = new();

}

public class ParentDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string WhatsAppNumber { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string Relationship { get; set; } = string.Empty;

public string Occupation { get; set; } = string.Empty;

}

public class CreateParentDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[Required]

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(20)]

public string WhatsAppNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[Required]

[StringLength(50)]

public string Relationship { get; set; } = string.Empty;

[StringLength(100)]

public string Occupation { get; set; } = string.Empty;

}

}

// Configuration/AutoMapperProfile.cs

using AutoMapper;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

namespace SchoolManagement.Configuration

{

public class AutoMapperProfile : Profile

{

public AutoMapperProfile()

{

// School mappings

CreateMap<School, SchoolDto>().ReverseMap();

CreateMap<CreateSchoolDto, School>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateSchoolDto, School>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Student mappings

CreateMap<Student, StudentDto>().ReverseMap();

CreateMap<CreateStudentDto, Student>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.Parents, opt => opt.MapFrom(src => src.Parents));

CreateMap<UpdateStudentDto, Student>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Teacher mappings

CreateMap<Teacher, TeacherDto>().ReverseMap();

CreateMap<CreateTeacherDto, Teacher>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateTeacherDto, Teacher>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Parent mappings

CreateMap<Parent, ParentDto>().ReverseMap();

CreateMap<CreateParentDto, Parent>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

// Grading mappings

CreateMap<GradingScheme, GradingSchemeDto>().ReverseMap();

CreateMap<CreateGradingSchemeDto, GradingScheme>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.IsActive, opt => opt.MapFrom(src => true));

CreateMap<UpdateGradingSchemeDto, GradingScheme>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

CreateMap<GradingScale, GradingScaleDto>().ReverseMap();

CreateMap<CreateGradingScaleDto, GradingScale>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateGradingScaleDto, GradingScale>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Core entity mappings

CreateMap<Grade, GradeDto>().ReverseMap();

CreateMap<Subject, SubjectDto>().ReverseMap();

CreateMap<Term, TermDto>().ReverseMap();

CreateMap<SchoolYear, SchoolYearDto>().ReverseMap();

// Result mappings

CreateMap<Result, ResultDto>().ReverseMap();

CreateMap<CreateResultDto, Result>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.Percentage, opt => opt.Ignore())

.ForMember(dest => dest.Grade, opt => opt.Ignore())

.ForMember(dest => dest.GradeUnit, opt => opt.Ignore());

CreateMap<UpdateResultDto, Result>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Attendance mappings

CreateMap<Attendance, AttendanceDto>().ReverseMap();

CreateMap<CreateAttendanceDto, Attendance>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateAttendanceDto, Attendance>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Timetable mappings

CreateMap<Timetable, TimetableDto>().ReverseMap();

CreateMap<TimetableSlot, TimetableSlotDto>().ReverseMap();

CreateMap<UpdateTimetableSlotDto, TimetableSlot>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Notification mappings

CreateMap<Notification, NotificationDto>().ReverseMap();

CreateMap<CreateNotificationDto, Notification>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.IsRead, opt => opt.MapFrom(src => false))

.ForMember(dest => dest.IsSent, opt => opt.MapFrom(src => false));

// AI Handwriting mappings

CreateMap<HandwritingTrainingData, HandwritingTrainingDataDto>().ReverseMap();

CreateMap<HandwritingRecognitionResult, HandwritingRecognitionResultDto>()

.ForMember(dest => dest.AlternativeTexts, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.AlternativeTexts) ?

new List<string>() :

System.Text.Json.JsonSerializer.Deserialize<List<string>>(src.AlternativeTexts)))

.ForMember(dest => dest.CharacterConfidences, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.CharacterConfidences) ?

new List<decimal>() :

System.Text.Json.JsonSerializer.Deserialize<List<decimal>>(src.CharacterConfidences)));

CreateMap<TrainingExercise, TrainingExerciseDto>()

.ForMember(dest => dest.Content, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.Content) ?

new ExerciseContentDto() :

System.Text.Json.JsonSerializer.Deserialize<ExerciseContentDto>(src.Content)));

CreateMap<StudentTrainingSession, StudentTrainingSessionDto>().ReverseMap();

CreateMap<ModelTrainingJob, ModelTrainingJobDto>().ReverseMap();

CreateMap<HandwritingModel, HandwritingModelDto>()

.ForMember(dest => dest.PerformanceMetrics, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.PerformanceMetrics) ?

new Dictionary<string, object>() :

System.Text.Json.JsonSerializer.Deserialize<Dictionary<string, object>>(src.PerformanceMetrics)));

}

}

}

// Services/Interfaces/Supporting/IImageProcessingService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IImageProcessingService

{

Task<ImageProcessingResult> ProcessImageAsync(string imagePath);

Task<TextExtractionResult> ExtractTextAsync(string imagePath);

Task<List<CharacterSegment>> SegmentCharactersAsync(string imagePath);

Task<string> PreprocessImageAsync(string imagePath);

Task<bool> ValidateImageQualityAsync(string imagePath);

}

public class ImageProcessingResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public string BoundingBoxes { get; set; } = string.Empty;

public string CharacterConfidences { get; set; } = string.Empty;

public string AlternativeTexts { get; set; } = string.Empty;

}

public class TextExtractionResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public List<CharacterSegment> Characters { get; set; } = new();

}

public class CharacterSegment

{

public string ImagePath { get; set; } = string.Empty;

public Rectangle BoundingBox { get; set; }

public decimal Confidence { get; set; }

public string FeatureVector { get; set; } = string.Empty;

}

public struct Rectangle

{

public int X { get; set; }

public int Y { get; set; }

public int Width { get; set; }

public int Height { get; set; }

}

}

// Services/Interfaces/Supporting/IMachineLearningService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IMachineLearningService

{

Task<LocalRecognitionResult> RecognizeTextLocallyAsync(string modelPath, string imagePath);

Task<CloudRecognitionResult> RecognizeTextInCloudAsync(string imagePath);

Task<ModelTrainingResult> TrainPersonalModelAsync(List<HandwritingTrainingData> trainingData);

Task<bool> ValidateModelAsync(string modelPath);

Task<ModelMetrics> EvaluateModelAsync(string modelPath, List<HandwritingTrainingData> testData);

Task<string> DeployModelToCloudAsync(string localModelPath);

Task<bool> UpdateModelAsync(string existingModelPath, List<HandwritingTrainingData> newTrainingData);

}

public class LocalRecognitionResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public TimeSpan ProcessingTime { get; set; }

public List<CharacterRecognition> Characters { get; set; } = new();

}

public class CloudRecognitionResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public TimeSpan ProcessingTime { get; set; }

public List<string> Alternatives { get; set; } = new();

}

public class ModelTrainingResult

{

public string CloudPath { get; set; } = string.Empty;

public string LocalPath { get; set; } = string.Empty;

public decimal Accuracy { get; set; }

public ModelMetrics Metrics { get; set; } = new();

public TimeSpan TrainingTime { get; set; }

}

public class ModelMetrics

{

public decimal Accuracy { get; set; }

public decimal Precision { get; set; }

public decimal Recall { get; set; }

public decimal F1Score { get; set; }

public Dictionary<string, decimal> CharacterAccuracies { get; set; } = new();

public int TotalSamples { get; set; }

public int CorrectPredictions { get; set; }

}

public class CharacterRecognition

{

public string Character { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public Rectangle BoundingBox { get; set; }

public List<string> Alternatives { get; set; } = new();

}

}

// Services/Interfaces/Supporting/IFileStorageService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IFileStorageService

{

Task<string> SaveImageAsync(byte[] imageData, string relativePath);

Task<string> SaveFileAsync(byte[] fileData, string relativePath, string contentType);

Task<byte[]> GetFileAsync(string filePath);

Task<bool> DeleteFileAsync(string filePath);

Task<string> DownloadModelAsync(string cloudPath, Guid modelId);

Task<string> UploadModelAsync(string localPath, string cloudPath);

Task<List<string>> GetFilesInDirectoryAsync(string directoryPath);

Task<bool> FileExistsAsync(string filePath);

Task<long> GetFileSizeAsync(string filePath);

}

}

// Services/Interfaces/Supporting/IPdfGenerationService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IPdfGenerationService

{

Task<byte[]> GenerateReportCardPdfAsync(StudentReportCardDto reportCard);

Task<byte[]> GenerateClassResultsPdfAsync(ClassResultSummaryDto classSummary);

Task<byte[]> GenerateTimetablePdfAsync(TimetableDto timetable);

Task<byte[]> GenerateAttendanceReportPdfAsync(AttendanceReportDto attendanceReport);

Task<byte[]> GenerateCertificatePdfAsync(StudentDto student, string certificateType);

Task<byte[]> GenerateTranscriptPdfAsync(StudentDto student, List<ResultDto> results);

}

public class PdfGenerationService : IPdfGenerationService

{

public async Task<byte[]> GenerateReportCardPdfAsync(StudentReportCardDto reportCard)

{

// Implementation using iTextSharp, PdfSharp, or similar library

// This is a placeholder implementation

await Task.Delay(100); // Simulate processing time

return new byte[0]; // Return generated PDF bytes

}

public async Task<byte[]> GenerateClassResultsPdfAsync(ClassResultSummaryDto classSummary)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateTimetablePdfAsync(TimetableDto timetable)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateAttendanceReportPdfAsync(AttendanceReportDto attendanceReport)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateCertificatePdfAsync(StudentDto student, string certificateType)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateTranscriptPdfAsync(StudentDto student, List<ResultDto> results)

{

await Task.Delay(100);

return new byte[0];

}

}

}

// Exceptions/CustomExceptions.cs

namespace SchoolManagement.Core.Exceptions

{

public class NotFoundException : Exception

{

public NotFoundException(string message) : base(message) { }

public NotFoundException(string message, Exception innerException) : base(message, innerException) { }

}

public class ValidationException : Exception

{

public Dictionary<string, List<string>> Errors { get; }

public ValidationException(string message) : base(message)

{

Errors = new Dictionary<string, List<string>>();

}

public ValidationException(Dictionary<string, List<string>> errors) : base("Validation failed")

{

Errors = errors;

}

}

public class BusinessRuleException : Exception

{

public string RuleCode { get; }

public BusinessRuleException(string ruleCode, string message) : base(message)

{

RuleCode = ruleCode;

}

}

public class UnauthorizedException : Exception

{

public UnauthorizedException(string message) : base(message) { }

}

public class ConflictException : Exception

{

public ConflictException(string message) : base(message) { }

}

}

public List<GradingScaleDto> GradingScales { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class CreateGradingSchemeDto

{

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Description { get; set; } = string.Empty;

[StringLength(50)]

public string Country { get; set; } = string.Empty;

public List<CreateGradingScaleDto> GradingScales { get; set; } = new();

}

public class UpdateGradingSchemeDto

{

[StringLength(100)]

public string? Name { get; set; }

[StringLength(500)]

public string? Description { get; set; }

[StringLength(50)]

public string? Country { get; set; }

public bool? IsActive { get; set; }

}

public class GradingScaleDto

{

public Guid Id { get; set; }

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class CreateGradingScaleDto

{

[Required]

[StringLength(10)]

public string Symbol { get; set; } = string.Empty;

[Required]

public int Unit { get; set; }

[Required]

[Range(0, 100)]

public decimal MinPercentage { get; set; }

[Required]

[Range(0, 100)]

public decimal MaxPercentage { get; set; }

[StringLength(100)]

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class UpdateGradingScaleDto

{

[StringLength(10)]

public string? Symbol { get; set; }

public int? Unit { get; set; }

[Range(0, 100)]

public decimal? MinPercentage { get; set; }

[Range(0, 100)]

public decimal? MaxPercentage { get; set; }

[StringLength(100)]

public string? Description { get; set; }

public int? SortOrder { get; set; }

}

public class GradeCalculationDto

{

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public string Description { get; set; } = string.Empty;

public decimal Percentage { get; set; }

}

}

// DTOs/ResultDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ResultDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public decimal Score { get; set; }

public decimal TotalMarks { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public int GradeUnit { get; set; }

public string AssessmentType { get; set; } = string.Empty;

public string Comments { get; set; } = string.Empty;

public DateTime AssessmentDate { get; set; }

public bool IsAiVerified { get; set; }

public decimal AiConfidenceScore { get; set; }

}

public class CreateResultDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

[Range(0, double.MaxValue)]

public decimal Score { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal TotalMarks { get; set; }

[Required]

[StringLength(50)]

public string AssessmentType { get; set; } = string.Empty;

[StringLength(500)]

public string Comments { get; set; } = string.Empty;

[Required]

public DateTime AssessmentDate { get; set; }

}

public class UpdateResultDto

{

[Range(0, double.MaxValue)]

public decimal? Score { get; set; }

[Range(0.01, double.MaxValue)]

public decimal? TotalMarks { get; set; }

[StringLength(50)]

public string? AssessmentType { get; set; }

[StringLength(500)]

public string? Comments { get; set; }

public DateTime? AssessmentDate { get; set; }

}

public class StudentReportCardDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public IEnumerable<ResultDto> Results { get; set; } = new List<ResultDto>();

public List<AttendanceSummaryDto> AttendanceSummary { get; set; } = new();

public DateTime GeneratedAt { get; set; }

public decimal OverallAverage { get; set; }

public string OverallGrade { get; set; } = string.Empty;

public int TotalSubjects { get; set; }

public int PassedSubjects { get; set; }

}

public class ClassResultSummaryDto

{

public GradeDto Grade { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<SubjectResultSummaryDto> SubjectSummaries { get; set; } = new();

public int TotalStudents { get; set; }

public decimal OverallAverage { get; set; }

}

public class SubjectResultSummaryDto

{

public SubjectDto Subject { get; set; } = null!;

public int TotalStudents { get; set; }

public decimal AverageScore { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public int PassCount { get; set; }

public int FailCount { get; set; }

public decimal PassRate { get; set; }

}

}

// DTOs/AttendanceDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class AttendanceDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

public string Remarks { get; set; } = string.Empty;

}

public class CreateAttendanceDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public DateTime Date { get; set; }

[Required]

public AttendanceStatus Status { get; set; }

[StringLength(200)]

public string Remarks { get; set; } = string.Empty;

}

public class UpdateAttendanceDto

{

public AttendanceStatus? Status { get; set; }

[StringLength(200)]

public string? Remarks { get; set; }

}

public class AttendanceReportDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<AttendanceSummaryDto> SubjectAttendance { get; set; } = new();

public int TotalClassesTaken { get; set; }

public int TotalClassesAttended { get; set; }

public decimal OverallAttendancePercentage { get; set; }

}

public class AttendanceSummaryDto

{

public Guid SubjectId { get; set; }

public string SubjectName { get; set; } = string.Empty;

public int TotalClasses { get; set; }

public int PresentClasses { get; set; }

public int AbsentClasses { get; set; }

public int LateClasses { get; set; }

public decimal AttendancePercentage { get; set; }

}

}

// DTOs/TimetableDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TimetableDto

{

public Guid Id { get; set; }

public SchoolDto School { get; set; } = null!;

public GradeDto Grade { get; set; } = null!;

public SchoolYearDto SchoolYear { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public string Name { get; set; } = string.Empty;

public bool IsActive { get; set; }

public bool IsGenerated { get; set; }

public DateTime GeneratedAt { get; set; }

public List<TimetableSlotDto> TimetableSlots { get; set; } = new();

}

public class TimetableSlotDto

{

public Guid Id { get; set; }

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public string Room { get; set; } = string.Empty;

public int PeriodNumber { get; set; }

public string Notes { get; set; } = string.Empty;

}

public class GenerateTimetableDto

{

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

public TimetableGenerationRulesDto Rules { get; set; } = new();

}

public class TimetableGenerationRulesDto

{

public List<DayOfWeek> WorkingDays { get; set; } = new()

{

DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday,

DayOfWeek.Thursday, DayOfWeek.Friday

};

public TimeSpan SchoolStartTime { get; set; } = new(8, 0, 0);

public TimeSpan SchoolEndTime { get; set; } = new(15, 0, 0);

public TimeSpan PeriodDuration { get; set; } = new(0, 40, 0);

public TimeSpan BreakDuration { get; set; } = new(0, 15, 0);

public TimeSpan LunchDuration { get; set; } = new(1, 0, 0);

public int MaxPeriodsPerDay { get; set; } = 8;

public int MaxConsecutivePeriods { get; set; } = 3;

// Subject-specific rules

public Dictionary<Guid, int> SubjectPeriodsPerWeek { get; set; } = new();

public Dictionary<Guid, List<DayOfWeek>> SubjectPreferredDays { get; set; } = new();

public Dictionary<Guid, List<int>> SubjectPreferredPeriods { get; set; } = new();

// Teacher constraints

public Dictionary<Guid, List<TimeSlotDto>> TeacherUnavailableSlots { get; set; } = new();

// Room assignments

public Dictionary<Guid, string> SubjectRoomPreferences { get; set; } = new();

}

public class TimeSlotDto

{

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

}

public class UpdateTimetableSlotDto

{

public Guid? SubjectId { get; set; }

public Guid? TeacherId { get; set; }

public TimeSpan? StartTime { get; set; }

public TimeSpan? EndTime { get; set; }

public string? Room { get; set; }

public string? Notes { get; set; }

}

public class TimetableConflictDto

{

public bool HasConflicts { get; set; }

public List<ConflictDto> Conflicts { get; set; } = new();

}

public class ConflictDto

{

public ConflictType Type { get; set; }

public string Description { get; set; } = string.Empty;

public List<Guid> AffectedSlotIds { get; set; } = new();

public string Severity { get; set; } = string.Empty;

}

public enum ConflictType

{

TeacherDoubleBooking,

RoomDoubleBooking,

SubjectOverallocation,

InvalidTimeSlot,

ConsecutivePeriodViolation

}

}

// DTOs/NotificationDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class NotificationDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Message { get; set; } = string.Empty;

public NotificationType Type { get; set; }

public NotificationChannel Channel { get; set; }

public bool IsRead { get; set; }

public bool IsSent { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? SentAt { get; set; }

public DateTime? ReadAt { get; set; }

public string ExternalId { get; set; } = string.Empty;

public ParentDto? Parent { get; set; }

public StudentDto? Student { get; set; }

public TeacherDto? Teacher { get; set; }

}

public class CreateNotificationDto

{

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[Required]

public string Message { get; set; } = string.Empty;

[Required]

public NotificationType Type { get; set; }

[Required]

public NotificationChannel Channel { get; set; }

public Guid? ParentId { get; set; }

public Guid? StudentId { get; set; }

public Guid? TeacherId { get; set; }

public string Metadata { get; set; } = string.Empty;

}

public class NotificationStatsDto

{

public int TotalNotifications { get; set; }

public int SentNotifications { get; set; }

public int PendingNotifications { get; set; }

public int ReadNotifications { get; set; }

public Dictionary<NotificationType, int> NotificationsByType { get; set; } = new();

public Dictionary<NotificationChannel, int> NotificationsByChannel { get; set; } = new();

}

}

// DTOs/HandwritingAIDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ProcessHandwritingDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public string ImagePath { get; set; } = string.Empty;

public Guid? SubjectId { get; set; }

public Guid? ResultId { get; set; }

public string? ExpectedAnswer { get; set; }

}

public class HandwritingRecognitionResultDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string RecognizedText { get; set; } = string.Empty;

public decimal OverallConfidence { get; set; }

public bool ProcessedLocally { get; set; }

public bool ProcessedInCloud { get; set; }

public TimeSpan ProcessingTime { get; set; }

public string ExpectedAnswer { get; set; } = string.Empty;

public bool IsCorrect { get; set; }

public decimal PartialCreditScore { get; set; }

public DateTime ProcessedAt { get; set; }

public List<string> AlternativeTexts { get; set; } = new();

public List<decimal> CharacterConfidences { get; set; } = new();

}

public class GenerateTrainingExerciseDto

{

[Required]

public Guid SchoolId { get; set; }

public Guid? GradeId { get; set; }

public Guid? SubjectId { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

[Required]

public ExerciseType Type { get; set; }

[Required]

public DifficultyLevel Difficulty { get; set; }

public int EstimatedMinutes { get; set; } = 10;

}

public class TrainingExerciseDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public ExerciseType Type { get; set; }

public DifficultyLevel Difficulty { get; set; }

public bool IsActive { get; set; }

public bool IsSystemGenerated { get; set; }

public int EstimatedMinutes { get; set; }

public ExerciseContentDto Content { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class StudentTrainingSessionDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingExerciseDto TrainingExercise { get; set; } = null!;

public DateTime StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public SessionStatus Status { get; set; }

public int TotalItems { get; set; }

public int CompletedItems { get; set; }

public int CorrectItems { get; set; }

public decimal OverallAccuracy { get; set; }

public TimeSpan TotalTime { get; set; }

}

public class SubmitTrainingDataDto

{

[Required]

public Guid SessionId { get; set; }

[Required]

public string OriginalContent { get; set; } = string.Empty;

[Required]

public byte[] HandwritingImage { get; set; } = Array.Empty<byte>();

}

public class HandwritingTrainingDataDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingDataType DataType { get; set; }

public string OriginalContent { get; set; } = string.Empty;

public string ImagePath { get; set; } = string.Empty;

public string ProcessedText { get; set; } = string.Empty;

public string VerifiedText { get; set; } = string.Empty;

public decimal ConfidenceScore { get; set; }

public TrainingStatus Status { get; set; }

public bool IsVerified { get; set; }

public bool IsTrainingComplete { get; set; }

public DateTime CapturedAt { get; set; }

public DateTime? VerifiedAt { get; set; }

public DateTime? TrainedAt { get; set; }

public StudentDto Student { get; set; } = null!;

}

public class ModelTrainingJobDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string JobName { get; set; } = string.Empty;

public JobType Type { get; set; }

public JobStatus Status { get; set; }

public int TotalDataPoints { get; set; }

public int ProcessedDataPoints { get; set; }

public decimal Progress { get; set; }

public DateTime QueuedAt { get; set; }

public DateTime? StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public string ErrorMessage { get; set; } = string.Empty;

public string ResultPath { get; set; } = string.Empty;

}

public class HandwritingModelDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string ModelName { get; set; } = string.Empty;

public string Version { get; set; } = string.Empty;

public ModelType Type { get; set; }

public decimal Accuracy { get; set; }

public int TrainingDataCount { get; set; }

public bool IsActive { get; set; }// Services/Implementations/TimetableService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class TimetableService : ITimetableService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly ILogger<TimetableService> \_logger;

private readonly IPdfGenerationService \_pdfService;

public TimetableService(

SchoolManagementDbContext context,

IMapper mapper,

ILogger<TimetableService> logger,

IPdfGenerationService pdfService)

{

\_context = context;

\_mapper = mapper;

\_logger = logger;

\_pdfService = pdfService;

}

public async Task<TimetableDto> GenerateTimetableAsync(GenerateTimetableDto generateDto)

{

try

{

\_logger.LogInformation("Starting timetable generation for Grade {GradeId}", generateDto.GradeId);

// Validate input data

await ValidateGenerationInputAsync(generateDto);

// Get required data

var gradeSubjects = await GetGradeSubjectsAsync(generateDto.GradeId, generateDto.SchoolYearId);

var teachers = await GetAvailableTeachersAsync(generateDto.SchoolId, generateDto.SchoolYearId);

// Create timetable entity

var timetable = new Timetable

{

SchoolId = generateDto.SchoolId,

GradeId = generateDto.GradeId,

SchoolYearId = generateDto.SchoolYearId,

TermId = generateDto.TermId,

Name = generateDto.Name,

IsActive = true,

IsGenerated = true,

GeneratedAt = DateTime.UtcNow

};

\_context.Timetables.Add(timetable);

await \_context.SaveChangesAsync();

// Generate timetable slots

var generatedSlots = await GenerateTimetableSlotsAsync(

timetable.Id,

gradeSubjects,

teachers,

generateDto.Rules);

timetable.TimetableSlots = generatedSlots;

await \_context.SaveChangesAsync();

// Validate the generated timetable

var conflicts = await CheckTimetableConflictsAsync(timetable.Id);

if (conflicts.HasConflicts)

{

\_logger.LogWarning("Generated timetable has conflicts: {ConflictCount}", conflicts.Conflicts.Count);

// Attempt to resolve conflicts

await ResolveConflictsAsync(timetable.Id, conflicts);

}

// Create student timetable assignments

await CreateStudentTimetableAssignmentsAsync(timetable.Id, generateDto.GradeId);

var result = await GetTimetableWithDetailsAsync(timetable.Id);

\_logger.LogInformation("Timetable generation completed for Grade {GradeId}", generateDto.GradeId);

return \_mapper.Map<TimetableDto>(result);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error generating timetable for Grade {GradeId}", generateDto.GradeId);

throw;

}

}

public async Task<TimetableDto> GetGradeTimetableAsync(Guid gradeId, Guid termId)

{

var timetable = await \_context.Timetables

.Include(t => t.School)

.Include(t => t.Grade)

.Include(t => t.SchoolYear)

.Include(t => t.Term)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Subject)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Teacher)

.FirstOrDefaultAsync(t => t.GradeId == gradeId && t.TermId == termId && t.IsActive);

if (timetable == null)

throw new NotFoundException($"No active timetable found for grade {gradeId} in term {termId}");

return \_mapper.Map<TimetableDto>(timetable);

}

public async Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.School)

.FirstOrDefaultAsync(s => s.Id == studentId);

if (student == null)

throw new NotFoundException($"Student {studentId} not found");

var currentTerm = await \_context.Terms

.Where(t => t.SchoolYear.SchoolId == student.SchoolId && t.IsCurrent)

.FirstOrDefaultAsync();

if (currentTerm == null)

throw new NotFoundException("No current term found");

var timetable = await GetGradeTimetableAsync(student.CurrentGradeId, currentTerm.Id);

// Get student-specific assignments (for elective subjects)

var studentSpecificSlots = await \_context.StudentTimetables

.Where(st => st.StudentId == studentId && st.IsActive)

.Include(st => st.TimetableSlot)

.ThenInclude(ts => ts.Subject)

.Include(st => st.TimetableSlot)

.ThenInclude(ts => ts.Teacher)

.Select(st => st.TimetableSlot)

.ToListAsync();

// Filter timetable based on student's subjects

var studentSubjects = await \_context.StudentSubjects

.Where(ss => ss.StudentId == studentId && ss.SchoolYearId == timetable.SchoolYear.Id && ss.IsActive)

.Select(ss => ss.SubjectId)

.ToListAsync();

var filteredSlots = timetable.TimetableSlots

.Where(slot => studentSubjects.Contains(slot.Subject.Id))

.Concat(studentSpecificSlots.Select(slot => \_mapper.Map<TimetableSlotDto>(slot)))

.Distinct()

.OrderBy(slot => slot.DayOfWeek)

.ThenBy(slot => slot.PeriodNumber)

.ToList();

return new StudentTimetableDto

{

Student = \_mapper.Map<StudentDto>(student),

TimetableSlots = filteredSlots,

TimetableName = timetable.Name

};

}

public async Task<TeacherTimetableDto> GetTeacherTimetableAsync(Guid teacherId, Guid termId)

{

var teacher = await \_context.Teachers

.Include(t => t.School)

.FirstOrDefaultAsync(t => t.Id == teacherId);

if (teacher == null)

throw new NotFoundException($"Teacher {teacherId} not found");

var term = await \_context.Terms.FindAsync(termId);

if (term == null)

throw new NotFoundException($"Term {termId} not found");

var teacherSlots = await \_context.TimetableSlots

.Where(ts => ts.TeacherId == teacherId && ts.Timetable.TermId == termId)

.Include(ts => ts.Subject)

.Include(ts => ts.Teacher)

.Include(ts => ts.Timetable)

.ThenInclude(t => t.Grade)

.OrderBy(ts => ts.DayOfWeek)

.ThenBy(ts => ts.PeriodNumber)

.ToListAsync();

return new TeacherTimetableDto

{

Teacher = \_mapper.Map<TeacherDto>(teacher),

Term = \_mapper.Map<TermDto>(term),

TimetableSlots = \_mapper.Map<List<TimetableSlotDto>>(teacherSlots)

};

}

public async Task<TimetableSlotDto> UpdateTimetableSlotAsync(Guid slotId, UpdateTimetableSlotDto updateDto)

{

var slot = await \_context.TimetableSlots.FindAsync(slotId);

if (slot == null)

throw new NotFoundException($"Timetable slot {slotId} not found");

// Check for conflicts before updating

if (updateDto.TeacherId.HasValue || updateDto.StartTime.HasValue || updateDto.EndTime.HasValue)

{

var tempSlot = new TimetableSlot

{

Id = slot.Id,

TimetableId = slot.TimetableId,

TeacherId = updateDto.TeacherId ?? slot.TeacherId,

SubjectId = updateDto.SubjectId ?? slot.SubjectId,

DayOfWeek = slot.DayOfWeek,

StartTime = updateDto.StartTime ?? slot.StartTime,

EndTime = updateDto.EndTime ?? slot.EndTime,

Room = updateDto.Room ?? slot.Room,

PeriodNumber = slot.PeriodNumber

};

var conflicts = await ValidateSlotConflictsAsync(tempSlot);

if (conflicts.Any())

{

throw new ConflictException($"Update would create conflicts: {string.Join(", ", conflicts)}");

}

}

\_mapper.Map(updateDto, slot);

await \_context.SaveChangesAsync();

return \_mapper.Map<TimetableSlotDto>(slot);

}

public async Task<bool> ValidateTimetableAsync(Guid timetableId)

{

var conflicts = await CheckTimetableConflictsAsync(timetableId);

return !conflicts.HasConflicts;

}

public async Task<TimetableConflictDto> CheckTimetableConflictsAsync(Guid timetableId)

{

var conflicts = new List<ConflictDto>();

var slots = await \_context.TimetableSlots

.Where(ts => ts.TimetableId == timetableId)

.Include(ts => ts.Teacher)

.Include(ts => ts.Subject)

.ToListAsync();

// Check teacher double booking

var teacherConflicts = slots

.GroupBy(s => new { s.TeacherId, s.DayOfWeek, s.StartTime, s.EndTime })

.Where(g => g.Count() > 1)

.Select(g => new ConflictDto

{

Type = ConflictType.TeacherDoubleBooking,

Description = $"Teacher {g.Key.TeacherId} is double-booked on {g.Key.DayOfWeek} from {g.Key.StartTime} to {g.Key.EndTime}",

AffectedSlotIds = g.Select(s => s.Id).ToList(),

Severity = "High"

});

conflicts.AddRange(teacherConflicts);

// Check room conflicts

var roomConflicts = slots

.Where(s => !string.IsNullOrEmpty(s.Room))

.GroupBy(s => new { s.Room, s.DayOfWeek, s.StartTime, s.EndTime })

.Where(g => g.Count() > 1)

.Select(g => new ConflictDto

{

Type = ConflictType.RoomDoubleBooking,

Description = $"Room {g.Key.Room} is double-booked on {g.Key.DayOfWeek} from {g.Key.StartTime} to {g.Key.EndTime}",

AffectedSlotIds = g.Select(s => s.Id).ToList(),

Severity = "Medium"

});

conflicts.AddRange(roomConflicts);

// Check consecutive period violations

var consecutiveViolations = CheckConsecutivePeriodViolations(slots);

conflicts.AddRange(consecutiveViolations);

return new TimetableConflictDto

{

HasConflicts = conflicts.Any(),

Conflicts = conflicts

};

}

public async Task<byte[]> PrintTimetableAsync(Guid timetableId)

{

var timetable = await GetTimetableWithDetailsAsync(timetableId);

var timetableDto = \_mapper.Map<TimetableDto>(timetable);

return await \_pdfService.GenerateTimetablePdfAsync(timetableDto);

}

// Private helper methods

private async Task ValidateGenerationInputAsync(GenerateTimetableDto generateDto)

{

var school = await \_context.Schools.FindAsync(generateDto.SchoolId);

if (school == null)

throw new NotFoundException($"School {generateDto.SchoolId} not found");

var grade = await \_context.Grades.FindAsync(generateDto.GradeId);

if (grade == null)

throw new NotFoundException($"Grade {generateDto.GradeId} not found");

var term = await \_context.Terms.FindAsync(generateDto.TermId);

if (term == null)

throw new NotFoundException($"Term {generateDto.TermId} not found");

// Check if timetable already exists

var existingTimetable = await \_context.Timetables

.FirstOrDefaultAsync(t => t.GradeId == generateDto.GradeId &&

t.TermId == generateDto.TermId &&

t.IsActive);

if (existingTimetable != null)

{

throw new ConflictException($"Active timetable already exists for this grade and term");

}

}

private async Task<List<GradeSubject>> GetGradeSubjectsAsync(Guid gradeId, Guid schoolYearId)

{

return await \_context.GradeSubjects

.Where(gs => gs.GradeId == gradeId)

.Include(gs => gs.Subject)

.Include(gs => gs.Grade)

.ToListAsync();

}

private async Task<List<Teacher>> GetAvailableTeachersAsync(Guid schoolId, Guid schoolYearId)

{

return await \_context.Teachers

.Where(t => t.SchoolId == schoolId)

.Include(t => t.SubjectTeachers.Where(st => st.SchoolYearId == schoolYearId))

.ThenInclude(st => st.Subject)

.ToListAsync();

}

private async Task<List<TimetableSlot>> GenerateTimetableSlotsAsync(

Guid timetableId,

List<GradeSubject> gradeSubjects,

List<Teacher> teachers,

TimetableGenerationRulesDto rules)

{

var slots = new List<TimetableSlot>();

var random = new Random();

// Calculate periods per day

var totalMinutesPerDay = (int)(rules.SchoolEndTime - rules.SchoolStartTime).TotalMinutes;

var breakMinutes = (int)rules.BreakDuration.TotalMinutes;

var lunchMinutes = (int)rules.LunchDuration.TotalMinutes;

var periodMinutes = (int)rules.PeriodDuration.TotalMinutes;

var availableMinutes = totalMinutesPerDay - breakMinutes - lunchMinutes;

var periodsPerDay = Math.Min(availableMinutes / periodMinutes, rules.MaxPeriodsPerDay);

// Generate time slots for each day

var timeSlots = GenerateTimeSlots(rules, periodsPerDay);

foreach (var gradeSubject in gradeSubjects)

{

// Get assigned teacher for this subject

var assignedTeacher = teachers

.FirstOrDefault(t => t.SubjectTeachers

.Any(st => st.SubjectId == gradeSubject.SubjectId));

if (assignedTeacher == null)

{

\_logger.LogWarning("No teacher assigned to subject {SubjectId}", gradeSubject.SubjectId);

continue;

}

// Determine periods per week for this subject

var periodsPerWeek = rules.SubjectPeriodsPerWeek.GetValueOrDefault(gradeSubject.SubjectId, 3);

// Get preferred days and periods for this subject

var preferredDays = rules.SubjectPreferredDays.GetValueOrDefault(

gradeSubject.SubjectId,

rules.WorkingDays);

var preferredPeriods = rules.SubjectPreferredPeriods.GetValueOrDefault(

gradeSubject.SubjectId,

Enumerable.Range(1, periodsPerDay).ToList());

// Schedule periods for this subject

var scheduledPeriods = 0;

var attempts = 0;

const int maxAttempts = 100;

while (scheduledPeriods < periodsPerWeek && attempts < maxAttempts)

{

attempts++;

// Select random day and period from preferred options

var day = preferredDays[random.Next(preferredDays.Count)];

var period = preferredPeriods[random.Next(preferredPeriods.Count)];

var timeSlot = timeSlots[day][period - 1];

// Check if slot is available

if (IsSlotAvailable(slots, assignedTeacher.Id, day, timeSlot.StartTime, timeSlot.EndTime))

{

var room = rules.SubjectRoomPreferences.GetValueOrDefault(gradeSubject.SubjectId, "");

var slot = new TimetableSlot

{

TimetableId = timetableId,

SubjectId = gradeSubject.SubjectId,

TeacherId = assignedTeacher.Id,

DayOfWeek = day,

StartTime = timeSlot.StartTime,

EndTime = timeSlot.EndTime,

Room = room,

PeriodNumber = period

};

slots.Add(slot);

scheduledPeriods++;

}

}

if (scheduledPeriods < periodsPerWeek)

{

\_logger.LogWarning("Could only schedule {Scheduled}/{Required} periods for subject {SubjectId}",

scheduledPeriods, periodsPerWeek, gradeSubject.SubjectId);

}

}

return slots;

}

private Dictionary<DayOfWeek, List<TimeSlotDto>> GenerateTimeSlots(

TimetableGenerationRulesDto rules,

int periodsPerDay)

{

var timeSlots = new Dictionary<DayOfWeek, List<TimeSlotDto>>();

foreach (var day in rules.WorkingDays)

{

var daySlots = new List<TimeSlotDto>();

var currentTime = rules.SchoolStartTime;

for (int period = 1; period <= periodsPerDay; period++)

{

// Add break after certain periods

if (period == 3) // After 2nd period

{

currentTime = currentTime.Add(rules.BreakDuration);

}

else if (period == 6) // After 5th period (lunch)

{

currentTime = currentTime.Add(rules.LunchDuration);

}

var slot = new TimeSlotDto

{

DayOfWeek = day,

StartTime = currentTime,

EndTime = currentTime.Add(rules.PeriodDuration)

};

daySlots.Add(slot);

currentTime = slot.EndTime;

}

timeSlots[day] = daySlots;

}

return timeSlots;

}

private bool IsSlotAvailable(

List<TimetableSlot> existingSlots,

Guid teacherId,

DayOfWeek day,

TimeSpan startTime,

TimeSpan endTime)

{

return !existingSlots.Any(slot =>

slot.TeacherId == teacherId &&

slot.DayOfWeek == day &&

((startTime >= slot.StartTime && startTime < slot.EndTime) ||

(endTime > slot.StartTime && endTime <= slot.EndTime) ||

(startTime <= slot.StartTime && endTime >= slot.EndTime)));

}

private async Task<List<string>> ValidateSlotConflictsAsync(TimetableSlot slot)

{

var conflicts = new List<string>();

// Check teacher conflicts

var teacherConflict = await \_context.TimetableSlots

.AnyAsync(ts => ts.Id != slot.Id &&

ts.TeacherId == slot.TeacherId &&

ts.DayOfWeek == slot.DayOfWeek &&

((slot.StartTime >= ts.StartTime && slot.StartTime < ts.EndTime) ||

(slot.EndTime > ts.StartTime && slot.EndTime <= ts.EndTime) ||

(slot.StartTime <= ts.StartTime && slot.EndTime >= ts.EndTime)));

if (teacherConflict)

conflicts.Add("Teacher scheduling conflict");

// Check room conflicts

if (!string.IsNullOrEmpty(slot.Room))

{

var roomConflict = await \_context.TimetableSlots

.AnyAsync(ts => ts.Id != slot.Id &&

ts.Room == slot.Room &&

ts.DayOfWeek == slot.DayOfWeek &&

((slot.StartTime >= ts.StartTime && slot.StartTime < ts.EndTime) ||

(slot.EndTime > ts.StartTime && slot.EndTime <= ts.EndTime) ||

(slot.StartTime <= ts.StartTime && slot.EndTime >= ts.EndTime)));

if (roomConflict)

conflicts.Add("Room scheduling conflict");

}

return conflicts;

}

private List<ConflictDto> CheckConsecutivePeriodViolations(List<TimetableSlot> slots)

{

var violations = new List<ConflictDto>();

const int maxConsecutive = 3; // Maximum consecutive periods

var teacherSlots = slots

.GroupBy(s => s.TeacherId)

.ToDictionary(g => g.Key, g => g.ToList());

foreach (var teacherGroup in teacherSlots)

{

var teacherId = teacherGroup.Key;

var teacherPeriods = teacherGroup.Value

.GroupBy(s => s.DayOfWeek)

.ToDictionary(g => g.Key, g => g.OrderBy(s => s.PeriodNumber).ToList());

foreach (var dayGroup in teacherPeriods)

{

var dayPeriods = dayGroup.Value;

var consecutiveCount = 1;

for (int i = 1; i < dayPeriods.Count; i++)

{

if (dayPeriods[i].PeriodNumber == dayPeriods[i - 1].PeriodNumber + 1)

{

consecutiveCount++;

if (consecutiveCount > maxConsecutive)

{

violations.Add(new ConflictDto

{

Type = ConflictType.ConsecutivePeriodViolation,

Description = $"Teacher has {consecutiveCount} consecutive periods on {dayGroup.Key}",

AffectedSlotIds = dayPeriods.Skip(i - consecutiveCount + 1).Take(consecutiveCount).Select(s => s.Id).ToList(),

Severity = "Medium"

});

break;

}

}

else

{

consecutiveCount = 1;

}

}

}

}

return violations;

}

private async Task ResolveConflictsAsync(Guid timetableId, TimetableConflictDto conflicts)

{

// Implement basic conflict resolution logic

foreach (var conflict in conflicts.Conflicts.Where(c => c.Severity == "High"))

{

try

{

await AttemptConflictResolutionAsync(conflict);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to resolve conflict: {ConflictDescription}", conflict.Description);

}

}

}

private async Task AttemptConflictResolutionAsync(ConflictDto conflict)

{

if (conflict.Type == ConflictType.TeacherDoubleBooking)

{

// Try to reschedule one of the conflicting slots

var conflictingSlots = await \_context.TimetableSlots

.Where(ts => conflict.AffectedSlotIds.Contains(ts.Id))

.ToListAsync();

if (conflictingSlots.Count >= 2)

{

var slotToReschedule = conflictingSlots.Last(); // Reschedule the last one

await TryRescheduleSlotAsync(slotToReschedule);

}

}

}

private async Task<bool> TryRescheduleSlotAsync(TimetableSlot slot)

{

var timetable = await \_context.Timetables.FindAsync(slot.TimetableId);

if (timetable == null) return false;

// Try to find an alternative time slot

var workingDays = new[] { DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday, DayOfWeek.Thursday, DayOfWeek.Friday };

var periods = Enumerable.Range(1, 8).ToList();

foreach (var day in workingDays)

{

foreach (var period in periods)

{

var startTime = new TimeSpan(8, 0, 0).Add(TimeSpan.FromMinutes((period - 1) \* 45)); // 45-minute periods

var endTime = startTime.Add(TimeSpan.FromMinutes(40));

var tempSlot = new TimetableSlot

{

Id = slot.Id,

TimetableId = slot.TimetableId,

TeacherId = slot.TeacherId,

SubjectId = slot.SubjectId,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Room = slot.Room,

PeriodNumber = period

};

var conflicts = await ValidateSlotConflictsAsync(tempSlot);

if (!conflicts.Any())

{

// Update the slot

slot.DayOfWeek = day;

slot.StartTime = startTime;

slot.EndTime = endTime;

slot.PeriodNumber = period;

await \_context.SaveChangesAsync();

return true;

}

}

}

return false;

}

private async Task CreateStudentTimetableAssignmentsAsync(Guid timetableId, Guid gradeId)

{

var students = await \_context.Students

.Where(s => s.CurrentGradeId == gradeId)

.ToListAsync();

var timetableSlots = await \_context.TimetableSlots

.Where(ts => ts.TimetableId == timetableId)

.ToListAsync();

var studentTimetables = new List<StudentTimetable>();

foreach (var student in students)

{

// Get student's enrolled subjects

var studentSubjects = await \_context.StudentSubjects

.Where(ss => ss.StudentId == student.Id && ss.IsActive)

.Select(ss => ss.SubjectId)

.ToListAsync();

// Assign relevant timetable slots to student

var relevantSlots = timetableSlots

.Where(slot => studentSubjects.Contains(slot.SubjectId))

.ToList();

foreach (var slot in relevantSlots)

{

studentTimetables.Add(new StudentTimetable

{

StudentId = student.Id,

TimetableSlotId = slot.Id,

IsActive = true

});

}

}

\_context.StudentTimetables.AddRange(studentTimetables);

await \_context.SaveChangesAsync();

}

private async Task<Timetable> GetTimetableWithDetailsAsync(Guid timetableId)

{

return await \_context.Timetables

.Include(t => t.School)

.Include(t => t.Grade)

.Include(t => t.SchoolYear)

.Include(t => t.Term)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Subject)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Teacher)

.FirstOrDefaultAsync(t => t.Id == timetableId)

?? throw new NotFoundException($"Timetable {timetableId} not found");

}

}

}// Services/Implementations/GradingService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class GradingService : IGradingService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

public GradingService(SchoolManagementDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

public async Task<IEnumerable<GradingSchemeDto>> GetAllGradingSchemesAsync()

{

var schemes = await \_context.GradingSchemes

.Include(gs => gs.GradingScales.OrderBy(scale => scale.SortOrder))

.Where(gs => gs.IsActive)

.ToListAsync();

return \_mapper.Map<IEnumerable<GradingSchemeDto>>(schemes);

}

public async Task<GradingSchemeDto> CreateGradingSchemeAsync(CreateGradingSchemeDto createDto)

{

var scheme = \_mapper.Map<GradingScheme>(createDto);

\_context.GradingSchemes.Add(scheme);

await \_context.SaveChangesAsync();

// Add grading scales

if (createDto.GradingScales.Any())

{

var scales = createDto.GradingScales.Select(scaleDto =>

{

var scale = \_mapper.Map<GradingScale>(scaleDto);

scale.GradingSchemeId = scheme.Id;

return scale;

}).ToList();

\_context.GradingScales.AddRange(scales);

await \_context.SaveChangesAsync();

}

return await GetGradingSchemeByIdAsync(scheme.Id);

}

public async Task<GradingSchemeDto> UpdateGradingSchemeAsync(Guid schemeId, UpdateGradingSchemeDto updateDto)

{

var scheme = await \_context.GradingSchemes.FindAsync(schemeId);

if (scheme == null)

throw new NotFoundException($"Grading scheme {schemeId} not found");

\_mapper.Map(updateDto, scheme);

await \_context.SaveChangesAsync();

return await GetGradingSchemeByIdAsync(schemeId);

}

public async Task<bool> DeleteGradingSchemeAsync(Guid schemeId)

{

var scheme = await \_context.GradingSchemes.FindAsync(schemeId);

if (scheme == null)

return false;

// Check if scheme is being used by any schools

var isInUse = await \_context.Schools.AnyAsync(s => s.GradingSchemeId == schemeId);

if (isInUse)

{

throw new ConflictException("Cannot delete grading scheme that is in use by schools");

}

scheme.IsActive = false;

await \_context.SaveChangesAsync();

return true;

}

public async Task<GradingScaleDto> AddGradingScaleAsync(Guid schemeId, CreateGradingScaleDto createDto)

{

var scheme = await \_context.GradingSchemes.FindAsync(schemeId);

if (scheme == null)

throw new NotFoundException($"Grading scheme {schemeId} not found");

var scale = \_mapper.Map<GradingScale>(createDto);

scale.GradingSchemeId = schemeId;

\_context.GradingScales.Add(scale);

await \_context.SaveChangesAsync();

return \_mapper.Map<GradingScaleDto>(scale);

}

public async Task<GradingScaleDto> UpdateGradingScaleAsync(Guid scaleId, UpdateGradingScaleDto updateDto)

{

var scale = await \_context.GradingScales.FindAsync(scaleId);

if (scale == null)

throw new NotFoundException($"Grading scale {scaleId} not found");

\_mapper.Map(updateDto, scale);

await \_context.SaveChangesAsync();

return \_mapper.Map<GradingScaleDto>(scale);

}

public async Task<bool> DeleteGradingScaleAsync(Guid scaleId)

{

var scale = await \_context.GradingScales.FindAsync(scaleId);

if (scale == null)

return false;

\_context.GradingScales.Remove(scale);

await \_context.SaveChangesAsync();

return true;

}

public async Task<GradeCalculationDto> CalculateGradeAsync(Guid gradingSchemeId, decimal percentage)

{

var gradingScale = await \_context.GradingScales

.Where(gs => gs.GradingSchemeId == gradingSchemeId &&

percentage >= gs.MinPercentage &&

percentage <= gs.MaxPercentage)

.FirstOrDefaultAsync();

if (gradingScale == null)

{

// Return default grade if no matching scale found

return new GradeCalculationDto

{

Symbol = "F",

Unit = 0,

Description = "Fail",

Percentage = percentage

};

}

return new GradeCalculationDto

{

Symbol = gradingScale.Symbol,

Unit = gradingScale.Unit,

Description = gradingScale.Description,

Percentage = percentage

};

}

public async Task<IEnumerable<GradingScaleDto>> GetGradingScalesAsync(Guid schemeId)

{

var scales = await \_context.GradingScales

.Where(gs => gs.GradingSchemeId == schemeId)

.OrderBy(gs => gs.SortOrder)

.ToListAsync();

return \_mapper.Map<IEnumerable<GradingScaleDto>>(scales);

}

private async Task<GradingSchemeDto> GetGradingSchemeByIdAsync(Guid schemeId)

{

var scheme = await \_context.GradingSchemes

.Include(gs => gs.GradingScales.OrderBy(scale => scale.SortOrder))

.FirstOrDefaultAsync(gs => gs.Id == schemeId);

if (scheme == null)

throw new NotFoundException($"Grading scheme {schemeId} not found");

return \_mapper.Map<GradingSchemeDto>(scheme);

}

}

}

// Services/Implementations/AttendanceService.cs

namespace SchoolManagement.Services.Implementations

{

public class AttendanceService : IAttendanceService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly INotificationService \_notificationService;

public AttendanceService(

SchoolManagementDbContext context,

IMapper mapper,

INotificationService notificationService)

{

\_context = context;

\_mapper = mapper;

\_notificationService = notificationService;

}

public async Task<AttendanceDto> MarkAttendanceAsync(CreateAttendanceDto createAttendanceDto)

{

// Check if attendance already exists for this date

var existingAttendance = await \_context.Attendances

.FirstOrDefaultAsync(a => a.StudentId == createAttendanceDto.StudentId &&

a.SubjectId == createAttendanceDto.SubjectId &&

a.Date.Date == createAttendanceDto.Date.Date);

if (existingAttendance != null)

{

// Update existing attendance

existingAttendance.Status = createAttendanceDto.Status;

existingAttendance.Remarks = createAttendanceDto.Remarks;

await \_context.SaveChangesAsync();

return \_mapper.Map<AttendanceDto>(existingAttendance);

}

// Create new attendance record

var attendance = \_mapper.Map<Attendance>(createAttendanceDto);

\_context.Attendances.Add(attendance);

await \_context.SaveChangesAsync();

// Send notification to parents if absent

if (attendance.Status == AttendanceStatus.Absent)

{

await NotifyParentsAboutAbsenceAsync(attendance);

}

return \_mapper.Map<AttendanceDto>(attendance);

}

public async Task<IEnumerable<AttendanceDto>> GetAttendanceByClassAsync(Guid gradeId, DateTime date)

{

var attendances = await \_context.Attendances

.Where(a => a.Student.CurrentGradeId == gradeId && a.Date.Date == date.Date)

.Include(a => a.Student)

.Include(a => a.Subject)

.Include(a => a.Teacher)

.Include(a => a.Term)

.ToListAsync();

return \_mapper.Map<IEnumerable<AttendanceDto>>(attendances);

}

public async Task<IEnumerable<AttendanceDto>> GetStudentAttendanceAsync(Guid studentId, Guid termId)

{

var attendances = await \_context.Attendances

.Where(a => a.StudentId == studentId && a.TermId == termId)

.Include(a => a.Student)

.Include(a => a.Subject)

.Include(a => a.Teacher)

.Include(a => a.Term)

.OrderBy(a => a.Date)

.ToListAsync();

return \_mapper.Map<IEnumerable<AttendanceDto>>(attendances);

}

public async Task<AttendanceReportDto> GetAttendanceReportAsync(Guid studentId, Guid termId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.School)

.FirstOrDefaultAsync(s => s.Id == studentId);

if (student == null)

throw new NotFoundException($"Student {studentId} not found");

var term = await \_context.Terms.FindAsync(termId);

if (term == null)

throw new NotFoundException($"Term {termId} not found");

var attendances = await GetStudentAttendanceAsync(studentId, termId);

var subjectAttendance = attendances

.GroupBy(a => new { a.Subject.Id, a.Subject.Name })

.Select(g => new AttendanceSummaryDto

{

SubjectId = g.Key.Id,

SubjectName = g.Key.Name,

TotalClasses = g.Count(),

PresentClasses = g.Count(a => a.Status == AttendanceStatus.Present),

AbsentClasses = g.Count(a => a.Status == AttendanceStatus.Absent),

LateClasses = g.Count(a => a.Status == AttendanceStatus.Late),

AttendancePercentage = g.Count() > 0 ?

(decimal)g.Count(a => a.Status == AttendanceStatus.Present) / g.Count() \* 100 : 0

})

.ToList();

var totalClasses = subjectAttendance.Sum(s => s.TotalClasses);

var totalPresent = subjectAttendance.Sum(s => s.PresentClasses);

return new AttendanceReportDto

{

Student = \_mapper.Map<StudentDto>(student),

Term = \_mapper.Map<TermDto>(term),

SubjectAttendance = subjectAttendance,

TotalClassesTaken = totalClasses,

TotalClassesAttended = totalPresent,

OverallAttendancePercentage = totalClasses > 0 ? (decimal)totalPresent / totalClasses \* 100 : 0

};

}

public async Task<bool> BulkMarkAttendanceAsync(List<CreateAttendanceDto> attendanceList)

{

var attendances = new List<Attendance>();

var notifications = new List<CreateNotificationDto>();

foreach (var createDto in attendanceList)

{

// Check for existing attendance

var existing = await \_context.Attendances

.FirstOrDefaultAsync(a => a.StudentId == createDto.StudentId &&

a.SubjectId == createDto.SubjectId &&

a.Date.Date == createDto.Date.Date);

if (existing != null)

{

existing.Status = createDto.Status;

existing.Remarks = createDto.Remarks;

}

else

{

var attendance = \_mapper.Map<Attendance>(createDto);

attendances.Add(attendance);

// Queue absence notifications

if (attendance.Status == AttendanceStatus.Absent)

{

var student = await \_context.Students

.Include(s => s.Parents)

.FirstOrDefaultAsync(s => s.Id == createDto.StudentId);

if (student?.Parents.Any() == true)

{

foreach (var parent in student.Parents)

{

notifications.Add(new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = student.Id,

Title = "Student Absence",

Message = $"{student.FirstName} {student.LastName} was absent on {createDto.Date:dd/MM/yyyy}",

Type = NotificationType.Attendance,

Channel = NotificationChannel.WhatsApp

});

}

}

}

}

}

if (attendances.Any())

\_context.Attendances.AddRange(attendances);

await \_context.SaveChangesAsync();

// Send notifications

foreach (var notification in notifications)

{

try

{

await \_notificationService.CreateNotificationAsync(notification);

}

catch (Exception ex)

{

// Log but don't fail the attendance marking

// \_logger.LogError(ex, "Failed to send attendance notification");

}

}

return true;

}

public async Task<AttendanceDto> UpdateAttendanceAsync(Guid attendanceId, UpdateAttendanceDto updateDto)

{

var attendance = await \_context.Attendances.FindAsync(attendanceId);

if (attendance == null)

throw new NotFoundException($"Attendance record {attendanceId} not found");

\_mapper.Map(updateDto, attendance);

await \_context.SaveChangesAsync();

return \_mapper.Map<AttendanceDto>(attendance);

}

private async Task NotifyParentsAboutAbsenceAsync(Attendance attendance)

{

var student = await \_context.Students

.Include(s => s.Parents)

.FirstOrDefaultAsync(s => s.Id == attendance.StudentId);

if (student?.Parents.Any() == true)

{

foreach (var parent in student.Parents)

{

var notification = new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = student.Id,

Title = "Student Absence Alert",

Message = $"Dear {parent.FirstName}, {student.FirstName} {student.LastName} was absent from school on {attendance.Date:dd/MM/yyyy}. Please contact the school if this absence was not planned.",

Type = NotificationType.Attendance,

Channel = NotificationChannel.WhatsApp

};

await \_notificationService.CreateNotificationAsync(notification);

}

}

}

}

}

// Services/Implementations/NotificationService.cs

namespace SchoolManagement.Services.Implementations

{

public class NotificationService : INotificationService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly ILogger<NotificationService> \_logger;

private readonly IWhatsAppService \_whatsAppService;

private readonly IEmailService \_emailService;

public NotificationService(

SchoolManagementDbContext context,

IMapper mapper,

ILogger<NotificationService> logger,

IWhatsAppService whatsAppService,

IEmailService emailService)

{

\_context = context;

\_mapper = mapper;

\_logger = logger;

\_whatsAppService = whatsAppService;

\_emailService = emailService;

}

public async Task<NotificationDto> CreateNotificationAsync(CreateNotificationDto createDto)

{

var notification = \_mapper.Map<Notification>(createDto);

\_context.Notifications.Add(notification);

await \_context.SaveChangesAsync();

// Auto-send the notification

await SendNotificationAsync(notification.Id);

return \_mapper.Map<NotificationDto>(notification);

}

public async Task<bool> SendNotificationAsync(Guid notificationId)

{

var notification = await \_context.Notifications

.Include(n => n.Parent)

.Include(n => n.Student)

.Include(n => n.Teacher)

.FirstOrDefaultAsync(n => n.Id == notificationId);

if (notification == null || notification.IsSent)

return false;

try

{

bool sent = false;

switch (notification.Channel)

{

case NotificationChannel.WhatsApp:

sent = await SendWhatsAppNotificationAsync(notification);

break;

case NotificationChannel.Email:

sent = await SendEmailNotificationAsync(notification);

break;

case NotificationChannel.SMS:

sent = await SendSMSNotificationAsync(notification);

break;

case NotificationChannel.InApp:

sent = true; // In-app notifications are always "sent"

break;

}

if (sent)

{

notification.IsSent = true;

notification.SentAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

}

return sent;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send notification {NotificationId}", notificationId);

return false;

}

}

public async Task<bool> SendBulkNotificationAsync(List<Guid> recipientIds, CreateNotificationDto notificationDto)

{

var notifications = new List<Notification>();

foreach (var recipientId in recipientIds)

{

var notification = \_mapper.Map<Notification>(notificationDto);

// Determine recipient type and set appropriate ID

if (notificationDto.ParentId.HasValue)

notification.ParentId = recipientId;

else if (notificationDto.StudentId.HasValue)

notification.StudentId = recipientId;

else if (notificationDto.TeacherId.HasValue)

notification.TeacherId = recipientId;

notifications.Add(notification);

}

\_context.Notifications.AddRange(notifications);

await \_context.SaveChangesAsync();

// Send all notifications

var sendTasks = notifications.Select(n => SendNotificationAsync(n.Id));

var results = await Task.WhenAll(sendTasks);

return results.All(r => r);

}

public async Task<IEnumerable<NotificationDto>> GetUserNotificationsAsync(Guid userId, string userType)

{

IQueryable<Notification> query = \_context.Notifications;

switch (userType.ToLower())

{

case "parent":

query = query.Where(n => n.ParentId == userId);

break;

case "student":

query = query.Where(n => n.StudentId == userId);

break;

case "teacher":

query = query.Where(n => n.TeacherId == userId);

break;

default:

throw new ArgumentException("Invalid user type");

}

var notifications = await query

.Include(n => n.Parent)

.Include(n => n.Student)

.Include(n => n.Teacher)

.OrderByDescending(n => n.CreatedAt)

.Take(50) // Limit for performance

.ToListAsync();

return \_mapper.Map<IEnumerable<NotificationDto>>(notifications);

}

public async Task<bool> MarkAsReadAsync(Guid notificationId)

{

var notification = await \_context.Notifications.FindAsync(notificationId);

if (notification == null)

return false;

if (!notification.IsRead)

{

notification.IsRead = true;

notification.ReadAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

}

return true;

}

public async Task<bool> SendWhatsAppMessageAsync(string phoneNumber, string message)

{

try

{

return await \_whatsAppService.SendMessageAsync(phoneNumber, message);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send WhatsApp message to {PhoneNumber}", phoneNumber);

return false;

}

}

public async Task<bool> SendEmailNotificationAsync(string email, string subject, string message)

{

try

{

return await \_emailService.SendEmailAsync(email, subject, message);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send email to {Email}", email);

return false;

}

}

public async Task<NotificationStatsDto> GetNotificationStatsAsync(Guid schoolId)

{

var notifications = await \_context.Notifications

.Where(n => (n.Student != null && n.Student.SchoolId == schoolId) ||

(n.Teacher != null && n.Teacher.SchoolId == schoolId))

.ToListAsync();

return new NotificationStatsDto

{

TotalNotifications = notifications.Count,

SentNotifications = notifications.Count(n => n.IsSent),

PendingNotifications = notifications.Count(n => !n.IsSent),

ReadNotifications = notifications.Count(n => n.IsRead),

NotificationsByType = notifications.GroupBy(n => n.Type)

.ToDictionary(g => g.Key, g => g.Count()),

NotificationsByChannel = notifications.GroupBy(n => n.Channel)

.ToDictionary(g => g.Key, g => g.Count())

};

}

private async Task<bool> SendWhatsAppNotificationAsync(Notification notification)

{

string phoneNumber = null;

if (notification.Parent != null && !string.IsNullOrEmpty(notification.Parent.WhatsAppNumber))

phoneNumber = notification.Parent.WhatsAppNumber;

else if (notification.Teacher != null && !string.IsNullOrEmpty(notification.Teacher.PhoneNumber))

phoneNumber = notification.Teacher.PhoneNumber;

if (string.IsNullOrEmpty(phoneNumber))

return false;

var success = await \_whatsAppService.SendMessageAsync(phoneNumber, notification.Message);

if (success)

{

notification.ExternalId = Guid.NewGuid().ToString(); // WhatsApp message ID would go here

}

return success;

}

private async Task<bool> SendEmailNotificationAsync(Notification notification)

{

string email = null;

if (notification.Parent != null && !string.IsNullOrEmpty(notification.Parent.Email))

email = notification.Parent.Email;

else if (notification.Teacher != null && !string.IsNullOrEmpty(notification.Teacher.Email))

email = notification.Teacher.Email;

if (string.IsNullOrEmpty(email))

return false;

return await \_emailService.SendEmailAsync(email, notification.Title, notification.Message);

}

private async Task<bool> SendSMSNotificationAsync(Notification notification)

{

string phoneNumber = null;

if (notification.Parent != null && !string.IsNullOrEmpty(notification.Parent.PhoneNumber))

phoneNumber = notification.Parent.PhoneNumber;

else if (notification.Teacher != null && !string.IsNullOrEmpty(notification.Teacher.PhoneNumber))

phoneNumber = notification.Teacher.PhoneNumber;

if (string.IsNullOrEmpty(phoneNumber))

return false;

// SMS service implementation would go here

await Task.Delay(100); // Placeholder

return true;

}

}

}

// Services/Interfaces/Supporting/IWhatsAppService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IWhatsAppService

{

Task<bool> SendMessageAsync(string phoneNumber, string message);

Task<bool> SendMediaMessageAsync(string phoneNumber, string mediaUrl, string caption);

Task<bool> SendTemplateMessageAsync(string phoneNumber, string templateName, Dictionary<string, string> parameters);

Task<List<WhatsAppMessageStatus>> GetMessageStatusAsync(List<string> messageIds);

}

public class WhatsAppMessageStatus

{

public string MessageId { get; set; } = string.Empty;

public string Status { get; set; } = string.Empty; // sent, delivered, read, failed

public DateTime Timestamp { get; set; }

}

}

// Services/Interfaces/Supporting/IEmailService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IEmailService

{

Task<bool> SendEmailAsync(string toEmail, string subject, string body);

Task<bool> SendEmailAsync(List<string> toEmails, string subject, string body);

Task<bool> SendEmailWithAttachmentAsync(string toEmail, string subject, string body, byte[] attachment, string attachmentName);

Task<bool> SendTemplateEmailAsync(string toEmail, string templateName, Dictionary<string, object> templateData);

}

}

// Services/Implementations/Supporting/WhatsAppService.cs (Basic Implementation)

namespace SchoolManagement.Services.Implementations

{

public class WhatsAppService : IWhatsAppService

{

private readonly HttpClient \_httpClient;

private readonly IConfiguration \_configuration;

private readonly ILogger<WhatsAppService> \_logger;

public WhatsAppService(

HttpClient httpClient,

IConfiguration configuration,

ILogger<WhatsAppService> logger)

{

\_httpClient = httpClient;

\_configuration = configuration;

\_logger = logger;

}

public async Task<bool> SendMessageAsync(string phoneNumber, string message)

{

try

{

// WhatsApp Business API implementation

var apiUrl = \_configuration["WhatsApp:ApiUrl"];

var accessToken = \_configuration["WhatsApp:AccessToken"];

var phoneNumberId = \_configuration["WhatsApp:PhoneNumberId"];

var payload = new

{

messaging\_product = "whatsapp",

to = phoneNumber,

type = "text",

text = new { body = message }

};

\_httpClient.DefaultRequestHeaders.Authorization =

new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", accessToken);

var json = System.Text.Json.JsonSerializer.Serialize(payload);

var content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

var response = await \_httpClient.PostAsync($"{apiUrl}/{phoneNumberId}/messages", content);

if (response.IsSuccessStatusCode)

{

\_logger.LogInformation("WhatsApp message sent successfully to {PhoneNumber}", phoneNumber);

return true;

}

else

{

var error = await response.Content.ReadAsStringAsync();

\_logger.LogError("Failed to send WhatsApp message to {PhoneNumber}. Error: {Error}", phoneNumber, error);

return false;

}

}

catch (Exception ex)

{

\_logger.LogError(ex, "Exception while sending WhatsApp message to {PhoneNumber}", phoneNumber);

return false;

}

}

public async Task<bool> SendMediaMessageAsync(string phoneNumber, string mediaUrl, string caption)

{

// Implementation for media messages

await Task.Delay(100); // Placeholder

return true;

}

public async Task<bool> SendTemplateMessageAsync(string phoneNumber, string templateName, Dictionary<string, string> parameters)

{

// Implementation for template messages

await Task.Delay(100); // Placeholder

return true;

}

public async Task<List<WhatsAppMessageStatus>> GetMessageStatusAsync(List<string> messageIds)

{

// Implementation for checking message status

await Task.Delay(100); // Placeholder

return new List<WhatsAppMessageStatus>();

}

}

}

// Services/Implementations/Supporting/EmailService.cs (Basic Implementation)

namespace SchoolManagement.Services.Implementations

{

public class EmailService : IEmailService

{

private readonly IConfiguration \_configuration;

private readonly ILogger<EmailService> \_logger;

public EmailService(IConfiguration configuration, ILogger<EmailService> logger)

{

\_configuration = configuration;

\_logger = logger;

}

public async Task<bool> SendEmailAsync(string toEmail, string subject, string body)

{

try

{

// SMTP configuration

var smtpHost = \_configuration["Email:SmtpHost"];

var smtpPort = int.Parse(\_configuration["Email:SmtpPort"] ?? "587");

var username = \_configuration["Email:Username"];

var password = \_configuration["Email:Password"];

var fromEmail = \_configuration["Email:FromEmail"];

using var client = new System.Net.Mail.SmtpClient(smtpHost, smtpPort)

{

Credentials = new System.Net.NetworkCredential(username, password),

EnableSsl = true

};

var message = new System.Net.Mail.MailMessage(fromEmail, toEmail, subject, body)

{

IsBodyHtml = true

};

await client.SendMailAsync(message);

\_logger.LogInformation("Email sent successfully to {ToEmail}", toEmail);

return true;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send email to {ToEmail}", toEmail);

return false;

}

}

public async Task<bool> SendEmailAsync(List<string> toEmails, string subject, string body)

{

var tasks = toEmails.Select(email => SendEmailAsync(email, subject, body));

var results = await Task.WhenAll(tasks);

return results.All(r => r);

}

public async Task<bool> SendEmailWithAttachmentAsync(string toEmail, string subject, string body, byte[] attachment, string attachmentName)

{

// Implementation with attachment support

await Task.Delay(100); // Placeholder

return true;

}

public async Task<bool> SendTemplateEmailAsync(string toEmail, string templateName, Dictionary<string, object> templateData)

{

// Template email implementation

await Task.Delay(100); // Placeholder

return true;

}

}

}

// Program.cs - Dependency Injection Setup

namespace SchoolManagement.API

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

// Add services to the container

builder.Services.AddDbContext<SchoolManagementDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// AutoMapper

builder.Services.AddAutoMapper(typeof(AutoMapperProfile));

// Register services

builder.Services.AddScoped<ISchoolService, SchoolService>();

builder.Services.AddScoped<IStudentService, StudentService>();

builder.Services.AddScoped<ITeacherService, TeacherService>();

builder.Services.AddScoped<IResultService, ResultService>();

builder.Services.AddScoped<IAttendanceService, AttendanceService>();

builder.Services.AddScoped<IGradingService, GradingService>();

builder.Services.AddScoped<ITimetableService, TimetableService>();

builder.Services.AddScoped<INotificationService, NotificationService>();

builder.Services.AddScoped<IHandwritingRecognitionService, HandwritingRecognitionService>();

// Supporting services

builder.Services.AddScoped<IImageProcessingService, ImageProcessingService>();

builder.Services.AddScoped<IMachineLearningService, MachineLearningService>();

builder.Services.AddScoped<IFileStorageService, FileStorageService>();

builder.Services.AddScoped<IPdfGenerationService, PdfGenerationService>();

builder.Services.AddScoped<IWhatsAppService, WhatsAppService>();

builder.Services.AddScoped<IEmailService, EmailService>();

// HTTP Client for WhatsApp

builder.Services.AddHttpClient<IWhatsAppService, WhatsAppService>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

}

}

}// Models/Entities/Exam/Exam.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.Entities

{

public class Exam : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public Guid GradeId { get; set; }

public virtual Grade Grade { get; set; } = null!;

public Guid TermId { get; set; }

public virtual Term Term { get; set; } = null!;

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

public Guid CreatedByTeacherId { get; set; }

public virtual Teacher CreatedByTeacher { get; set; } = null!;

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

public ExamType Type { get; set; }

public DateTime ExamDate { get; set; }

public TimeSpan Duration { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public decimal TotalMarks { get; set; }

public decimal PassingMarks { get; set; }

[StringLength(100)]

public string Venue { get; set; } = string.Empty;

public ExamStatus Status { get; set; } = ExamStatus.Draft;

public bool IsPublished { get; set; } = false;

public DateTime? PublishedAt { get; set; }

public bool AllowHandwritingRecognition { get; set; } = false;

public bool IsOnline { get; set; } = false;

public bool RequiresSupervision { get; set; } = true;

[StringLength(500)]

public string Instructions { get; set; } = string.Empty;

public string MaterialsRequired { get; set; } = string.Empty; // JSON list

// Navigation Properties

public virtual ICollection<ExamQuestion> Questions { get; set; } = new List<ExamQuestion>();

public virtual ICollection<ExamRegistration> Registrations { get; set; } = new List<ExamRegistration>();

public virtual ICollection<ExamSession> Sessions { get; set; } = new List<ExamSession>();

public virtual ICollection<ExamResult> Results { get; set; } = new List<ExamResult>();

public virtual ICollection<ExamSupervisor> Supervisors { get; set; } = new List<ExamSupervisor>();

}

public enum ExamType

{

Quiz,

Test,

MidTerm,

FinalExam,

Assignment,

Project,

Practical,

Oral,

Continuous

}

public enum ExamStatus

{

Draft,

Published,

InProgress,

Completed,

Cancelled,

Postponed

}

}

// Models/Entities/Exam/ExamQuestion.cs

namespace SchoolManagement.Core.Entities

{

public class ExamQuestion : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public int QuestionNumber { get; set; }

[Required]

public string QuestionText { get; set; } = string.Empty;

public QuestionType Type { get; set; }

public decimal Marks { get; set; }

public bool IsRequired { get; set; } = true;

public TimeSpan? TimeLimit { get; set; }

// For multiple choice questions

public string Options { get; set; } = string.Empty; // JSON array

public string CorrectAnswer { get; set; } = string.Empty;

// For essay/written questions

public int? WordLimit { get; set; }

public string SampleAnswer { get; set; } = string.Empty;

public string MarkingRubric { get; set; } = string.Empty; // JSON

// For handwriting recognition

public bool EnableHandwritingRecognition { get; set; } = false;

public string ExpectedKeywords { get; set; } = string.Empty; // JSON array

// Media attachments

public string AttachmentPath { get; set; } = string.Empty;

public int SortOrder { get; set; }

// Navigation Properties

public virtual ICollection<ExamAnswer> Answers { get; set; } = new List<ExamAnswer>();

}

public enum QuestionType

{

MultipleChoice,

TrueFalse,

ShortAnswer,

Essay,

FillInTheBlank,

Matching,

Numerical,

Drawing,

FileUpload

}

}

// Models/Entities/Exam/ExamRegistration.cs

namespace SchoolManagement.Core.Entities

{

public class ExamRegistration : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public DateTime RegistrationDate { get; set; } = DateTime.UtcNow;

public RegistrationStatus Status { get; set; } = RegistrationStatus.Registered;

public bool IsPresent { get; set; } = false;

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

[StringLength(100)]

public string SeatNumber { get; set; } = string.Empty;

public bool RequiresSpecialAccommodation { get; set; } = false;

public string SpecialAccommodations { get; set; } = string.Empty;

[StringLength(500)]

public string Remarks { get; set; } = string.Empty;

}

public enum RegistrationStatus

{

Registered,

Present,

Absent,

Late,

Disqualified,

Withdrawn

}

}

// Models/Entities/Exam/ExamSession.cs

namespace SchoolManagement.Core.Entities

{

public class ExamSession : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public DateTime StartTime { get; set; }

public DateTime? EndTime { get; set; }

public TimeSpan ActualDuration { get; set; }

public SessionStatus Status { get; set; } = SessionStatus.NotStarted;

public bool IsSubmitted { get; set; } = false;

public DateTime? SubmittedAt { get; set; }

public bool IsAutoSubmitted { get; set; } = false;

public int TotalQuestions { get; set; }

public int AnsweredQuestions { get; set; }

// Proctoring data

public string DeviceInfo { get; set; } = string.Empty; // JSON

public string IPAddress { get; set; } = string.Empty;

public string BrowserInfo { get; set; } = string.Empty;

// Security flags

public bool HasViolations { get; set; } = false;

public string ViolationDetails { get; set; } = string.Empty; // JSON

public decimal? Score { get; set; }

public decimal? Percentage { get; set; }

// Navigation Properties

public virtual ICollection<ExamAnswer> Answers { get; set; } = new List<ExamAnswer>();

public virtual ICollection<ExamSessionLog> SessionLogs { get; set; } = new List<ExamSessionLog>();

}

public enum SessionStatus

{

NotStarted,

InProgress,

Paused,

Completed,

TimedOut,

Interrupted,

Disqualified

}

}

// Models/Entities/Exam/ExamAnswer.cs

namespace SchoolManagement.Core.Entities

{

public class ExamAnswer : BaseEntity

{

public Guid ExamSessionId { get; set; }

public virtual ExamSession ExamSession { get; set; } = null!;

public Guid ExamQuestionId { get; set; }

public virtual ExamQuestion ExamQuestion { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public string AnswerText { get; set; } = string.Empty;

// For file uploads or handwritten answers

public string AttachmentPath { get; set; } = string.Empty;

public string HandwritingImagePath { get; set; } = string.Empty;

// AI Recognition results

public string RecognizedText { get; set; } = string.Empty;

public decimal AiConfidenceScore { get; set; }

public bool IsAiProcessed { get; set; } = false;

// Marking

public decimal? MarksObtained { get; set; }

public bool IsMarked { get; set; } = false;

public Guid? MarkedByTeacherId { get; set; }

public virtual Teacher? MarkedByTeacher { get; set; }

public DateTime? MarkedAt { get; set; }

public string TeacherFeedback { get; set; } = string.Empty;

// Answer metadata

public DateTime AnsweredAt { get; set; } = DateTime.UtcNow;

public TimeSpan TimeSpent { get; set; }

public int AttemptNumber { get; set; } = 1;

public bool IsFlagged { get; set; } = false;

public string FlagReason { get; set; } = string.Empty;

}

}

// Models/Entities/Exam/ExamResult.cs

namespace SchoolManagement.Core.Entities

{

public class ExamResult : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid ExamSessionId { get; set; }

public virtual ExamSession ExamSession { get; set; } = null!;

public decimal TotalMarks { get; set; }

public decimal MarksObtained { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public int GradeUnit { get; set; }

public ResultStatus Status { get; set; } = ResultStatus.Pending;

public bool IsPassed { get; set; }

public string Remarks { get; set; } = string.Empty;

// Marking details

public bool IsFullyMarked { get; set; } = false;

public int QuestionsMarked { get; set; }

public int TotalQuestions { get; set; }

public Guid? MarkedByTeacherId { get; set; }

public virtual Teacher? MarkedByTeacher { get; set; }

public DateTime? MarkedAt { get; set; }

// Result publication

public bool IsPublished { get; set; } = false;

public DateTime? PublishedAt { get; set; }

// Analytics

public int Rank { get; set; }

public decimal? ClassAverage { get; set; }

public decimal? HighestScore { get; set; }

public decimal? LowestScore { get; set; }

public string DetailedAnalysis { get; set; } = string.Empty; // JSON

}

public enum ResultStatus

{

Pending,

InProgress,

Completed,

Published,

UnderReview,

Disputed

}

}

// Models/Entities/Exam/ExamSupervisor.cs

namespace SchoolManagement.Core.Entities

{

public class ExamSupervisor : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid TeacherId { get; set; }

public virtual Teacher Teacher { get; set; } = null!;

public SupervisorRole Role { get; set; }

public DateTime AssignedAt { get; set; } = DateTime.UtcNow;

public bool IsPresent { get; set; } = false;

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

public string Responsibilities { get; set; } = string.Empty;

public string Notes { get; set; } = string.Empty;

// Navigation Properties

public virtual ICollection<ExamIncident> ReportedIncidents { get; set; } = new List<ExamIncident>();

}

public enum SupervisorRole

{

ChiefSupervisor,

AssistantSupervisor,

RoomSupervisor,

FloatingSupervisor,

TechnicalSupport

}

}

// Models/Entities/Exam/ExamIncident.cs

namespace SchoolManagement.Core.Entities

{

public class ExamIncident : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid? StudentId { get; set; }

public virtual Student? Student { get; set; }

public Guid ReportedByTeacherId { get; set; }

public virtual Teacher ReportedByTeacher { get; set; } = null!;

public IncidentType Type { get; set; }

public IncidentSeverity Severity { get; set; }

[Required]

public string Description { get; set; } = string.Empty;

public DateTime IncidentTime { get; set; }

public string Location { get; set; } = string.Empty;

public string ActionTaken { get; set; } = string.Empty;

public bool RequiresFollowUp { get; set; } = false;

public string FollowUpNotes { get; set; } = string.Empty;

public string AttachmentsPath { get; set; } = string.Empty; // Evidence photos/videos

public IncidentStatus Status { get; set; } = IncidentStatus.Open;

public DateTime? ResolvedAt { get; set; }

public Guid? ResolvedByTeacherId { get; set; }

public virtual Teacher? ResolvedByTeacher { get; set; }

}

public enum IncidentType

{

Cheating,

MobilePhone,

UnauthorizedMaterial,

Disruption,

LateComing,

EarlyLeaving,

TechnicalIssue,

MedicalEmergency,

Misconduct,

Other

}

public enum IncidentSeverity

{

Low,

Medium,

High,

Critical

}

public enum IncidentStatus

{

Open,

UnderInvestigation,

Resolved,

Closed,

Escalated

}

}

// Models/Entities/Exam/ExamSessionLog.cs

namespace SchoolManagement.Core.Entities

{

public class ExamSessionLog : BaseEntity

{

public Guid ExamSessionId { get; set; }

public virtual ExamSession ExamSession { get; set; } = null!;

public LogType Type { get; set; }

public string Event { get; set; } = string.Empty;

public string Details { get; set; } = string.Empty; // JSON

public DateTime Timestamp { get; set; } = DateTime.UtcNow;

public string IPAddress { get; set; } = string.Empty;

public string UserAgent { get; set; } = string.Empty;

public bool IsSuspicious { get; set; } = false;

public string SuspicionReason { get; set; } = string.Empty;

}

public enum LogType

{

SessionStart,

SessionEnd,

QuestionViewed,

AnswerSaved,

AnswerChanged,

PageRefresh,

WindowFocusLost,

WindowFocusGained,

CopyDetected,

PasteDetected,

RightClickDisabled,

DevToolsDetected,

NetworkChange,

SuspiciousActivity,

SystemError

}

}

// DTOs/ExamDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ExamDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public ExamType Type { get; set; }

public DateTime ExamDate { get; set; }

public TimeSpan Duration { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public decimal TotalMarks { get; set; }

public decimal PassingMarks { get; set; }

public string Venue { get; set; } = string.Empty;

public ExamStatus Status { get; set; }

public bool IsPublished { get; set; }

public bool AllowHandwritingRecognition { get; set; }

public bool IsOnline { get; set; }

public string Instructions { get; set; } = string.Empty;

public SubjectDto Subject { get; set; } = null!;

public GradeDto Grade { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public TeacherDto CreatedByTeacher { get; set; } = null!;

public List<ExamQuestionDto> Questions { get; set; } = new();

public List<ExamSupervisorDto> Supervisors { get; set; } = new();

public int RegisteredStudents { get; set; }

public int CompletedStudents { get; set; }

public DateTime CreatedAt { get; set; }

}

public class CreateExamDto

{

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

[Required]

public ExamType Type { get; set; }

[Required]

public DateTime ExamDate { get; set; }

[Required]

public TimeSpan Duration { get; set; }

[Required]

public TimeSpan StartTime { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal TotalMarks { get; set; }

[Required]

[Range(0, double.MaxValue)]

public decimal PassingMarks { get; set; }

[StringLength(100)]

public string Venue { get; set; } = string.Empty;

public bool AllowHandwritingRecognition { get; set; } = false;

public bool IsOnline { get; set; } = false;

public bool RequiresSupervision { get; set; } = true;

[StringLength(500)]

public string Instructions { get; set; } = string.Empty;

public List<string> MaterialsRequired { get; set; } = new();

public List<CreateExamQuestionDto> Questions { get; set; } = new();

public List<Guid> SupervisorIds { get; set; } = new();

}

public class ExamQuestionDto

{

public Guid Id { get; set; }

public int QuestionNumber { get; set; }

public string QuestionText { get; set; } = string.Empty;

public QuestionType Type { get; set; }

public decimal Marks { get; set; }

public bool IsRequired { get; set; }

public TimeSpan? TimeLimit { get; set; }

public List<string> Options { get; set; } = new();

public string CorrectAnswer { get; set; } = string.Empty;

public int? WordLimit { get; set; }

public bool EnableHandwritingRecognition { get; set; }

public List<string> ExpectedKeywords { get; set; } = new();

public string AttachmentPath { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class CreateExamQuestionDto

{

[Required]

public int QuestionNumber { get; set; }

[Required]

public string QuestionText { get; set; } = string.Empty;

[Required]

public QuestionType Type { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal Marks { get; set; }

public bool IsRequired { get; set; } = true;

public TimeSpan? TimeLimit { get; set; }

// For multiple choice

public List<string> Options { get; set; } = new();

public string CorrectAnswer { get; set; } = string.Empty;

// For written answers

public int? WordLimit { get; set; }

public string SampleAnswer { get; set; } = string.Empty;

public bool EnableHandwritingRecognition { get; set; } = false;

public List<string> ExpectedKeywords { get; set; } = new();

public int SortOrder { get; set; }

}

public class ExamSessionDto

{

public Guid Id { get; set; }

public ExamDto Exam { get; set; } = null!;

public StudentDto Student { get; set; } = null!;

public DateTime StartTime { get; set; }

public DateTime? EndTime { get; set; }

public TimeSpan ActualDuration { get; set; }

public SessionStatus Status { get; set; }

public bool IsSubmitted { get; set; }

public int TotalQuestions { get; set; }

public int AnsweredQuestions { get; set; }

public decimal? Score { get; set; }

public decimal? Percentage { get; set; }

public bool HasViolations { get; set; }

public List<ExamAnswerDto> Answers { get; set; } = new();

}

public class ExamAnswerDto

{

public Guid Id { get; set; }

public Guid ExamQuestionId { get; set; }

public string AnswerText { get; set; } = string.Empty;

public string AttachmentPath { get; set; } = string.Empty;

public string HandwritingImagePath { get; set; } = string.Empty;

public string RecognizedText { get; set; } = string.Empty;

public decimal AiConfidenceScore { get; set; }

public decimal? MarksObtained { get; set; }

public bool IsMarked { get; set; }

public string TeacherFeedback { get; set; } = string.Empty;

public DateTime AnsweredAt { get; set; }

public TimeSpan TimeSpent { get; set; }

public bool IsFlagged { get; set; }

}

public class ExamResultDto

{

public Guid Id { get; set; }

public ExamDto Exam { get; set; } = null!;

public StudentDto Student { get; set; } = null!;

public decimal TotalMarks { get; set; }

public decimal MarksObtained { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public bool IsPassed { get; set; }

public string Remarks { get; set; } = string.Empty;

public bool IsPublished { get; set; }

public int Rank { get; set; }

public decimal? ClassAverage { get; set; }

public Dictionary<string, object> DetailedAnalysis { get; set; } = new();

}

public class StartExamSessionDto

{

[Required]

public Guid ExamId { get; set; }

[Required]

public Guid StudentId { get; set; }

public string DeviceInfo { get; set; } = string.Empty;

public string BrowserInfo { get; set; } = string.Empty;

}

public class SubmitExamAnswerDto

{

[Required]

public Guid ExamSessionId { get; set; }

[Required]

public Guid ExamQuestionId { get; set; }

public string AnswerText { get; set; } = string.Empty;

public byte[]? AttachmentFile { get; set; }

public byte[]? HandwritingImage { get; set; }

}

public class ExamStatisticsDto

{

public Guid ExamId { get; set; }

public int TotalRegistered { get; set; }

public int TotalCompleted { get; set; }

public int TotalPassed { get; set; }

public int TotalFailed { get; set; }

public decimal AverageScore { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public decimal PassPercentage { get; set; }

public Dictionary<string, int> GradeDistribution { get; set; } = new();

public List<QuestionAnalysisDto> QuestionAnalysis { get; set; } = new();

}

public class QuestionAnalysisDto

{

public Guid QuestionId { get; set; }

public int QuestionNumber { get; set; }

public decimal AverageScore { get; set; }

public int TotalAttempts { get; set; }

public int CorrectAnswers { get; set; }

public decimal DifficultyIndex { get; set; }

public List<string> CommonMistakes { get; set; } = new();

}

}// Services/Interfaces/IExamService.cs

using SchoolManagement.Core.DTOs;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Services.Interfaces

{

public interface IExamService

{

// Exam Management

Task<ExamDto> CreateExamAsync(CreateExamDto createExamDto, Guid teacherId);

Task<ExamDto> UpdateExamAsync(Guid examId, UpdateExamDto updateExamDto);

Task<bool> DeleteExamAsync(Guid examId);

Task<ExamDto> GetExamByIdAsync(Guid examId);

Task<IEnumerable<ExamDto>> GetExamsBySchoolAsync(Guid schoolId);

Task<IEnumerable<ExamDto>> GetExamsBySubjectAsync(Guid subjectId, Guid termId);

Task<IEnumerable<ExamDto>> GetExamsByGradeAsync(Guid gradeId, Guid termId);

Task<IEnumerable<ExamDto>> GetUpcomingExamsAsync(Guid schoolId, DateTime fromDate, DateTime toDate);

// Exam Publication

Task<bool> PublishExamAsync(Guid examId);

Task<bool> UnpublishExamAsync(Guid examId);

// Student Registration

Task<bool> RegisterStudentForExamAsync(Guid examId, Guid studentId);

Task<bool> RegisterStudentsForExamAsync(Guid examId, List<Guid> studentIds);

Task<bool> UnregisterStudentFromExamAsync(Guid examId, Guid studentId);

Task<IEnumerable<ExamRegistrationDto>> GetExamRegistrationsAsync(Guid examId);

// Exam Sessions

Task<ExamSessionDto> StartExamSessionAsync(StartExamSessionDto startSessionDto);

Task<ExamSessionDto> GetExamSessionAsync(Guid sessionId);

Task<bool> EndExamSessionAsync(Guid sessionId);

Task<bool> PauseExamSessionAsync(Guid sessionId);

Task<bool> ResumeExamSessionAsync(Guid sessionId);

// Answer Submission

Task<bool> SubmitAnswerAsync(SubmitExamAnswerDto submitAnswerDto);

Task<bool> SaveAnswerDraftAsync(SubmitExamAnswerDto saveAnswerDto);

Task<bool> SubmitExamAsync(Guid sessionId);

// Marking & Results

Task<bool> MarkAnswerAsync(Guid answerId, decimal marks, string feedback);

Task<bool> AutoMarkExamAsync(Guid examId);

Task<ExamResultDto> GenerateExamResultAsync(Guid sessionId);

Task<bool> PublishResultsAsync(Guid examId);

Task<IEnumerable<ExamResultDto>> GetExamResultsAsync(Guid examId);

Task<ExamResultDto> GetStudentExamResultAsync(Guid examId, Guid studentId);

// Statistics & Analytics

Task<ExamStatisticsDto> GetExamStatisticsAsync(Guid examId);

Task<IEnumerable<QuestionAnalysisDto>> GetQu

Task<IEnumerable<QuestionAnalysisDto>> GetQuestionAnalysisAsync(Guid examId);

Task<byte[]> GenerateExamReportAsync(Guid examId);

Task<byte[]> GenerateStudentExamResultAsync(Guid examResultId);

// Supervision & Security

Task<bool> AssignSupervisorAsync(Guid examId, Guid teacherId, SupervisorRole role);

Task<bool> RemoveSupervisorAsync(Guid examId, Guid teacherId);

Task<bool> CheckInSupervisorAsync(Guid examId, Guid teacherId);

Task<bool> CheckOutSupervisorAsync(Guid examId, Guid teacherId);

Task<bool> ReportIncidentAsync(CreateExamIncidentDto incidentDto);

Task<IEnumerable<ExamIncidentDto>> GetExamIncidentsAsync(Guid examId);

// Student Experience

Task<IEnumerable<ExamDto>> GetStudentUpcomingExamsAsync(Guid studentId);

Task<IEnumerable<ExamResultDto>> GetStudentExamHistoryAsync(Guid studentId);

Task<ExamDto> GetStudentExamDetailsAsync(Guid examId, Guid studentId);

Task<bool> ValidateExamAccessAsync(Guid examId, Guid studentId);

// Handwriting Integration

Task<bool> ProcessHandwrittenAnswersAsync(Guid examId);

Task<bool> ReviewHandwritingRecognitionAsync(Guid answerId, string correctedText);

}

}

// Services/Implementations/ExamService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class ExamService : IExamService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly ILogger<ExamService> \_logger;

private readonly IHandwritingRecognitionService \_handwritingService;

private readonly IPdfGenerationService \_pdfService;

private readonly INotificationService \_notificationService;

private readonly IGradingService \_gradingService;

public ExamService(

SchoolManagementDbContext context,

IMapper mapper,

ILogger<ExamService> logger,

IHandwritingRecognitionService handwritingService,

IPdfGenerationService pdfService,

INotificationService notificationService,

IGradingService gradingService)

{

\_context = context;

\_mapper = mapper;

\_logger = logger;

\_handwritingService = handwritingService;

\_pdfService = pdfService;

\_notificationService = notificationService;

\_gradingService = gradingService;

}

public async Task<ExamDto> CreateExamAsync(CreateExamDto createExamDto, Guid teacherId)

{

// Validate teacher has permission for this subject and grade

var hasPermission = await \_context.SubjectTeachers

.AnyAsync(st => st.TeacherId == teacherId &&

st.SubjectId == createExamDto.SubjectId &&

st.SchoolYearId == createExamDto.SchoolYearId &&

st.IsActive);

if (!hasPermission)

throw new UnauthorizedException("Teacher not authorized to create exam for this subject");

// Validate exam date is not in the past

if (createExamDto.ExamDate.Date < DateTime.Today)

throw new ValidationException("Exam date cannot be in the past");

var exam = \_mapper.Map<Exam>(createExamDto);

exam.CreatedByTeacherId = teacherId;

exam.EndTime = exam.StartTime.Add(exam.Duration);

// Get school ID from subject

var subject = await \_context.Subjects.FindAsync(createExamDto.SubjectId);

exam.SchoolId = subject!.SchoolId;

\_context.Exams.Add(exam);

await \_context.SaveChangesAsync();

// Add questions

if (createExamDto.Questions.Any())

{

var questions = createExamDto.Questions.Select(q =>

{

var question = \_mapper.Map<ExamQuestion>(q);

question.ExamId = exam.Id;

question.Options = q.Options.Any() ? System.Text.Json.JsonSerializer.Serialize(q.Options) : "";

question.ExpectedKeywords = q.ExpectedKeywords.Any() ? System.Text.Json.JsonSerializer.Serialize(q.ExpectedKeywords) : "";

return question;

}).ToList();

\_context.ExamQuestions.AddRange(questions);

}

// Assign supervisors

if (createExamDto.SupervisorIds.Any())

{

var supervisors = createExamDto.SupervisorIds.Select(supervisorId => new ExamSupervisor

{

ExamId = exam.Id,

TeacherId = supervisorId,

Role = SupervisorRole.RoomSupervisor

}).ToList();

\_context.ExamSupervisors.AddRange(supervisors);

}

await \_context.SaveChangesAsync();

return await GetExamByIdAsync(exam.Id);

}

public async Task<ExamDto> UpdateExamAsync(Guid examId, UpdateExamDto updateExamDto)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null)

throw new NotFoundException($"Exam {examId} not found");

if (exam.Status == ExamStatus.InProgress || exam.Status == ExamStatus.Completed)

throw new ConflictException("Cannot update exam that is in progress or completed");

\_mapper.Map(updateExamDto, exam);

if (updateExamDto.Duration.HasValue && updateExamDto.StartTime.HasValue)

{

exam.EndTime = exam.StartTime.Add(exam.Duration);

}

await \_context.SaveChangesAsync();

return await GetExamByIdAsync(examId);

}

public async Task<bool> DeleteExamAsync(Guid examId)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null)

return false;

if (exam.Status == ExamStatus.InProgress)

throw new ConflictException("Cannot delete exam that is in progress");

exam.IsDeleted = true;

await \_context.SaveChangesAsync();

return true;

}

public async Task<ExamDto> GetExamByIdAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.Include(e => e.Questions.OrderBy(q => q.SortOrder))

.Include(e => e.Supervisors)

.ThenInclude(s => s.Teacher)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null)

throw new NotFoundException($"Exam {examId} not found");

var examDto = \_mapper.Map<ExamDto>(exam);

// Get registration counts

examDto.RegisteredStudents = await \_context.ExamRegistrations

.CountAsync(r => r.ExamId == examId);

examDto.CompletedStudents = await \_context.ExamSessions

.CountAsync(s => s.ExamId == examId && s.IsSubmitted);

return examDto;

}

public async Task<IEnumerable<ExamDto>> GetExamsBySchoolAsync(Guid schoolId)

{

var exams = await \_context.Exams

.Where(e => e.SchoolId == schoolId)

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderByDescending(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<IEnumerable<ExamDto>> GetExamsBySubjectAsync(Guid subjectId, Guid termId)

{

var exams = await \_context.Exams

.Where(e => e.SubjectId == subjectId && e.TermId == termId)

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<IEnumerable<ExamDto>> GetExamsByGradeAsync(Guid gradeId, Guid termId)

{

var exams = await \_context.Exams

.Where(e => e.GradeId == gradeId && e.TermId == termId)

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<IEnumerable<ExamDto>> GetUpcomingExamsAsync(Guid schoolId, DateTime fromDate, DateTime toDate)

{

var exams = await \_context.Exams

.Where(e => e.SchoolId == schoolId &&

e.ExamDate >= fromDate &&

e.ExamDate <= toDate &&

e.IsPublished)

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<bool> PublishExamAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Questions)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null)

return false;

// Validate exam is ready for publication

if (!exam.Questions.Any())

throw new ValidationException("Cannot publish exam without questions");

if (exam.ExamDate <= DateTime.Now)

throw new ValidationException("Cannot publish exam with past date");

exam.IsPublished = true;

exam.PublishedAt = DateTime.UtcNow;

exam.Status = ExamStatus.Published;

await \_context.SaveChangesAsync();

// Auto-register eligible students

await AutoRegisterStudentsAsync(examId);

// Send notifications

await NotifyStudentsAboutExamAsync(examId);

return true;

}

public async Task<bool> UnpublishExamAsync(Guid examId)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null)

return false;

if (exam.Status == ExamStatus.InProgress || exam.Status == ExamStatus.Completed)

throw new ConflictException("Cannot unpublish exam that is in progress or completed");

exam.IsPublished = false;

exam.Status = ExamStatus.Draft;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> RegisterStudentForExamAsync(Guid examId, Guid studentId)

{

// Check if already registered

var existingRegistration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

if (existingRegistration != null)

return true; // Already registered

// Validate student is in the correct grade

var exam = await \_context.Exams

.Include(e => e.Grade)

.FirstOrDefaultAsync(e => e.Id == examId);

var student = await \_context.Students.FindAsync(studentId);

if (exam == null || student == null)

return false;

if (student.CurrentGradeId != exam.GradeId)

throw new ValidationException("Student is not in the correct grade for this exam");

var registration = new ExamRegistration

{

ExamId = examId,

StudentId = studentId,

RegistrationDate = DateTime.UtcNow,

Status = RegistrationStatus.Registered

};

\_context.ExamRegistrations.Add(registration);

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> RegisterStudentsForExamAsync(Guid examId, List<Guid> studentIds)

{

var tasks = studentIds.Select(studentId => RegisterStudentForExamAsync(examId, studentId));

var results = await Task.WhenAll(tasks);

return results.All(r => r);

}

public async Task<ExamSessionDto> StartExamSessionAsync(StartExamSessionDto startSessionDto)

{

// Validate exam is available

var exam = await \_context.Exams

.Include(e => e.Questions)

.FirstOrDefaultAsync(e => e.Id == startSessionDto.ExamId);

if (exam == null)

throw new NotFoundException("Exam not found");

if (!exam.IsPublished)

throw new ValidationException("Exam is not published");

if (exam.Status != ExamStatus.Published)

throw new ValidationException("Exam is not available for taking");

// Check exam timing

var currentTime = DateTime.Now.TimeOfDay;

var examDate = exam.ExamDate.Date;

if (DateTime.Today != examDate)

throw new ValidationException("Exam is not scheduled for today");

if (currentTime < exam.StartTime)

throw new ValidationException("Exam has not started yet");

if (currentTime > exam.EndTime)

throw new ValidationException("Exam time has ended");

// Check if student is registered

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == startSessionDto.ExamId &&

r.StudentId == startSessionDto.StudentId &&

r.Status == RegistrationStatus.Registered);

if (registration == null)

throw new ValidationException("Student is not registered for this exam");

// Check if session already exists

var existingSession = await \_context.ExamSessions

.FirstOrDefaultAsync(s => s.ExamId == startSessionDto.ExamId &&

s.StudentId == startSessionDto.StudentId);

if (existingSession != null)

{

if (existingSession.IsSubmitted)

throw new ValidationException("Exam has already been submitted");

// Resume existing session

existingSession.Status = SessionStatus.InProgress;

await \_context.SaveChangesAsync();

return \_mapper.Map<ExamSessionDto>(existingSession);

}

// Create new session

var session = new ExamSession

{

ExamId = startSessionDto.ExamId,

StudentId = startSessionDto.StudentId,

StartTime = DateTime.UtcNow,

Status = SessionStatus.InProgress,

TotalQuestions = exam.Questions.Count,

DeviceInfo = startSessionDto.DeviceInfo,

BrowserInfo = startSessionDto.BrowserInfo,

IPAddress = GetClientIPAddress()

};

\_context.ExamSessions.Add(session);

// Mark student as present

registration.IsPresent = true;

registration.CheckInTime = DateTime.UtcNow;

registration.Status = RegistrationStatus.Present;

// Log session start

var log = new ExamSessionLog

{

ExamSessionId = session.Id,

Type = LogType.SessionStart,

Event = "Session Started",

Details = System.Text.Json.JsonSerializer.Serialize(new { Device = startSessionDto.DeviceInfo, Browser = startSessionDto.BrowserInfo }),

IPAddress = session.IPAddress

};

\_context.ExamSessionLogs.Add(log);

await \_context.SaveChangesAsync();

return \_mapper.Map<ExamSessionDto>(session);

}

public async Task<bool> SubmitAnswerAsync(SubmitExamAnswerDto submitAnswerDto)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.FirstOrDefaultAsync(s => s.Id == submitAnswerDto.ExamSessionId);

if (session == null || session.IsSubmitted)

return false;

var question = await \_context.ExamQuestions

.FirstOrDefaultAsync(q => q.Id == submitAnswerDto.ExamQuestionId);

if (question == null)

return false;

// Check if answer already exists

var existingAnswer = await \_context.ExamAnswers

.FirstOrDefaultAsync(a => a.ExamSessionId == submitAnswerDto.ExamSessionId &&

a.ExamQuestionId == submitAnswerDto.ExamQuestionId);

ExamAnswer answer;

if (existingAnswer != null)

{

// Update existing answer

answer = existingAnswer;

answer.AnswerText = submitAnswerDto.AnswerText;

answer.AnsweredAt = DateTime.UtcNow;

answer.AttemptNumber++;

}

else

{

// Create new answer

answer = new ExamAnswer

{

ExamSessionId = submitAnswerDto.ExamSessionId,

ExamQuestionId = submitAnswerDto.ExamQuestionId,

StudentId = session.StudentId,

AnswerText = submitAnswerDto.AnswerText,

AnsweredAt = DateTime.UtcNow,

AttemptNumber = 1

};

\_context.ExamAnswers.Add(answer);

}

// Handle file attachments

if (submitAnswerDto.AttachmentFile != null)

{

var attachmentPath = await SaveAttachmentAsync(

submitAnswerDto.AttachmentFile,

session.StudentId,

submitAnswerDto.ExamSessionId);

answer.AttachmentPath = attachmentPath;

}

// Handle handwriting image

if (submitAnswerDto.HandwritingImage != null && question.EnableHandwritingRecognition)

{

var imagePath = await SaveHandwritingImageAsync(

submitAnswerDto.HandwritingImage,

session.StudentId,

submitAnswerDto.ExamSessionId);

answer.HandwritingImagePath = imagePath;

// Process handwriting asynchronously

\_ = Task.Run(() => ProcessHandwritingAnswerAsync(answer.Id, imagePath, question.ExpectedKeywords));

}

// Auto-mark if possible (multiple choice, true/false)

if (question.Type == QuestionType.MultipleChoice || question.Type == QuestionType.TrueFalse)

{

if (answer.AnswerText.Trim().Equals(question.CorrectAnswer.Trim(), StringComparison.OrdinalIgnoreCase))

{

answer.MarksObtained = question.Marks;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

else

{

answer.MarksObtained = 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

}

// Update session progress

var answeredQuestions = await \_context.ExamAnswers

.CountAsync(a => a.ExamSessionId == submitAnswerDto.ExamSessionId && !string.IsNullOrEmpty(a.AnswerText));

session.AnsweredQuestions = answeredQuestions;

// Log answer submission

var log = new ExamSessionLog

{

ExamSessionId = submitAnswerDto.ExamSessionId,

Type = LogType.AnswerSaved,

Event = "Answer Submitted",

Details = System.Text.Json.JsonSerializer.Serialize(new { QuestionId = submitAnswerDto.ExamQuestionId, AnswerLength = submitAnswerDto.AnswerText.Length }),

IPAddress = GetClientIPAddress()

};

\_context.ExamSessionLogs.Add(log);

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> SubmitExamAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == sessionId);

if (session == null || session.IsSubmitted)

return false;

session.IsSubmitted = true;

session.EndTime = DateTime.UtcNow;

session.ActualDuration = session.EndTime.Value - session.StartTime;

session.Status = SessionStatus.Completed;

// Update registration status

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == session.ExamId && r.StudentId == session.StudentId);

if (registration != null)

{

registration.CheckOutTime = DateTime.UtcNow;

}

// Log submission

var log = new ExamSessionLog

{

ExamSessionId = sessionId,

Type = LogType.SessionEnd,

Event = "Exam Submitted",

Details = System.Text.Json.JsonSerializer.Serialize(new { TotalAnswers = session.Answers.Count, Duration = session.ActualDuration }),

IPAddress = GetClientIPAddress()

};

\_context.ExamSessionLogs.Add(log);

await \_context.SaveChangesAsync();

// Start auto-marking process

\_ = Task.Run(() => AutoMarkSessionAsync(sessionId));

return true;

}

public async Task<bool> MarkAnswerAsync(Guid answerId, decimal marks, string feedback)

{

var answer = await \_context.ExamAnswers

.Include(a => a.ExamQuestion)

.FirstOrDefaultAsync(a => a.Id == answerId);

if (answer == null)

return false;

if (marks > answer.ExamQuestion.Marks)

throw new ValidationException($"Marks cannot exceed question maximum of {answer.ExamQuestion.Marks}");

answer.MarksObtained = marks;

answer.TeacherFeedback = feedback;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

// Update session and result if all questions are marked

await UpdateSessionResultAsync(answer.ExamSessionId);

return true;

}

public async Task<ExamResultDto> GenerateExamResultAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.ThenInclude(e => e.Subject)

.Include(s => s.Exam)

.ThenInclude(e => e.Grade)

.Include(s => s.Student)

.Include(s => s.Answers)

.ThenInclude(a => a.ExamQuestion)

.FirstOrDefaultAsync(s => s.Id == sessionId);

if (session == null)

throw new NotFoundException("Exam session not found");

var marksObtained = session.Answers.Where(a => a.IsMarked).Sum(a => a.MarksObtained ?? 0);

var totalMarks = session.Exam.TotalMarks;

var percentage = totalMarks > 0 ? (marksObtained / totalMarks) \* 100 : 0;

// Calculate grade using school's grading scheme

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

.FirstOrDefaultAsync(s => s.Id == session.StudentId);

string grade = "F";

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

percentage);

grade = gradeCalculation.Symbol;

}

// Check if result already exists

var existingResult = await \_context.ExamResults

.FirstOrDefaultAsync(r => r.ExamSessionId == sessionId);

ExamResult result;

if (existingResult != null)

{

result = existingResult;

}

else

{

result = new ExamResult

{

ExamId = session.ExamId,

StudentId = session.StudentId,

ExamSessionId = sessionId

};

\_context.ExamResults.Add(result);

}

result.TotalMarks = totalMarks;

result.MarksObtained = marksObtained;

result.Percentage = percentage;

result.Grade = grade;

result.IsPassed = percentage >= ((session.Exam.PassingMarks / session.Exam.TotalMarks) \* 100);

result.QuestionsMarked = session.Answers.Count(a => a.IsMarked);

result.TotalQuestions = session.Answers.Count;

result.IsFullyMarked = result.QuestionsMarked == result.TotalQuestions;

if (result.IsFullyMarked)

{

result.Status = ResultStatus.Completed;

}

await \_context.SaveChangesAsync();

return \_mapper.Map<ExamResultDto>(result);

}

public async Task<ExamStatisticsDto> GetExamStatisticsAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Questions)

.Include(e => e.Results)

.ThenInclude(r => r.Student)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null)

throw new NotFoundException("Exam not found");

var totalRegistered = await \_context.ExamRegistrations.CountAsync(r => r.ExamId == examId);

var totalCompleted = exam.Results.Count(r => r.IsFullyMarked);

var totalPassed = exam.Results.Count(r => r.IsPassed);

var totalFailed = totalCompleted - totalPassed;

var completedResults = exam.Results.Where(r => r.IsFullyMarked).ToList();

var averageScore = completedResults.Any() ? completedResults.Average(r => r.Percentage) : 0;

var highestScore = completedResults.Any() ? completedResults.Max(r => r.Percentage) : 0;

var lowestScore = completedResults.Any() ? completedResults.Min(r => r.Percentage) : 0;

var gradeDistribution = completedResults

.GroupBy(r => r.Grade)

.ToDictionary(g => g.Key, g => g.Count());

// Question analysis

var questionAnalysis = new List<QuestionAnalysisDto>();

foreach (var question in exam.Questions)

{

var questionAnswers = await \_context.ExamAnswers

.Where(a => a.ExamQuestionId == question.Id && a.IsMarked)

.ToListAsync();

if (questionAnswers.Any())

{

var analysis = new QuestionAnalysisDto

{

QuestionId = question.Id,

QuestionNumber = question.QuestionNumber,

AverageScore = questionAnswers.Average(a => a.MarksObtained ?? 0),

TotalAttempts = questionAnswers.Count,

CorrectAnswers = questionAnswers.Count(a => (a.MarksObtained ?? 0) == question.Marks),

DifficultyIndex = questionAnswers.Count > 0 ?

questionAnswers.Average(a => (a.MarksObtained ?? 0) / question.Marks) : 0

};

questionAnalysis.Add(analysis);

}

}

return new ExamStatisticsDto

{

ExamId = examId,

TotalRegistered = totalRegistered,

TotalCompleted = totalCompleted,

TotalPassed = totalPassed,

TotalFailed = totalFailed,

AverageScore = averageScore,

HighestScore = highestScore,

LowestScore = lowestScore,

PassPercentage = totalCompleted > 0 ? (decimal)totalPassed / totalCompleted \* 100 : 0,

GradeDistribution = gradeDistribution,

QuestionAnalysis = questionAnalysis

};

}

// Private helper methods

private async Task AutoRegisterStudentsAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Grade)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null) return;

// Get all students in the grade who are enrolled in the subject

var eligibleStudents = await \_context.Students

.Where(s => s.CurrentGradeId == exam.GradeId)

.Where(s => s.StudentSubjects.Any(ss => ss.SubjectId == exam.SubjectId &&

ss.SchoolYearId == exam.SchoolYearId &&

ss.IsActive))

.Select(s => s.Id)

.ToListAsync();

await RegisterStudentsForExamAsync(examId, eligibleStudents);

}

private async Task NotifyStudentsAboutExamAsync(Guid examId)

{

var registrations = await \_context.ExamRegistrations

.Where(r => r.ExamId == examId)

.Include(r => r.Student)

.ThenInclude(s => s.Parents)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.ToListAsync();

foreach (var registration in registrations)

{

// Notify student and parents

var exam = registration.Exam;

var student = registration.Student;

var message = $"Exam Alert: {exam.Subject.Name} exam is scheduled for {exam.ExamDate:dd/MM/yyyy} at {exam.StartTime}. Duration: {exam.Duration}. Venue: {exam.Venue}";

foreach (var parent in student.Parents)

{

await \_notificationService.CreateNotificationAsync(new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = student.Id,

Title = $"Exam Notification - {exam.Subject.Name}",

Message = message,

Type = NotificationType.General,

Channel = NotificationChannel.WhatsApp

});

}

}

}

private async Task ProcessHandwritingAnswerAsync(Guid answerId, string imagePath, string expectedKeywordsJson)

{

try

{

var answer = await \_context.ExamAnswers

.Include(a => a.Student)

.FirstOrDefaultAsync(a => a.Id == answerId);

if (answer == null) return;

var processDto = new ProcessHandwritingDto

{

StudentId = answer.StudentId,

ImagePath = imagePath,

ExpectedAnswer = !string.IsNullOrEmpty(expectedKeywordsJson) ?

string.Join(" ", System.Text.Json.JsonSerializer.Deserialize<List<string>>(expectedKeywordsJson) ?? new List<string>()) :

""

};

var result = await \_handwritingService.ProcessHandwritingAsync(processDto);

answer.RecognizedText = result.RecognizedText;

answer.AiConfidenceScore = result.OverallConfidence;

answer.IsAiProcessed = true;

// If confidence is high enough, auto-mark

if (result.OverallConfidence > 0.8m && result.IsCorrect)

{

var question = await \_context.ExamQuestions.FindAsync(answer.ExamQuestionId);

answer.MarksObtained = question?.Marks ?? 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

else if (result.OverallConfidence < 0.6m)

{

// Flag for teacher review

answer.IsFlagged = true;

answer.FlagReason = "Low AI confidence in handwriting recognition";

}

await \_context.SaveChangesAsync();

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error processing handwriting for answer {AnswerId}", answerId);

}

}

private async Task AutoMarkSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Answers)

.ThenInclude(a => a.ExamQuestion)

.FirstOrDefaultAsync(s => s.Id == sessionId);

if (session == null) return;

foreach (var answer in session.Answers.Where(a => !a.IsMarked))

{

var question = answer.ExamQuestion;

// Auto-mark objective questions

if (question.Type == QuestionType.MultipleChoice || question.Type == QuestionType.TrueFalse)

{

var isCorrect = answer.AnswerText.Trim().Equals(question.CorrectAnswer.Trim(), StringComparison.OrdinalIgnoreCase);

answer.MarksObtained = isCorrect ? question.Marks : 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

// Auto-mark numerical questions with exact match

else if (question.Type == QuestionType.Numerical)

{

if (decimal.TryParse(answer.AnswerText.Trim(), out var studentAnswer) &&

decimal.TryParse(question.CorrectAnswer.Trim(), out var correctAnswer))

{

var tolerance = 0.01m; // Allow small rounding differences

var isCorrect = Math.Abs(studentAnswer - correctAnswer) <= tolerance;

answer.MarksObtained = isCorrect ? question.Marks : 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

}

}

await \_context.SaveChangesAsync();

await UpdateSessionResultAsync(sessionId);

}

private async Task UpdateSessionResultAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == sessionId);

if (session == null) return;

var markedAnswers = session.Answers.Where(a => a.IsMarked).ToList();

var totalMarks = markedAnswers.Sum(a => a.MarksObtained ?? 0);

var totalPossible = markedAnswers.Sum(a => a.ExamQuestion?.Marks ?? 0);

session.Score = totalMarks;

session.Percentage = totalPossible > 0 ? (totalMarks / totalPossible) \* 100 : 0;

await \_context.SaveChangesAsync();

// Generate result if all questions are marked

if (session.Answers.All(a => a.IsMarked))

{

await GenerateExamResultAsync(sessionId);

}

}

private async Task<string> SaveAttachmentAsync(byte[] fileData, Guid studentId, Guid sessionId)

{

// Implementation would save to file storage service

var fileName = $"exam\_attachment\_{studentId}\_{sessionId}\_{Guid.NewGuid()}.pdf";

var filePath = $"exam\_attachments/{fileName}";

// Save file logic here

await Task.Delay(10); // Placeholder

return filePath;

}

private async Task<string> SaveHandwritingImageAsync(byte[] imageData, Guid studentId, Guid sessionId)

{

// Implementation would save to file storage service

var fileName = $"handwriting\_{studentId}\_{sessionId}\_{Guid.NewGuid()}.jpg";

var filePath = $"exam\_handwriting/{fileName}";

// Save image logic here

await Task.Delay(10); // Placeholder

return filePath;

}

private string GetClientIPAddress()

{

// Implementation would get actual client IP

return "127.0.0.1"; // Placeholder

}

public async Task<ExamDto> GetStudentExamDetailsAsync(Guid examId, Guid studentId)

{

var exam = await GetExamByIdAsync(examId);

// Check if student is registered

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

if (registration == null)

throw new UnauthorizedException("Student is not registered for this exam");

return exam;

}

public async Task<bool> ValidateExamAccessAsync(Guid examId, Guid studentId)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null || !exam.IsPublished)

return false;

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

return registration != null && registration.Status == RegistrationStatus.Registered;

}

public async Task<IEnumerable<ExamDto>> GetStudentUpcomingExamsAsync(Guid studentId)

{

var upcomingExams = await \_context.ExamRegistrations

.Where(r => r.StudentId == studentId &&

r.Status == RegistrationStatus.Registered &&

r.Exam.ExamDate >= DateTime.Today &&

r.Exam.IsPublished)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.Include(r => r.Exam)

.ThenInclude(e => e.Grade)

.Include(r => r.Exam)

.ThenInclude(e => e.Term)

.Select(r => r.Exam)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(upcomingExams);

}

public async Task<IEnumerable<ExamResultDto>> GetStudentExamHistoryAsync(Guid studentId)

{

var results = await \_context.ExamResults

.Where(r => r.StudentId == studentId && r.IsPublished)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.Include(r => r.Exam)

.ThenInclude(e => e.Grade)

.Include(r => r.Student)

.OrderByDescending(r => r.Exam.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamResultDto>>(results);

}

public async Task<bool> ReportIncidentAsync(CreateExamIncidentDto incidentDto)

{

var incident = \_mapper.Map<ExamIncident>(incidentDto);

incident.IncidentTime = DateTime.UtcNow;

\_context.ExamIncidents.Add(incident);

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<ExamIncidentDto>> GetExamIncidentsAsync(Guid examId)

{

var incidents = await \_context.ExamIncidents

.Where(i => i.ExamId == examId)

.Include(i => i.Student)

.Include(i => i.ReportedByTeacher)

.OrderByDescending(i => i.IncidentTime)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamIncidentDto>>(incidents);

}

public async Task<bool> AssignSupervisorAsync(Guid examId, Guid teacherId, SupervisorRole role)

{

var existingSupervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (existingSupervisor != null)

{

existingSupervisor.Role = role;

}

else

{

var supervisor = new ExamSupervisor

{

ExamId = examId,

TeacherId = teacherId,

Role = role,

AssignedAt = DateTime.UtcNow

};

\_context.ExamSupervisors.Add(supervisor);

}

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> PublishResultsAsync(Guid examId)

{

var results = await \_context.ExamResults

.Where(r => r.ExamId == examId && r.IsFullyMarked)

.ToListAsync();

foreach (var result in results)

{

result.IsPublished = true;

result.PublishedAt = DateTime.UtcNow;

result.Status = ResultStatus.Published;

}

await \_context.SaveChangesAsync();

// Notify students and parents

await NotifyResultsPublishedAsync(examId);

return true;

}

private async Task NotifyResultsPublishedAsync(Guid examId)

{

var results = await \_context.ExamResults

.Where(r => r.ExamId == examId && r.IsPublished)

.Include(r => r.Student)

.ThenInclude(s => s.Parents)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.ToListAsync();

foreach (var result in results)

{

var message = $"Exam results published for {result.Exam.Subject.Name}. " +

$"Score: {result.MarksObtained}/{result.TotalMarks} ({result.Percentage:F1}%). " +

$"Grade: {result.Grade}. {(result.IsPassed ? "PASSED" : "FAILED")}";

foreach (var parent in result.Student.Parents)

{

await \_notificationService.CreateNotificationAsync(new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = result.StudentId,

Title = $"Exam Results - {result.Exam.Subject.Name}",

Message = message,

Type = NotificationType.Results,

Channel = NotificationChannel.WhatsApp

});

}

}

}

// Additional interface methods implementation

public async Task<ExamSessionDto> GetExamSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.Include(s => s.Student)

.Include(s => s.Answers)

.FirstOrDefaultAsync(s => s.Id == sessionId);

if (session == null)

throw new NotFoundException($"Exam session {sessionId} not found");

return \_mapper.Map<ExamSessionDto>(session);

}

public async Task<bool> EndExamSessionAsync(Guid sessionId)

{

return await SubmitExamAsync(sessionId);

}

public async Task<bool> PauseExamSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions.FindAsync(sessionId);

if (session == null || session.IsSubmitted)

return false;

session.Status = SessionStatus.Paused;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> ResumeExamSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions.FindAsync(sessionId);

if (session == null || session.IsSubmitted)

return false;

session.Status = SessionStatus.InProgress;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> SaveAnswerDraftAsync(SubmitExamAnswerDto saveAnswerDto)

{

// Same as SubmitAnswerAsync but mark as draft

return await SubmitAnswerAsync(saveAnswerDto);

}

public async Task<bool> AutoMarkExamAsync(Guid examId)

{

var sessions = await \_context.ExamSessions

.Where(s => s.ExamId == examId && s.IsSubmitted)

.ToListAsync();

foreach (var session in sessions)

{

await AutoMarkSessionAsync(session.Id);

}

return true;

}

public async Task<IEnumerable<ExamResultDto>> GetExamResultsAsync(Guid examId)

{

var results = await \_context.ExamResults

.Where(r => r.ExamId == examId)

.Include(r => r.Student)

.Include(r => r.Exam)

.OrderByDescending(r => r.Percentage)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamResultDto>>(results);

}

public async Task<ExamResultDto> GetStudentExamResultAsync(Guid examId, Guid studentId)

{

var result = await \_context.ExamResults

.Where(r => r.ExamId == examId && r.StudentId == studentId)

.Include(r => r.Student)

.Include(r => r.Exam)

.FirstOrDefaultAsync();

if (result == null)

throw new NotFoundException("Exam result not found");

return \_mapper.Map<ExamResultDto>(result);

}

public async Task<IEnumerable<QuestionAnalysisDto>> GetQuestionAnalysisAsync(Guid examId)

{

var stats = await GetExamStatisticsAsync(examId);

return stats.QuestionAnalysis;

}

public async Task<byte[]> GenerateExamReportAsync(Guid examId)

{

var stats = await GetExamStatisticsAsync(examId);

return await \_pdfService.GenerateExamAnalysisReportAsync(stats);

}

public async Task<byte[]> GenerateStudentExamResultAsync(Guid examResultId)

{

var result = await \_context.ExamResults

.Include(r => r.Student)

.Include(r => r.Exam)

.FirstOrDefaultAsync(r => r.Id == examResultId);

if (result == null)

throw new NotFoundException("Exam result not found");

var resultDto = \_mapper.Map<ExamResultDto>(result);

return await \_pdfService.GenerateStudentExamResultAsync(resultDto);

}

public async Task<bool> RemoveSupervisorAsync(Guid examId, Guid teacherId)

{

var supervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (supervisor == null)

return false;

\_context.ExamSupervisors.Remove(supervisor);

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> CheckInSupervisorAsync(Guid examId, Guid teacherId)

{

var supervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (supervisor == null)

return false;

supervisor.IsPresent = true;

supervisor.CheckInTime = DateTime.UtcNow;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> CheckOutSupervisorAsync(Guid examId, Guid teacherId)

{

var supervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (supervisor == null)

return false;

supervisor.CheckOutTime = DateTime.UtcNow;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> ProcessHandwrittenAnswersAsync(Guid examId)

{

var answers = await \_context.ExamAnswers

.Where(a => a.ExamSession.ExamId == examId &&

!string.IsNullOrEmpty(a.HandwritingImagePath) &&

!a.IsAiProcessed)

.Include(a => a.ExamQuestion)

.ToListAsync();

foreach (var answer in answers)

{

await ProcessHandwritingAnswerAsync(answer.Id, answer.HandwritingImagePath, answer.ExamQuestion.ExpectedKeywords);

}

return true;

}

public async Task<bool> ReviewHandwritingRecognitionAsync(Guid answerId, string correctedText)

{

var answer = await \_context.ExamAnswers.FindAsync(answerId);

if (answer == null)

return false;

answer.RecognizedText = correctedText;

answer.AnswerText = correctedText;

answer.IsFlagged = false;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> UnregisterStudentFromExamAsync(Guid examId, Guid studentId)

{

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

if (registration == null)

return false;

registration.Status = RegistrationStatus.Withdrawn;

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<ExamRegistrationDto>> GetExamRegistrationsAsync(Guid examId)

{

var registrations = await \_context.ExamRegistrations

.Where(r => r.ExamId == examId)

.Include(r => r.Student)

.Include(r => r.Exam)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamRegistrationDto>>(registrations);

}

}

}

// Additional DTOs for Exam Management

namespace SchoolManagement.Core.DTOs

{

public class UpdateExamDto

{

[StringLength(200)]

public string? Title { get; set; }

[StringLength(1000)]

public string? Description { get; set; }

public DateTime? ExamDate { get; set; }

public TimeSpan? Duration { get; set; }

public TimeSpan? StartTime { get; set; }

[Range(0.01, double.MaxValue)]

public decimal? TotalMarks { get; set; }

[Range(0, double.MaxValue)]

public decimal? PassingMarks { get; set; }

[StringLength(100)]

public string? Venue { get; set; }

public bool? AllowHandwritingRecognition { get; set; }

public bool? IsOnline { get; set; }

[StringLength(500)]

public string? Instructions { get; set; }

}

public class ExamRegistrationDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public ExamDto Exam { get; set; } = null!;

public DateTime RegistrationDate { get; set; }

public RegistrationStatus Status { get; set; }

public bool IsPresent { get; set; }

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

public string SeatNumber { get; set; } = string.Empty;

public bool RequiresSpecialAccommodation { get; set; }

public string SpecialAccommodations { get; set; } = string.Empty;

public string Remarks { get; set; } = string.Empty;

}

public class ExamSupervisorDto

{

public Guid Id { get; set; }

public TeacherDto Teacher { get; set; } = null!;

public SupervisorRole Role { get; set; }

public DateTime AssignedAt { get; set; }

public bool IsPresent { get; set; }

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

public string Responsibilities { get; set; } = string.Empty;

public string Notes { get; set; } = string.Empty;

}

public class ExamIncidentDto

{

public Guid Id { get; set; }

public StudentDto? Student { get; set; }

public TeacherDto ReportedByTeacher { get; set; } = null!;

public IncidentType Type { get; set; }

public IncidentSeverity Severity { get; set; }

public string Description { get; set; } = string.Empty;

public DateTime IncidentTime { get; set; }

public string Location { get; set; } = string.Empty;

public string ActionTaken { get; set; } = string.Empty;

public bool RequiresFollowUp { get; set; }

public IncidentStatus Status { get; set; }

}

public class CreateExamIncidentDto

{

[Required]

public Guid ExamId { get; set; }

public Guid? StudentId { get; set; }

[Required]

public Guid ReportedByTeacherId { get; set; }

[Required]

public IncidentType Type { get; set; }

[Required]

public IncidentSeverity Severity { get; set; }

[Required]

public string Description { get; set; } = string.Empty;

public string Location { get; set; } = string.Empty;

public string ActionTaken { get; set; } = string.Empty;

public bool RequiresFollowUp { get; set; } = false;

}

}// Services/Interfaces/IExamService.cs

using SchoolManagement.Core.DTOs;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Services.Interfaces

{

public interface IExamService

{

// Exam Management

Task<ExamDto> CreateExamAsync(CreateExamDto createExamDto, Guid teacherId);

Task<ExamDto> UpdateExamAsync(Guid examId, UpdateExamDto updateExamDto);

Task<bool> DeleteExamAsync(Guid examId);

Task<ExamDto> GetExamByIdAsync(Guid examId);

Task<IEnumerable<ExamDto>> GetExamsBySchoolAsync(Guid schoolId);

Task<IEnumerable<ExamDto>> GetExamsBySubjectAsync(Guid subjectId, Guid termId);

Task<IEnumerable<ExamDto>> GetExamsByGradeAsync(Guid gradeId, Guid termId);

Task<IEnumerable<ExamDto>> GetUpcomingExamsAsync(Guid schoolId, DateTime fromDate, DateTime toDate);

// Exam Publication

Task<bool> PublishExamAsync(Guid examId);

Task<bool> UnpublishExamAsync(Guid examId);

// Student Registration

Task<bool> RegisterStudentForExamAsync(Guid examId, Guid studentId);

Task<bool> RegisterStudentsForExamAsync(Guid examId, List<Guid> studentIds);

Task<bool> UnregisterStudentFromExamAsync(Guid examId, Guid studentId);

Task<IEnumerable<ExamRegistrationDto>> GetExamRegistrationsAsync(Guid examId);

// Exam Sessions

Task<ExamSessionDto> StartExamSessionAsync(StartExamSessionDto startSessionDto);

Task<ExamSessionDto> GetExamSessionAsync(Guid sessionId);

Task<bool> EndExamSessionAsync(Guid sessionId);

Task<bool> PauseExamSessionAsync(Guid sessionId);

Task<bool> ResumeExamSessionAsync(Guid sessionId);

// Answer Submission

Task<bool> SubmitAnswerAsync(SubmitExamAnswerDto submitAnswerDto);

Task<bool> SaveAnswerDraftAsync(SubmitExamAnswerDto saveAnswerDto);

Task<bool> SubmitExamAsync(Guid sessionId);

// Marking & Results

Task<bool> MarkAnswerAsync(Guid answerId, decimal marks, string feedback);

Task<bool> AutoMarkExamAsync(Guid examId);

Task<ExamResultDto> GenerateExamResultAsync(Guid sessionId);

Task<bool> PublishResultsAsync(Guid examId);

Task<IEnumerable<ExamResultDto>> GetExamResultsAsync(Guid examId);

Task<ExamResultDto> GetStudentExamResultAsync(Guid examId, Guid studentId);

// Statistics & Analytics

Task<ExamStatisticsDto> GetExamStatisticsAsync(Guid examId);

Task<IEnumerable<QuestionAnalysisDto>> GetQu