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