Second Version

Data/SchoolManagementDbContext.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Data

{

public class SchoolManagementDbContext : DbContext

{

public SchoolManagementDbContext(DbContextOptions<SchoolManagementDbContext> options)

: base(options)

{

}

// Core Entities

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

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

public DbSet<GradingScale> GradingScales { get; set; }

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

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

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

public DbSet<Grade> Grades { get; set; }

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

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

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

// Relationship Entities

public DbSet<GradeSubject> GradeSubjects { get; set; }

public DbSet<SubjectTeacher> SubjectTeachers { get; set; }

public DbSet<StudentSubject> StudentSubjects { get; set; }

public DbSet<ClassTeacher> ClassTeachers { get; set; }

// Academic Records

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

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

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

// Timetable

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

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

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

// AI Handwriting Recognition

public DbSet<HandwritingTrainingData> HandwritingTrainingData { get; set; }

public DbSet<HandwritingCharacterData> HandwritingCharacterData { get; set; }

public DbSet<HandwritingModel> HandwritingModels { get; set; }

public DbSet<HandwritingRecognitionResult> HandwritingRecognitionResults { get; set; }

public DbSet<TrainingExercise> TrainingExercises { get; set; }

public DbSet<StudentTrainingSession> StudentTrainingSessions { get; set; }

public DbSet<ModelTrainingJob> ModelTrainingJobs { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

// Configure entity relationships and constraints

ConfigureSchoolEntities(modelBuilder);

ConfigureGradingEntities(modelBuilder);

ConfigureUserEntities(modelBuilder);

ConfigureAcademicEntities(modelBuilder);

ConfigureTimetableEntities(modelBuilder);

ConfigureAIEntities(modelBuilder);

ConfigureIndexes(modelBuilder);

ConfigureSoftDelete(modelBuilder);

}

private void ConfigureSchoolEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<School>(entity =>

{

entity.HasIndex(e => e.RegistrationNumber).IsUnique();

entity.HasIndex(e => e.Email).IsUnique();

entity.Property(e => e.Name).IsRequired();

});

modelBuilder.Entity<SchoolYear>(entity =>

{

entity.HasOne(sy => sy.School)

.WithMany(s => s.SchoolYears)

.HasForeignKey(sy => sy.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Name }).IsUnique();

entity.HasIndex(e => new { e.SchoolId, e.IsCurrent });

});

modelBuilder.Entity<Term>(entity =>

{

entity.HasOne(t => t.SchoolYear)

.WithMany(sy => sy.Terms)

.HasForeignKey(t => t.SchoolYearId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolYearId, e.TermNumber }).IsUnique();

});

}

private void ConfigureGradingEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradingScheme>(entity =>

{

entity.HasIndex(e => new { e.Name, e.Country }).IsUnique();

entity.Property(e => e.Name).IsRequired();

});

modelBuilder.Entity<GradingScale>(entity =>

{

entity.HasOne(gs => gs.GradingScheme)

.WithMany(g => g.GradingScales)

.HasForeignKey(gs => gs.GradingSchemeId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.GradingSchemeId, e.Symbol }).IsUnique();

entity.HasIndex(e => new { e.GradingSchemeId, e.Unit }).IsUnique();

entity.Property(e => e.MinPercentage).HasColumnType("decimal(5,2)");

entity.Property(e => e.MaxPercentage).HasColumnType("decimal(5,2)");

});

modelBuilder.Entity<School>()

.HasOne(s => s.GradingScheme)

.WithMany(gs => gs.Schools)

.HasForeignKey(s => s.GradingSchemeId)

.OnDelete(DeleteBehavior.SetNull);

}

private void ConfigureUserEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Teacher>(entity =>

{

entity.HasOne(t => t.School)

.WithMany(s => s.Teachers)

.HasForeignKey(t => t.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Email }).IsUnique();

entity.HasIndex(e => new { e.SchoolId, e.EmployeeId }).IsUnique();

entity.Property(e => e.FirstName).IsRequired();

entity.Property(e => e.LastName).IsRequired();

});

modelBuilder.Entity<Student>(entity =>

{

entity.HasOne(s => s.School)

.WithMany(sch => sch.Students)

.HasForeignKey(s => s.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(s => s.CurrentGrade)

.WithMany(g => g.Students)

.HasForeignKey(s => s.CurrentGradeId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.SchoolId, e.StudentNumber }).IsUnique();

entity.Property(e => e.FirstName).IsRequired();

entity.Property(e => e.LastName).IsRequired();

// Many-to-many relationship with Parents

entity.HasMany(s => s.Parents)

.WithMany(p => p.Students)

.UsingEntity(j => j.ToTable("StudentParents"));

});

modelBuilder.Entity<Parent>(entity =>

{

entity.HasIndex(e => e.Email).IsUnique();

entity.Property(e => e.FirstName).IsRequired();

entity.Property(e => e.LastName).IsRequired();

});

}

private void ConfigureAcademicEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Grade>(entity =>

{

entity.HasOne(g => g.School)

.WithMany(s => s.Grades)

.HasForeignKey(g => g.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Name }).IsUnique();

entity.HasIndex(e => new { e.SchoolId, e.Level });

entity.Property(e => e.Name).IsRequired();

});

modelBuilder.Entity<Subject>(entity =>

{

entity.HasOne(s => s.School)

.WithMany(sch => sch.Subjects)

.HasForeignKey(s => s.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.SchoolId, e.Code }).IsUnique();

entity.Property(e => e.Name).IsRequired();

});

// Configure relationship entities

ConfigureRelationshipEntities(modelBuilder);

// Configure attendance

modelBuilder.Entity<Attendance>(entity =>

{

entity.HasOne(a => a.Student)

.WithMany(s => s.AttendanceRecords)

.HasForeignKey(a => a.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(a => a.Subject)

.WithMany()

.HasForeignKey(a => a.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(a => a.Teacher)

.WithMany(t => t.AttendanceRecords)

.HasForeignKey(a => a.TeacherId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(a => a.Term)

.WithMany(t => t.AttendanceRecords)

.HasForeignKey(a => a.TermId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.StudentId, e.SubjectId, e.Date }).IsUnique();

});

// Configure results

modelBuilder.Entity<Result>(entity =>

{

entity.HasOne(r => r.Student)

.WithMany(s => s.Results)

.HasForeignKey(r => r.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(r => r.Subject)

.WithMany(s => s.Results)

.HasForeignKey(r => r.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(r => r.Teacher)

.WithMany(t => t.Results)

.HasForeignKey(r => r.TeacherId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(r => r.Term)

.WithMany(t => t.Results)

.HasForeignKey(r => r.TermId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(r => r.SchoolYear)

.WithMany(sy => sy.Results)

.HasForeignKey(r => r.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.Property(e => e.Score).HasColumnType("decimal(6,2)");

entity.Property(e => e.TotalMarks).HasColumnType("decimal(6,2)");

entity.Property(e => e.Percentage).HasColumnType("decimal(5,2)");

entity.Property(e => e.AiConfidenceScore).HasColumnType("decimal(5,4)");

entity.Property(e => e.PartialCreditScore).HasColumnType("decimal(5,2)");

});

}

private void ConfigureRelationshipEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<GradeSubject>(entity =>

{

entity.HasOne(gs => gs.Grade)

.WithMany(g => g.GradeSubjects)

.HasForeignKey(gs => gs.GradeId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(gs => gs.Subject)

.WithMany(s => s.GradeSubjects)

.HasForeignKey(gs => gs.SubjectId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.GradeId, e.SubjectId }).IsUnique();

});

modelBuilder.Entity<SubjectTeacher>(entity =>

{

entity.HasOne(st => st.Subject)

.WithMany(s => s.SubjectTeachers)

.HasForeignKey(st => st.SubjectId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(st => st.Teacher)

.WithMany(t => t.SubjectTeachers)

.HasForeignKey(st => st.TeacherId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(st => st.Grade)

.WithMany()

.HasForeignKey(st => st.GradeId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(st => st.SchoolYear)

.WithMany(sy => sy.SubjectTeachers)

.HasForeignKey(st => st.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.SubjectId, e.TeacherId, e.GradeId, e.SchoolYearId }).IsUnique();

});

modelBuilder.Entity<StudentSubject>(entity =>

{

entity.HasOne(ss => ss.Student)

.WithMany(s => s.StudentSubjects)

.HasForeignKey(ss => ss.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(ss => ss.Subject)

.WithMany(s => s.StudentSubjects)

.HasForeignKey(ss => ss.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(ss => ss.SchoolYear)

.WithMany(sy => sy.StudentSubjects)

.HasForeignKey(ss => ss.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.StudentId, e.SubjectId, e.SchoolYearId }).IsUnique();

});

modelBuilder.Entity<ClassTeacher>(entity =>

{

entity.HasOne(ct => ct.Teacher)

.WithMany(t => t.ClassTeachers)

.HasForeignKey(ct => ct.TeacherId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(ct => ct.Grade)

.WithMany(g => g.ClassTeachers)

.HasForeignKey(ct => ct.GradeId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(ct => ct.SchoolYear)

.WithMany(sy => sy.ClassTeachers)

.HasForeignKey(ct => ct.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.GradeId, e.SchoolYearId }).IsUnique();

});

}

private void ConfigureTimetableEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Timetable>(entity =>

{

entity.HasOne(t => t.School)

.WithMany(s => s.Timetables)

.HasForeignKey(t => t.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(t => t.Grade)

.WithMany(g => g.Timetables)

.HasForeignKey(t => t.GradeId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(t => t.SchoolYear)

.WithMany()

.HasForeignKey(t => t.SchoolYearId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(t => t.Term)

.WithMany()

.HasForeignKey(t => t.TermId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.GradeId, e.SchoolYearId, e.TermId }).IsUnique();

});

modelBuilder.Entity<TimetableSlot>(entity =>

{

entity.HasOne(ts => ts.Timetable)

.WithMany(t => t.TimetableSlots)

.HasForeignKey(ts => ts.TimetableId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(ts => ts.Subject)

.WithMany(s => s.TimetableSlots)

.HasForeignKey(ts => ts.SubjectId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasOne(ts => ts.Teacher)

.WithMany(t => t.TimetableSlots)

.HasForeignKey(ts => ts.TeacherId)

.OnDelete(DeleteBehavior.Restrict);

entity.HasIndex(e => new { e.TimetableId, e.DayOfWeek, e.PeriodNumber }).IsUnique();

entity.HasIndex(e => new { e.TeacherId, e.DayOfWeek, e.StartTime, e.EndTime });

});

modelBuilder.Entity<StudentTimetable>(entity =>

{

entity.HasOne(st => st.Student)

.WithMany(s => s.StudentTimetables)

.HasForeignKey(st => st.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(st => st.TimetableSlot)

.WithMany()

.HasForeignKey(st => st.TimetableSlotId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.StudentId, e.TimetableSlotId }).IsUnique();

});

}

private void ConfigureAIEntities(ModelBuilder modelBuilder)

{

modelBuilder.Entity<HandwritingTrainingData>(entity =>

{

entity.HasOne(htd => htd.Student)

.WithMany(s => s.HandwritingTrainingData)

.HasForeignKey(htd => htd.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(htd => htd.Teacher)

.WithMany()

.HasForeignKey(htd => htd.TeacherId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasIndex(e => new { e.StudentId, e.Status });

entity.HasIndex(e => e.IsVerified);

entity.Property(e => e.ConfidenceScore).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<HandwritingCharacterData>(entity =>

{

entity.HasOne(hcd => hcd.HandwritingTrainingData)

.WithMany(htd => htd.CharacterData)

.HasForeignKey(hcd => hcd.HandwritingTrainingDataId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.HandwritingTrainingDataId, e.SequenceOrder });

entity.Property(e => e.ConfidenceScore).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<HandwritingModel>(entity =>

{

entity.HasOne(hm => hm.Student)

.WithMany()

.HasForeignKey(hm => hm.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.StudentId, e.Version }).IsUnique();

entity.HasIndex(e => new { e.StudentId, e.IsActive });

entity.Property(e => e.Accuracy).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<HandwritingRecognitionResult>(entity =>

{

entity.HasOne(hrr => hrr.Student)

.WithMany()

.HasForeignKey(hrr => hrr.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(hrr => hrr.Subject)

.WithMany()

.HasForeignKey(hrr => hrr.SubjectId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasOne(hrr => hrr.Result)

.WithMany()

.HasForeignKey(hrr => hrr.ResultId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasOne(hrr => hrr.HandwritingModel)

.WithMany(hm => hm.RecognitionResults)

.HasForeignKey(hrr => hrr.HandwritingModelId)

.OnDelete(DeleteBehavior.Restrict);

entity.Property(e => e.OverallConfidence).HasColumnType("decimal(5,4)");

entity.Property(e => e.PartialCreditScore).HasColumnType("decimal(5,2)");

});

modelBuilder.Entity<TrainingExercise>(entity =>

{

entity.HasOne(te => te.School)

.WithMany()

.HasForeignKey(te => te.SchoolId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(te => te.Grade)

.WithMany()

.HasForeignKey(te => te.GradeId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasOne(te => te.Subject)

.WithMany()

.HasForeignKey(te => te.SubjectId)

.OnDelete(DeleteBehavior.SetNull);

entity.HasIndex(e => new { e.SchoolId, e.Type, e.Difficulty });

});

modelBuilder.Entity<StudentTrainingSession>(entity =>

{

entity.HasOne(sts => sts.Student)

.WithMany()

.HasForeignKey(sts => sts.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasOne(sts => sts.TrainingExercise)

.WithMany(te => te.StudentSessions)

.HasForeignKey(sts => sts.TrainingExerciseId)

.OnDelete(DeleteBehavior.Cascade);

entity.Property(e => e.OverallAccuracy).HasColumnType("decimal(5,4)");

});

modelBuilder.Entity<ModelTrainingJob>(entity =>

{

entity.HasOne(mtj => mtj.Student)

.WithMany()

.HasForeignKey(mtj => mtj.StudentId)

.OnDelete(DeleteBehavior.Cascade);

entity.HasIndex(e => new { e.Status, e.QueuedAt });

entity.Property(e => e.Progress).HasColumnType("decimal(5,2)");

});

}

private void ConfigureIndexes(ModelBuilder modelBuilder)

{

// Additional performance indexes

modelBuilder.Entity<Result>()

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

modelBuilder.Entity<Attendance>()

.HasIndex(e => new { e.Date, e.Status });

modelBuilder.Entity<Notification>()

.HasIndex(e => new { e.Type, e.IsSent, e.CreatedAt });

}

private void ConfigureSoftDelete(ModelBuilder modelBuilder)

{

// Configure soft delete for all entities inheriting from BaseEntity

foreach (var entityType in modelBuilder.Model.GetEntityTypes())

{

if (typeof(BaseEntity).IsAssignableFrom(entityType.ClrType))

{

modelBuilder.Entity(entityType.ClrType)

.HasQueryFilter(e => !((BaseEntity)e).IsDeleted);

}

}

}

public override int SaveChanges()

{

UpdateTimestamps();

return base.SaveChanges();

}

public override Task<int> SaveChangesAsync(CancellationToken cancellationToken = default)

{

UpdateTimestamps();

return base.SaveChangesAsync(cancellationToken);

}

private void UpdateTimestamps()

{

var entries = ChangeTracker.Entries<BaseEntity>();

foreach (var entry in entries)

{

switch (entry.State)

{

case EntityState.Added:

entry.Entity.CreatedAt = DateTime.UtcNow;

entry.Entity.UpdatedAt = DateTime.UtcNow;

break;

case EntityState.Modified:

entry.Entity.UpdatedAt = DateTime.UtcNow;

break;

}

}

}

}

}// Services/Interfaces/ISchoolService.cs

using SchoolManagement.Core.DTOs;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Services.Interfaces

{

public interface ISchoolService

{

Task<IEnumerable<SchoolDto>> GetAllSchoolsAsync();

Task<SchoolDto> GetSchoolByIdAsync(Guid schoolId);

Task<SchoolDto> CreateSchoolAsync(CreateSchoolDto createSchoolDto);

Task<SchoolDto> UpdateSchoolAsync(Guid schoolId, UpdateSchoolDto updateSchoolDto);

Task<bool> DeleteSchoolAsync(Guid schoolId);

Task<IEnumerable<SchoolDto>> GetSchoolsByGradingSchemeAsync(Guid gradingSchemeId);

}

}

// Services/Interfaces/IStudentService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IStudentService

{

Task<IEnumerable<StudentDto>> GetStudentsBySchoolAsync(Guid schoolId);

Task<IEnumerable<StudentDto>> GetStudentsByGradeAsync(Guid gradeId);

Task<StudentDto> GetStudentByIdAsync(Guid studentId);

Task<StudentDto> CreateStudentAsync(CreateStudentDto createStudentDto);

Task<StudentDto> UpdateStudentAsync(Guid studentId, UpdateStudentDto updateStudentDto);

Task<bool> DeleteStudentAsync(Guid studentId);

Task<bool> AssignStudentToSubjectsAsync(Guid studentId, List<Guid> subjectIds);

Task<IEnumerable<SubjectDto>> GetStudentSubjectsAsync(Guid studentId, Guid schoolYearId);

Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId);

}

}

// Services/Interfaces/ITeacherService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface ITeacherService

{

Task<IEnumerable<TeacherDto>> GetTeachersBySchoolAsync(Guid schoolId);

Task<TeacherDto> GetTeacherByIdAsync(Guid teacherId);

Task<TeacherDto> CreateTeacherAsync(CreateTeacherDto createTeacherDto);

Task<TeacherDto> UpdateTeacherAsync(Guid teacherId, UpdateTeacherDto updateTeacherDto);

Task<bool> DeleteTeacherAsync(Guid teacherId);

Task<bool> AssignTeacherToSubjectAsync(AssignTeacherSubjectDto assignDto);

Task<bool> AssignClassTeacherAsync(AssignClassTeacherDto assignDto);

Task<IEnumerable<SubjectDto>> GetTeacherSubjectsAsync(Guid teacherId, Guid schoolYearId);

Task<IEnumerable<GradeDto>> GetTeacherClassesAsync(Guid teacherId, Guid schoolYearId);

}

}

// Services/Interfaces/IAttendanceService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IAttendanceService

{

Task<AttendanceDto> MarkAttendanceAsync(CreateAttendanceDto createAttendanceDto);

Task<IEnumerable<AttendanceDto>> GetAttendanceByClassAsync(Guid gradeId, DateTime date);

Task<IEnumerable<AttendanceDto>> GetStudentAttendanceAsync(Guid studentId, Guid termId);

Task<AttendanceReportDto> GetAttendanceReportAsync(Guid studentId, Guid termId);

Task<bool> BulkMarkAttendanceAsync(List<CreateAttendanceDto> attendanceList);

Task<AttendanceDto> UpdateAttendanceAsync(Guid attendanceId, UpdateAttendanceDto updateDto);

}

}

// Services/Interfaces/IResultService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IResultService

{

Task<ResultDto> CreateResultAsync(CreateResultDto createResultDto);

Task<ResultDto> UpdateResultAsync(Guid resultId, UpdateResultDto updateResultDto);

Task<IEnumerable<ResultDto>> GetStudentResultsAsync(Guid studentId, Guid termId);

Task<IEnumerable<ResultDto>> GetClassResultsAsync(Guid gradeId, Guid subjectId, Guid termId);

Task<StudentReportCardDto> GenerateReportCardAsync(Guid studentId, Guid termId);

Task<bool> UploadResultsAsync(Guid teacherId, List<CreateResultDto> results);

Task<ClassResultSummaryDto> GetClassResultSummaryAsync(Guid gradeId, Guid termId);

Task<byte[]> PrintTermResultsAsync(Guid studentId, Guid termId);

Task<byte[]> PrintClassResultsAsync(Guid gradeId, Guid termId);

}

}

// Services/Interfaces/IGradingService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IGradingService

{

Task<IEnumerable<GradingSchemeDto>> GetAllGradingSchemesAsync();

Task<GradingSchemeDto> CreateGradingSchemeAsync(CreateGradingSchemeDto createDto);

Task<GradingSchemeDto> UpdateGradingSchemeAsync(Guid schemeId, UpdateGradingSchemeDto updateDto);

Task<bool> DeleteGradingSchemeAsync(Guid schemeId);

Task<GradingScaleDto> AddGradingScaleAsync(Guid schemeId, CreateGradingScaleDto createDto);

Task<GradingScaleDto> UpdateGradingScaleAsync(Guid scaleId, UpdateGradingScaleDto updateDto);

Task<bool> DeleteGradingScaleAsync(Guid scaleId);

Task<GradeCalculationDto> CalculateGradeAsync(Guid gradingSchemeId, decimal percentage);

Task<IEnumerable<GradingScaleDto>> GetGradingScalesAsync(Guid schemeId);

}

}

// Services/Interfaces/ITimetableService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface ITimetableService

{

Task<TimetableDto> GenerateTimetableAsync(GenerateTimetableDto generateDto);

Task<TimetableDto> GetGradeTimetableAsync(Guid gradeId, Guid termId);

Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId);

Task<TeacherTimetableDto> GetTeacherTimetableAsync(Guid teacherId, Guid termId);

Task<TimetableSlotDto> UpdateTimetableSlotAsync(Guid slotId, UpdateTimetableSlotDto updateDto);

Task<bool> ValidateTimetableAsync(Guid timetableId);

Task<TimetableConflictDto> CheckTimetableConflictsAsync(Guid timetableId);

Task<byte[]> PrintTimetableAsync(Guid timetableId);

}

}

// Services/Interfaces/INotificationService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface INotificationService

{

Task<NotificationDto> CreateNotificationAsync(CreateNotificationDto createDto);

Task<bool> SendNotificationAsync(Guid notificationId);

Task<bool> SendBulkNotificationAsync(List<Guid> recipientIds, CreateNotificationDto notificationDto);

Task<IEnumerable<NotificationDto>> GetUserNotificationsAsync(Guid userId, string userType);

Task<bool> MarkAsReadAsync(Guid notificationId);

Task<bool> SendWhatsAppMessageAsync(string phoneNumber, string message);

Task<bool> SendEmailNotificationAsync(string email, string subject, string message);

Task<NotificationStatsDto> GetNotificationStatsAsync(Guid schoolId);

}

}

// Services/Interfaces/IHandwritingRecognitionService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IHandwritingRecognitionService

{

Task<HandwritingRecognitionResultDto> ProcessHandwritingAsync(ProcessHandwritingDto processDto);

Task<TrainingExerciseDto> GenerateTrainingExerciseAsync(GenerateTrainingExerciseDto generateDto);

Task<StudentTrainingSessionDto> StartTrainingSessionAsync(Guid studentId, Guid exerciseId);

Task<StudentTrainingSessionDto> SubmitTrainingDataAsync(SubmitTrainingDataDto submitDto);

Task<bool> VerifyTrainingDataAsync(Guid trainingDataId, string verifiedText);

Task<ModelTrainingJobDto> StartModelTrainingAsync(Guid studentId);

Task<ModelTrainingJobDto> GetTrainingJobStatusAsync(Guid jobId);

Task<HandwritingModelDto> GetStudentModelAsync(Guid studentId);

Task<bool> DeployModelLocallyAsync(Guid modelId);

Task<TrainingProgressDto> GetStudentTrainingProgressAsync(Guid studentId);

Task<IEnumerable<HandwritingTrainingDataDto>> GetUnverifiedTrainingDataAsync(Guid teacherId);

}

}

// Services/Implementations/SchoolService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class SchoolService : ISchoolService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

public SchoolService(SchoolManagementDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

public async Task<IEnumerable<SchoolDto>> GetAllSchoolsAsync()

{

var schools = await \_context.Schools

.Include(s => s.GradingScheme)

.ToListAsync();

return \_mapper.Map<IEnumerable<SchoolDto>>(schools);

}

public async Task<SchoolDto> GetSchoolByIdAsync(Guid schoolId)

{

var school = await \_context.Schools

.Include(s => s.GradingScheme)

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

if (school == null)

throw new NotFoundException($"School with ID {schoolId} not found");

return \_mapper.Map<SchoolDto>(school);

}

public async Task<SchoolDto> CreateSchoolAsync(CreateSchoolDto createSchoolDto)

{

var school = \_mapper.Map<School>(createSchoolDto);

\_context.Schools.Add(school);

await \_context.SaveChangesAsync();

return \_mapper.Map<SchoolDto>(school);

}

public async Task<SchoolDto> UpdateSchoolAsync(Guid schoolId, UpdateSchoolDto updateSchoolDto)

{

var school = await \_context.Schools.FindAsync(schoolId);

if (school == null)

throw new NotFoundException($"School with ID {schoolId} not found");

\_mapper.Map(updateSchoolDto, school);

await \_context.SaveChangesAsync();

return \_mapper.Map<SchoolDto>(school);

}

public async Task<bool> DeleteSchoolAsync(Guid schoolId)

{

var school = await \_context.Schools.FindAsync(schoolId);

if (school == null)

return false;

school.IsDeleted = true;

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<SchoolDto>> GetSchoolsByGradingSchemeAsync(Guid gradingSchemeId)

{

var schools = await \_context.Schools

.Where(s => s.GradingSchemeId == gradingSchemeId)

.Include(s => s.GradingScheme)

.ToListAsync();

return \_mapper.Map<IEnumerable<SchoolDto>>(schools);

}

}

}

// Services/Implementations/StudentService.cs

namespace SchoolManagement.Services.Implementations

{

public class StudentService : IStudentService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

public StudentService(SchoolManagementDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

public async Task<IEnumerable<StudentDto>> GetStudentsBySchoolAsync(Guid schoolId)

{

var students = await \_context.Students

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

.Include(s => s.CurrentGrade)

.Include(s => s.Parents)

.ToListAsync();

return \_mapper.Map<IEnumerable<StudentDto>>(students);

}

public async Task<IEnumerable<StudentDto>> GetStudentsByGradeAsync(Guid gradeId)

{

var students = await \_context.Students

.Where(s => s.CurrentGradeId == gradeId)

.Include(s => s.CurrentGrade)

.Include(s => s.Parents)

.ToListAsync();

return \_mapper.Map<IEnumerable<StudentDto>>(students);

}

public async Task<StudentDto> GetStudentByIdAsync(Guid studentId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.Parents)

.Include(s => s.School)

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

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

return \_mapper.Map<StudentDto>(student);

}

public async Task<StudentDto> CreateStudentAsync(CreateStudentDto createStudentDto)

{

var student = \_mapper.Map<Student>(createStudentDto);

// Generate student number if not provided

if (string.IsNullOrEmpty(student.StudentNumber))

{

student.StudentNumber = await GenerateStudentNumberAsync(student.SchoolId);

}

\_context.Students.Add(student);

await \_context.SaveChangesAsync();

return \_mapper.Map<StudentDto>(student);

}

public async Task<StudentDto> UpdateStudentAsync(Guid studentId, UpdateStudentDto updateStudentDto)

{

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

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

\_mapper.Map(updateStudentDto, student);

await \_context.SaveChangesAsync();

return \_mapper.Map<StudentDto>(student);

}

public async Task<bool> DeleteStudentAsync(Guid studentId)

{

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

if (student == null)

return false;

student.IsDeleted = true;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> AssignStudentToSubjectsAsync(Guid studentId, List<Guid> subjectIds)

{

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

if (student == null)

return false;

var currentSchoolYear = await \_context.SchoolYears

.Where(sy => sy.SchoolId == student.SchoolId && sy.IsCurrent)

.FirstOrDefaultAsync();

if (currentSchoolYear == null)

return false;

// Remove existing assignments for current school year

var existingAssignments = await \_context.StudentSubjects

.Where(ss => ss.StudentId == studentId && ss.SchoolYearId == currentSchoolYear.Id)

.ToListAsync();

\_context.StudentSubjects.RemoveRange(existingAssignments);

// Add new assignments

var newAssignments = subjectIds.Select(subjectId => new StudentSubject

{

StudentId = studentId,

SubjectId = subjectId,

SchoolYearId = currentSchoolYear.Id,

EnrollmentDate = DateTime.UtcNow

});

\_context.StudentSubjects.AddRange(newAssignments);

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<SubjectDto>> GetStudentSubjectsAsync(Guid studentId, Guid schoolYearId)

{

var subjects = await \_context.StudentSubjects

.Where(ss => ss.StudentId == studentId && ss.SchoolYearId == schoolYearId && ss.IsActive)

.Include(ss => ss.Subject)

.Select(ss => ss.Subject)

.ToListAsync();

return \_mapper.Map<IEnumerable<SubjectDto>>(subjects);

}

public async Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

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

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

var currentTerm = await \_context.Terms

.Where(t => t.SchoolYear.SchoolId == student.SchoolId && t.IsCurrent)

.FirstOrDefaultAsync();

if (currentTerm == null)

throw new NotFoundException("No current term found");

var timetable = await \_context.Timetables

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Subject)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Teacher)

.FirstOrDefaultAsync(t => t.GradeId == student.CurrentGradeId && t.TermId == currentTerm.Id);

return \_mapper.Map<StudentTimetableDto>(timetable);

}

private async Task<string> GenerateStudentNumberAsync(Guid schoolId)

{

var year = DateTime.Now.Year;

var lastStudent = await \_context.Students

.Where(s => s.SchoolId == schoolId && s.StudentNumber.StartsWith(year.ToString()))

.OrderByDescending(s => s.StudentNumber)

.FirstOrDefaultAsync();

var sequence = 1;

if (lastStudent != null && int.TryParse(lastStudent.StudentNumber.Substring(4), out var lastSequence))

{

sequence = lastSequence + 1;

}

return $"{year}{sequence:D4}";

}

}

}

// Services/Implementations/ResultService.cs

namespace SchoolManagement.Services.Implementations

{

public class ResultService : IResultService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly IGradingService \_gradingService;

private readonly IHandwritingRecognitionService \_handwritingService;

public ResultService(

SchoolManagementDbContext context,

IMapper mapper,

IGradingService gradingService,

IHandwritingRecognitionService handwritingService)

{

\_context = context;

\_mapper = mapper;

\_gradingService = gradingService;

\_handwritingService = handwritingService;

}

public async Task<ResultDto> CreateResultAsync(CreateResultDto createResultDto)

{

var result = \_mapper.Map<r>(createResultDto);

// Calculate percentage

result.Percentage = (result.Score / result.TotalMarks) \* 100;

// Get student's school grading scheme

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

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

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

result.Percentage);

result.Grade = gradeCalculation.Symbol;

result.GradeUnit = gradeCalculation.Unit;

}

\_context.Results.Add(result);

await \_context.SaveChangesAsync();

return \_mapper.Map<ResultDto>(result);

}

public async Task<ResultDto> UpdateResultAsync(Guid resultId, UpdateResultDto updateResultDto)

{

var result = await \_context.Results.FindAsync(resultId);

if (result == null)

throw new NotFoundException($"Result with ID {resultId} not found");

\_mapper.Map(updateResultDto, result);

// Recalculate grade if score changed

if (updateResultDto.Score.HasValue || updateResultDto.TotalMarks.HasValue)

{

result.Percentage = (result.Score / result.TotalMarks) \* 100;

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

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

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

result.Percentage);

result.Grade = gradeCalculation.Symbol;

result.GradeUnit = gradeCalculation.Unit;

}

}

await \_context.SaveChangesAsync();

return \_mapper.Map<ResultDto>(result);

}

public async Task<IEnumerable<ResultDto>> GetStudentResultsAsync(Guid studentId, Guid termId)

{

var results = await \_context.Results

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

.Include(r => r.Subject)

.Include(r => r.Teacher)

.ToListAsync();

return \_mapper.Map<IEnumerable<ResultDto>>(results);

}

public async Task<IEnumerable<ResultDto>> GetClassResultsAsync(Guid gradeId, Guid subjectId, Guid termId)

{

var results = await \_context.Results

.Where(r => r.Student.CurrentGradeId == gradeId &&

r.SubjectId == subjectId &&

r.TermId == termId)

.Include(r => r.Student)

.Include(r => r.Subject)

.ToListAsync();

return \_mapper.Map<IEnumerable<ResultDto>>(results);

}

public async Task<StudentReportCardDto> GenerateReportCardAsync(Guid studentId, Guid termId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.School)

.Include(s => s.Parents)

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

if (student == null)

throw new NotFoundException($"Student with ID {studentId} not found");

var term = await \_context.Terms

.Include(t => t.SchoolYear)

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

var results = await GetStudentResultsAsync(studentId, termId);

var attendance = await \_context.Attendances

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

.GroupBy(a => a.SubjectId)

.Select(g => new

{

SubjectId = g.Key,

TotalClasses = g.Count(),

PresentClasses = g.Count(a => a.Status == AttendanceStatus.Present)

})

.ToListAsync();

var reportCard = new StudentReportCardDto

{

Student = \_mapper.Map<StudentDto>(student),

Term = \_mapper.Map<TermDto>(term),

Results = results,

AttendanceSummary = attendance.Select(a => new AttendanceSummaryDto

{

SubjectId = a.SubjectId,

TotalClasses = a.TotalClasses,

PresentClasses = a.PresentClasses,

AttendancePercentage = (decimal)a.PresentClasses / a.TotalClasses \* 100

}).ToList(),

GeneratedAt = DateTime.UtcNow

};

return reportCard;

}

public async Task<bool> UploadResultsAsync(Guid teacherId, List<CreateResultDto> results)

{

var teacher = await \_context.Teachers.FindAsync(teacherId);

if (teacher == null)

return false;

var resultEntities = new List<r>();

foreach (var createResultDto in results)

{

var result = \_mapper.Map<r>(createResultDto);

result.TeacherId = teacherId;

result.Percentage = (result.Score / result.TotalMarks) \* 100;

// Calculate grade

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

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

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

result.Percentage);

result.Grade = gradeCalculation.Symbol;

result.GradeUnit = gradeCalculation.Unit;

}

resultEntities.Add(result);

}

\_context.Results.AddRange(resultEntities);

await \_context.SaveChangesAsync();

return true;

}

public async Task<ClassResultSummaryDto> GetClassResultSummaryAsync(Guid gradeId, Guid termId)

{

var results = await \_context.Results

.Where(r => r.Student.CurrentGradeId == gradeId && r.TermId == termId)

.Include(r => r.Subject)

.Include(r => r.Student)

.ToListAsync();

var subjectSummaries = results

.GroupBy(r => r.Subject)

.Select(g => new SubjectResultSummaryDto

{

Subject = \_mapper.Map<SubjectDto>(g.Key),

TotalStudents = g.Count(),

AverageScore = g.Average(r => r.Percentage),

HighestScore = g.Max(r => r.Percentage),

LowestScore = g.Min(r => r.Percentage),

PassCount = g.Count(r => r.Percentage >= 50), // Assuming 50% is pass mark

FailCount = g.Count(r => r.Percentage < 50)

}).ToList();

var grade = await \_context.Grades.FindAsync(gradeId);

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

return new ClassResultSummaryDto

{

Grade = \_mapper.Map<GradeDto>(grade),

Term = \_mapper.Map<TermDto>(term),

SubjectSummaries = subjectSummaries,

TotalStudents = results.Select(r => r.StudentId).Distinct().Count(),

OverallAverage = results.Average(r => r.Percentage)

};

}

public async Task<byte[]> PrintTermResultsAsync(Guid studentId, Guid termId)

{

var reportCard = await GenerateReportCardAsync(studentId, termId);

// Implementation would generate PDF using a library like iTextSharp or similar

// This is a placeholder for PDF generation logic

var pdfService = new PdfGenerationService();

return await pdfService.GenerateReportCardPdfAsync(reportCard);

}

public async Task<byte[]> PrintClassResultsAsync(Guid gradeId, Guid termId)

{

var classSummary = await GetClassResultSummaryAsync(gradeId, termId);

// Implementation would generate PDF

var pdfService = new PdfGenerationService();

return await pdfService.GenerateClassResultsPdfAsync(classSummary);

}

}

}// Services/Implementations/HandwritingRecognitionService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Services.Interfaces;

using System.Drawing;

using System.Drawing.Imaging;

namespace SchoolManagement.Services.Implementations

{

public class HandwritingRecognitionService : IHandwritingRecognitionService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly IImageProcessingService \_imageProcessingService;

private readonly IMachineLearningService \_mlService;

private readonly IFileStorageService \_fileStorageService;

private readonly ILogger<HandwritingRecognitionService> \_logger;

public HandwritingRecognitionService(

SchoolManagementDbContext context,

IMapper mapper,

IImageProcessingService imageProcessingService,

IMachineLearningService mlService,

IFileStorageService fileStorageService,

ILogger<HandwritingRecognitionService> logger)

{

\_context = context;

\_mapper = mapper;

\_imageProcessingService = imageProcessingService;

\_mlService = mlService;

\_fileStorageService = fileStorageService;

\_logger = logger;

}

public async Task<HandwritingRecognitionResultDto> ProcessHandwritingAsync(ProcessHandwritingDto processDto)

{

try

{

var startTime = DateTime.UtcNow;

// Get student's personal model if available

var personalModel = await GetActiveModelForStudentAsync(processDto.StudentId);

// Process image and extract text

var imageProcessingResult = await \_imageProcessingService.ProcessImageAsync(processDto.ImagePath);

string recognizedText;

decimal confidence;

bool processedLocally = false;

if (personalModel != null && personalModel.IsDeployedLocally)

{

// Use local personal model for recognition

var localResult = await \_mlService.RecognizeTextLocallyAsync(

personalModel.ModelFilePath,

processDto.ImagePath);

recognizedText = localResult.Text;

confidence = localResult.Confidence;

processedLocally = true;

}

else

{

// Fallback to cloud-based recognition

var cloudResult = await \_mlService.RecognizeTextInCloudAsync(processDto.ImagePath);

recognizedText = cloudResult.Text;

confidence = cloudResult.Confidence;

processedLocally = false;

}

var processingTime = DateTime.UtcNow - startTime;

// Create recognition result record

var recognitionResult = new HandwritingRecognitionResult

{

StudentId = processDto.StudentId,

SubjectId = processDto.SubjectId,

ResultId = processDto.ResultId,

HandwritingModelId = personalModel?.Id ?? Guid.Empty,

OriginalImagePath = processDto.ImagePath,

RecognizedText = recognizedText,

OverallConfidence = confidence,

ProcessedLocally = processedLocally,

ProcessedInCloud = !processedLocally,

ProcessingTime = processingTime,

CharacterConfidences = imageProcessingResult.CharacterConfidences,

AlternativeTexts = imageProcessingResult.AlternativeTexts,

ExpectedAnswer = processDto.ExpectedAnswer ?? "",

ProcessedAt = DateTime.UtcNow

};

// Calculate correctness if expected answer provided

if (!string.IsNullOrEmpty(processDto.ExpectedAnswer))

{

var answerComparison = await CompareAnswersAsync(recognizedText, processDto.ExpectedAnswer);

recognitionResult.IsCorrect = answerComparison.IsCorrect;

recognitionResult.PartialCreditScore = answerComparison.PartialCredit;

}

\_context.HandwritingRecognitionResults.Add(recognitionResult);

await \_context.SaveChangesAsync();

return \_mapper.Map<HandwritingRecognitionResultDto>(recognitionResult);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error processing handwriting for student {StudentId}", processDto.StudentId);

throw new HandwritingProcessingException("Failed to process handwriting", ex);

}

}

public async Task<TrainingExerciseDto> GenerateTrainingExerciseAsync(GenerateTrainingExerciseDto generateDto)

{

var exercise = new TrainingExercise

{

SchoolId = generateDto.SchoolId,

GradeId = generateDto.GradeId,

SubjectId = generateDto.SubjectId,

Title = generateDto.Title,

Description = generateDto.Description,

Type = generateDto.Type,

Difficulty = generateDto.Difficulty,

IsSystemGenerated = true,

EstimatedMinutes = generateDto.EstimatedMinutes

};

// Generate exercise content based on type and difficulty

var content = await GenerateExerciseContentAsync(generateDto);

exercise.Content = System.Text.Json.JsonSerializer.Serialize(content);

\_context.TrainingExercises.Add(exercise);

await \_context.SaveChangesAsync();

return \_mapper.Map<TrainingExerciseDto>(exercise);

}

public async Task<StudentTrainingSessionDto> StartTrainingSessionAsync(Guid studentId, Guid exerciseId)

{

var exercise = await \_context.TrainingExercises.FindAsync(exerciseId);

if (exercise == null)

throw new NotFoundException($"Training exercise {exerciseId} not found");

var session = new StudentTrainingSession

{

StudentId = studentId,

TrainingExerciseId = exerciseId,

StartedAt = DateTime.UtcNow,

Status = SessionStatus.InProgress

};

// Parse exercise content to get total items

var exerciseContent = System.Text.Json.JsonSerializer.Deserialize<ExerciseContentDto>(exercise.Content);

session.TotalItems = exerciseContent?.Items?.Count ?? 0;

\_context.StudentTrainingSessions.Add(session);

await \_context.SaveChangesAsync();

return \_mapper.Map<StudentTrainingSessionDto>(session);

}

public async Task<StudentTrainingSessionDto> SubmitTrainingDataAsync(SubmitTrainingDataDto submitDto)

{

var session = await \_context.StudentTrainingSessions

.Include(s => s.TrainingExercise)

.FirstOrDefaultAsync(s => s.Id == submitDto.SessionId);

if (session == null)

throw new NotFoundException($"Training session {submitDto.SessionId} not found");

// Store the uploaded image

var imagePath = await \_fileStorageService.SaveImageAsync(

submitDto.HandwritingImage,

$"training/{session.StudentId}/{session.Id}");

// Process the handwritten text

var processedText = await \_imageProcessingService.ExtractTextAsync(imagePath);

// Create training data record

var trainingData = new HandwritingTrainingData

{

StudentId = session.StudentId,

DataType = DetermineDataType(submitDto.OriginalContent),

OriginalContent = submitDto.OriginalContent,

ImagePath = imagePath,

ProcessedText = processedText.Text,

ConfidenceScore = processedText.Confidence,

Status = TrainingStatus.Processing,

CapturedAt = DateTime.UtcNow,

BoundingBoxes = processedText.BoundingBoxes,

ModelVersion = await GetCurrentModelVersionAsync()

};

// Extract individual characters for detailed training

var characterData = await ExtractCharacterDataAsync(trainingData);

trainingData.CharacterData = characterData;

\_context.HandwritingTrainingData.Add(trainingData);

// Update session progress

session.CompletedItems++;

session.OverallAccuracy = CalculateSessionAccuracy(session);

if (session.CompletedItems >= session.TotalItems)

{

session.Status = SessionStatus.Completed;

session.CompletedAt = DateTime.UtcNow;

session.TotalTime = session.CompletedAt.Value - session.StartedAt;

}

await \_context.SaveChangesAsync();

// Queue for verification if confidence is low

if (trainingData.ConfidenceScore < 0.8m) // 80% threshold

{

trainingData.Status = TrainingStatus.NeedsVerification;

await \_context.SaveChangesAsync();

}

return \_mapper.Map<StudentTrainingSessionDto>(session);

}

public async Task<bool> VerifyTrainingDataAsync(Guid trainingDataId, string verifiedText)

{

var trainingData = await \_context.HandwritingTrainingData

.Include(td => td.CharacterData)

.FirstOrDefaultAsync(td => td.Id == trainingDataId);

if (trainingData == null)

return false;

trainingData.VerifiedText = verifiedText;

trainingData.IsVerified = true;

trainingData.VerifiedAt = DateTime.UtcNow;

trainingData.Status = TrainingStatus.Verified;

// Update character-level verification if needed

await UpdateCharacterVerificationAsync(trainingData, verifiedText);

await \_context.SaveChangesAsync();

// Check if student has enough verified data to start training

await CheckAndQueueModelTrainingAsync(trainingData.StudentId);

return true;

}

public async Task<ModelTrainingJobDto> StartModelTrainingAsync(Guid studentId)

{

// Check if student has sufficient training data

var verifiedDataCount = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.IsVerified);

if (verifiedDataCount < 100) // Minimum threshold

{

throw new InvalidOperationException($"Insufficient training data. Need at least 100 verified samples, have {verifiedDataCount}");

}

var job = new ModelTrainingJob

{

StudentId = studentId,

JobName = $"Personal Model Training - {DateTime.UtcNow:yyyy-MM-dd HH:mm}",

Type = JobType.InitialTraining,

Status = JobStatus.Queued,

TotalDataPoints = verifiedDataCount,

TrainingParameters = System.Text.Json.JsonSerializer.Serialize(GetDefaultTrainingParameters())

};

\_context.ModelTrainingJobs.Add(job);

await \_context.SaveChangesAsync();

// Start training process asynchronously

\_ = Task.Run(() => ProcessModelTrainingAsync(job.Id));

return \_mapper.Map<ModelTrainingJobDto>(job);

}

public async Task<ModelTrainingJobDto> GetTrainingJobStatusAsync(Guid jobId)

{

var job = await \_context.ModelTrainingJobs.FindAsync(jobId);

if (job == null)

throw new NotFoundException($"Training job {jobId} not found");

return \_mapper.Map<ModelTrainingJobDto>(job);

}

public async Task<HandwritingModelDto> GetStudentModelAsync(Guid studentId)

{

var model = await \_context.HandwritingModels

.Where(m => m.StudentId == studentId && m.IsActive)

.OrderByDescending(m => m.TrainingCompleted)

.FirstOrDefaultAsync();

if (model == null)

throw new NotFoundException($"No active model found for student {studentId}");

return \_mapper.Map<HandwritingModelDto>(model);

}

public async Task<bool> DeployModelLocallyAsync(Guid modelId)

{

var model = await \_context.HandwritingModels.FindAsync(modelId);

if (model == null)

return false;

try

{

// Download model from cloud storage

var localModelPath = await \_fileStorageService.DownloadModelAsync(model.CloudModelPath, modelId);

// Validate model

var isValid = await \_mlService.ValidateModelAsync(localModelPath);

if (!isValid)

{

\_logger.LogError("Model validation failed for model {ModelId}", modelId);

return false;

}

model.ModelFilePath = localModelPath;

model.IsDeployedLocally = true;

model.LastUsed = DateTime.UtcNow;

await \_context.SaveChangesAsync();

\_logger.LogInformation("Model {ModelId} successfully deployed locally", modelId);

return true;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to deploy model {ModelId} locally", modelId);

return false;

}

}

public async Task<TrainingProgressDto> GetStudentTrainingProgressAsync(Guid studentId)

{

var totalData = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId);

var verifiedData = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.IsVerified);

var pendingVerification = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.Status == TrainingStatus.NeedsVerification);

var currentModel = await \_context.HandwritingModels

.Where(m => m.StudentId == studentId && m.IsActive)

.OrderByDescending(m => m.TrainingCompleted)

.FirstOrDefaultAsync();

var recentSessions = await \_context.StudentTrainingSessions

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

.OrderByDescending(s => s.StartedAt)

.Take(5)

.ToListAsync();

return new TrainingProgressDto

{

StudentId = studentId,

TotalTrainingData = totalData,

VerifiedData = verifiedData,

PendingVerification = pendingVerification,

CurrentModelAccuracy = currentModel?.Accuracy ?? 0,

ModelIsDeployedLocally = currentModel?.IsDeployedLocally ?? false,

RecentSessions = \_mapper.Map<List<StudentTrainingSessionDto>>(recentSessions),

ReadyForTraining = verifiedData >= 100,

CompletionPercentage = Math.Min(100, (verifiedData / 100.0m) \* 100)

};

}

public async Task<IEnumerable<HandwritingTrainingDataDto>> GetUnverifiedTrainingDataAsync(Guid teacherId)

{

var teacher = await \_context.Teachers

.Include(t => t.School)

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

if (teacher == null)

throw new NotFoundException($"Teacher {teacherId} not found");

var unverifiedData = await \_context.HandwritingTrainingData

.Where(td => td.Student.SchoolId == teacher.SchoolId &&

td.Status == TrainingStatus.NeedsVerification)

.Include(td => td.Student)

.OrderBy(td => td.CapturedAt)

.Take(50) // Limit for performance

.ToListAsync();

return \_mapper.Map<IEnumerable<HandwritingTrainingDataDto>>(unverifiedData);

}

// Private helper methods

private async Task<HandwritingModel> GetActiveModelForStudentAsync(Guid studentId)

{

return await \_context.HandwritingModels

.Where(m => m.StudentId == studentId && m.IsActive)

.OrderByDescending(m => m.TrainingCompleted)

.FirstOrDefaultAsync();

}

private async Task<AnswerComparisonResult> CompareAnswersAsync(string recognizedText, string expectedAnswer)

{

// Implement fuzzy matching logic for answer comparison

var similarity = CalculateStringSimilarity(recognizedText.Trim().ToLower(), expectedAnswer.Trim().ToLower());

return new AnswerComparisonResult

{

IsCorrect = similarity >= 0.9m, // 90% similarity threshold

PartialCredit = similarity \* 100,

Similarity = similarity

};

}

private decimal CalculateStringSimilarity(string text1, string text2)

{

if (string.IsNullOrEmpty(text1) || string.IsNullOrEmpty(text2))

return 0;

if (text1 == text2)

return 1;

// Levenshtein distance implementation

var distance = CalculateLevenshteinDistance(text1, text2);

var maxLength = Math.Max(text1.Length, text2.Length);

return 1 - (decimal)distance / maxLength;

}

private int CalculateLevenshteinDistance(string source, string target)

{

if (string.IsNullOrEmpty(source)) return target?.Length ?? 0;

if (string.IsNullOrEmpty(target)) return source.Length;

var matrix = new int[source.Length + 1, target.Length + 1];

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

matrix[i, 0] = i;

for (int j = 0; j <= target.Length; j++)

matrix[0, j] = j;

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

{

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

{

var cost = source[i - 1] == target[j - 1] ? 0 : 1;

matrix[i, j] = Math.Min(

Math.Min(matrix[i - 1, j] + 1, matrix[i, j - 1] + 1),

matrix[i - 1, j - 1] + cost);

}

}

return matrix[source.Length, target.Length];

}

private async Task<ExerciseContentDto> GenerateExerciseContentAsync(GenerateTrainingExerciseDto generateDto)

{

var content = new ExerciseContentDto

{

Items = new List<ExerciseItemDto>()

};

switch (generateDto.Type)

{

case ExerciseType.Alphabet:

content.Items = GenerateAlphabetItems(generateDto.Difficulty);

break;

case ExerciseType.Numbers:

content.Items = GenerateNumberItems(generateDto.Difficulty);

break;

case ExerciseType.Words:

content.Items = await GenerateWordItems(generateDto.Difficulty, generateDto.SubjectId);

break;

case ExerciseType.Sentences:

content.Items = await GenerateSentenceItems(generateDto.Difficulty, generateDto.SubjectId);

break;

case ExerciseType.MathProblems:

content.Items = GenerateMathItems(generateDto.Difficulty);

break;

}

return content;

}

private List<ExerciseItemDto> GenerateAlphabetItems(DifficultyLevel difficulty)

{

var items = new List<ExerciseItemDto>();

var letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

switch (difficulty)

{

case DifficultyLevel.Beginner:

// First 10 letters

for (int i = 0; i < 10; i++)

{

items.Add(new ExerciseItemDto

{

Content = letters[i].ToString(),

ExpectedResponse = letters[i].ToString(),

ItemType = "Letter"

});

}

break;

case DifficultyLevel.Elementary:

// All uppercase letters

foreach (char letter in letters)

{

items.Add(new ExerciseItemDto

{

Content = letter.ToString(),

ExpectedResponse = letter.ToString(),

ItemType = "Letter"

});

}

break;

case DifficultyLevel.Intermediate:

// Mixed case

foreach (char letter in letters)

{

items.Add(new ExerciseItemDto

{

Content = letter.ToString(),

ExpectedResponse = letter.ToString(),

ItemType = "Letter"

});

items.Add(new ExerciseItemDto

{

Content = letter.ToString().ToLower(),

ExpectedResponse = letter.ToString().ToLower(),

ItemType = "Letter"

});

}

break;

}

return items;

}

private List<ExerciseItemDto> GenerateNumberItems(DifficultyLevel difficulty)

{

var items = new List<ExerciseItemDto>();

switch (difficulty)

{

case DifficultyLevel.Beginner:

for (int i = 0; i <= 20; i++)

{

items.Add(new ExerciseItemDto

{

Content = i.ToString(),

ExpectedResponse = i.ToString(),

ItemType = "Number"

});

}

break;

case DifficultyLevel.Elementary:

for (int i = 0; i <= 100; i += 5)

{

items.Add(new ExerciseItemDto

{

Content = i.ToString(),

ExpectedResponse = i.ToString(),

ItemType = "Number"

});

}

break;

case DifficultyLevel.Intermediate:

var random = new Random();

for (int i = 0; i < 50; i++)

{

var number = random.Next(1, 1000);

items.Add(new ExerciseItemDto

{

Content = number.ToString(),

ExpectedResponse = number.ToString(),

ItemType = "Number"

});

}

break;

}

return items;

}

private async Task<List<ExerciseItemDto>> GenerateWordItems(DifficultyLevel difficulty, Guid? subjectId)

{

var items = new List<ExerciseItemDto>();

// This would typically load from a word database

var basicWords = new[] { "cat", "dog", "sun", "book", "tree", "house", "car", "ball" };

var intermediateWords = new[] { "computer", "elephant", "butterfly", "mountain", "rainbow", "chocolate" };

var advancedWords = new[] { "responsibility", "environment", "democracy", "photosynthesis", "constellation" };

string[] wordsToUse = difficulty switch

{

DifficultyLevel.Beginner => basicWords,

DifficultyLevel.Elementary => basicWords.Concat(intermediateWords).ToArray(),

DifficultyLevel.Intermediate => intermediateWords.Concat(advancedWords).ToArray(),

\_ => advancedWords

};

foreach (var word in wordsToUse.Take(20))

{

items.Add(new ExerciseItemDto

{

Content = word,

ExpectedResponse = word,

ItemType = "Word"

});

}

return items;

}

private async Task<List<ExerciseItemDto>> GenerateSentenceItems(DifficultyLevel difficulty, Guid? subjectId)

{

var items = new List<ExerciseItemDto>();

var beginnerSentences = new[]

{

"The cat is sleeping.",

"I like to read books.",

"The sun is bright today."

};

var intermediateSentences = new[]

{

"Students should always complete their homework on time.",

"The weather forecast predicts rain for tomorrow.",

"Science helps us understand the world around us."

};

string[] sentencesToUse = difficulty switch

{

DifficultyLevel.Beginner => beginnerSentences,

\_ => intermediateSentences

};

foreach (var sentence in sentencesToUse)

{

items.Add(new ExerciseItemDto

{

Content = sentence,

ExpectedResponse = sentence,

ItemType = "Sentence"

});

}

return items;

}

private List<ExerciseItemDto> GenerateMathItems(DifficultyLevel difficulty)

{

var items = new List<ExerciseItemDto>();

var random = new Random();

for (int i = 0; i < 10; i++)

{

int a, b;

string operation, expression, answer;

switch (difficulty)

{

case DifficultyLevel.Beginner:

a = random.Next(1, 10);

b = random.Next(1, 10);

operation = random.Next(2) == 0 ? "+" : "-";

expression = $"{a} {operation} {b} = ?";

answer = operation == "+" ? (a + b).ToString() : Math.Max(a - b, 0).ToString();

break;

default:

a = random.Next(10, 100);

b = random.Next(10, 100);

operation = new[] { "+", "-", "×", "÷" }[random.Next(4)];

expression = $"{a} {operation} {b} = ?";

answer = operation switch

{

"+" => (a + b).ToString(),

"-" => (a - b).ToString(),

"×" => (a \* b).ToString(),

"÷" => (a / b).ToString(),

\_ => "0"

};

break;

}

items.Add(new ExerciseItemDto

{

Content = expression,

ExpectedResponse = answer,

ItemType = "MathProblem"

});

}

return items;

}

private TrainingDataType DetermineDataType(string content)

{

if (content.All(char.IsLetter) && content.Length == 1)

return TrainingDataType.Alphabet;

if (content.All(char.IsDigit))

return TrainingDataType.Numbers;

if (content.Contains('+') || content.Contains('-') || content.Contains('×') || content.Contains('÷'))

return TrainingDataType.MathExpressions;

if (content.Split(' ').Length > 5)

return TrainingDataType.Sentences;

return TrainingDataType.Words;

}

private async Task<List<HandwritingCharacterData>> ExtractCharacterDataAsync(HandwritingTrainingData trainingData)

{

var characterData = new List<HandwritingCharacterData>();

// This would use computer vision to segment the image into individual characters

var characterSegments = await \_imageProcessingService.SegmentCharactersAsync(trainingData.ImagePath);

for (int i = 0; i < characterSegments.Count && i < trainingData.OriginalContent.Length; i++)

{

var segment = characterSegments[i];

var character = trainingData.OriginalContent[i].ToString();

characterData.Add(new HandwritingCharacterData

{

HandwritingTrainingDataId = trainingData.Id,

Character = character,

ImageSegmentPath = segment.ImagePath,

X = segment.BoundingBox.X,

Y = segment.BoundingBox.Y,

Width = segment.BoundingBox.Width,

Height = segment.BoundingBox.Height,

ConfidenceScore = segment.Confidence,

SequenceOrder = i,

FeatureVector = segment.FeatureVector

});

}

return characterData;

}

private decimal CalculateSessionAccuracy(StudentTrainingSession session)

{

if (session.CompletedItems == 0) return 0;

return (decimal)session.CorrectItems / session.CompletedItems \* 100;

}

private async Task UpdateCharacterVerificationAsync(HandwritingTrainingData trainingData, string verifiedText)

{

for (int i = 0; i < Math.Min(trainingData.CharacterData.Count, verifiedText.Length); i++)

{

var characterData = trainingData.CharacterData.Skip(i).First();

characterData.Character = verifiedText[i].ToString();

characterData.IsVerified = true;

}

}

private async Task CheckAndQueueModelTrainingAsync(Guid studentId)

{

var verifiedCount = await \_context.HandwritingTrainingData

.CountAsync(td => td.StudentId == studentId && td.IsVerified);

var existingJob = await \_context.ModelTrainingJobs

.Where(j => j.StudentId == studentId &&

(j.Status == JobStatus.Queued || j.Status == JobStatus.Running))

.FirstOrDefaultAsync();

if (verifiedCount >= 100 && existingJob == null)

{

await StartModelTrainingAsync(studentId);

}

}

private async Task ProcessModelTrainingAsync(Guid jobId)

{

var job = await \_context.ModelTrainingJobs.FindAsync(jobId);

if (job == null) return;

try

{

job.Status = JobStatus.Running;

job.StartedAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

// Get training data

var trainingData = await \_context.HandwritingTrainingData

.Where(td => td.StudentId == job.StudentId && td.IsVerified)

.Include(td => td.CharacterData)

.ToListAsync();

// Train the model

var modelResult = await \_mlService.TrainPersonalModelAsync(trainingData);

// Create model record

var model = new HandwritingModel

{

StudentId = job.StudentId,

ModelName = $"Personal Model v{DateTime.UtcNow:yyyyMMdd}",

Version = "1.0",

Type = ModelType.Personal,

Accuracy = modelResult.Accuracy,

TrainingDataCount = trainingData.Count,

CloudModelPath = modelResult.CloudPath,

TrainingStarted = job.StartedAt.Value,

TrainingCompleted = DateTime.UtcNow,

TrainingParameters = job.TrainingParameters,

PerformanceMetrics = System.Text.Json.JsonSerializer.Serialize(modelResult.Metrics),

IsActive = true,

IsDeployedCloud = true

};

\_context.HandwritingModels.Add(model);

job.Status = JobStatus.Completed;

job.CompletedAt = DateTime.UtcNow;

job.Progress = 100;

job.ResultPath = modelResult.CloudPath;

await \_context.SaveChangesAsync();

// Optionally deploy locally immediately

await DeployModelLocallyAsync(model.Id);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Model training failed for job {JobId}", jobId);

job.Status = JobStatus.Failed;

job.ErrorMessage = ex.Message;

await \_context.SaveChangesAsync();

}

}

private object GetDefaultTrainingParameters()

{

return new

{

LearningRate = 0.001,

BatchSize = 32,

Epochs = 100,

ValidationSplit = 0.2,

EarlyStopping = true,

Patience = 10

};

}

private async Task<string> GetCurrentModelVersionAsync()

{

return $"v{DateTime.UtcNow:yyyyMMdd}";

}

}

// Supporting DTOs and classes

public class AnswerComparisonResult

{

public bool IsCorrect { get; set; }

public decimal PartialCredit { get; set; }

public decimal Similarity { get; set; }

}

public class ExerciseContentDto

{

public List<ExerciseItemDto> Items { get; set; } = new();

}

public class ExerciseItemDto

{

public string Content { get; set; } = "";

public string ExpectedResponse { get; set; } = "";

public string ItemType { get; set; } = "";

}

public class HandwritingProcessingException : Exception

{

public HandwritingProcessingException(string message) : base(message) { }

public HandwritingProcessingException(string message, Exception innerException) : base(message, innerException) { }

}

}// DTOs/SchoolDtos.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.DTOs

{

public class SchoolDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string Website { get; set; } = string.Empty;

public string RegistrationNumber { get; set; } = string.Empty;

public string Logo { get; set; } = string.Empty;

public GradingSchemeDto? GradingScheme { get; set; }

public DateTime CreatedAt { get; set; }

}

public class CreateSchoolDto

{

[Required]

[StringLength(200)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Address { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(100)]

public string Website { get; set; } = string.Empty;

[StringLength(50)]

public string RegistrationNumber { get; set; } = string.Empty;

public Guid? GradingSchemeId { get; set; }

}

public class UpdateSchoolDto

{

[StringLength(200)]

public string? Name { get; set; }

[StringLength(500)]

public string? Address { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(100)]

public string? Website { get; set; }

public Guid? GradingSchemeId { get; set; }

}

}

// DTOs/StudentDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class StudentDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string StudentNumber { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public GradeDto CurrentGrade { get; set; } = null!;

public SchoolDto School { get; set; } = null!;

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

public DateTime CreatedAt { get; set; }

}

public class CreateStudentDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[StringLength(50)]

public string? StudentNumber { get; set; }

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

[StringLength(10)]

public string Gender { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid CurrentGradeId { get; set; }

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

}

public class UpdateStudentDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[StringLength(200)]

public string? Address { get; set; }

public Guid? CurrentGradeId { get; set; }

}

public class StudentTimetableDto

{

public StudentDto Student { get; set; } = null!;

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

public string TimetableName { get; set; } = string.Empty;

}

}

// DTOs/TeacherDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TeacherDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string EmployeeId { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public DateTime HireDate { get; set; }

public string Qualification { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public SchoolDto School { get; set; } = null!;

}

public class CreateTeacherDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[Required]

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[StringLength(50)]

public string EmployeeId { get; set; } = string.Empty;

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

public DateTime HireDate { get; set; }

[StringLength(100)]

public string Qualification { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

}

public class UpdateTeacherDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[StringLength(200)]

public string? Address { get; set; }

[StringLength(100)]

public string? Qualification { get; set; }

}

public class AssignTeacherSubjectDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class AssignClassTeacherDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class TeacherTimetableDto

{

public TeacherDto Teacher { get; set; } = null!;

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

public TermDto Term { get; set; } = null!;

}

}

// DTOs/GradingDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class GradingSchemeDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public string Country { get; set; } = string.Empty;

public bool IsActive { get; set; }

public List<GradingScaleDto> GradingScales { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class CreateGradingSchemeDto

{

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Description { get; set; } = string.Empty;

[StringLength(50)]

public string Country { get; set; } = string.Empty;

public List<CreateGradingScaleDto> GradingScales { get; set; } = new();

}

public class UpdateGradingSchemeDto

{

[StringLength(100)]

public string? Name { get; set; }

[StringLength(500)]

public string? Description { get; set; }

[StringLength(50)]

public string? Country { get; set; }

public bool? IsActive { get; set; }

}

public class GradingScaleDto

{

public Guid Id { get; set; }

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class CreateGradingScaleDto

{

[Required]

[StringLength(10)]

public string Symbol { get; set; } = string.Empty;

[Required]

public int Unit { get; set; }

[Required]

[Range(0, 100)]

public decimal MinPercentage { get; set; }

[Required]

[Range(0, 100)]

public decimal MaxPercentage { get; set; }

[StringLength(100)]

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class UpdateGradingScaleDto

{

[StringLength(10)]

public string? Symbol { get; set; }

public int? Unit { get; set; }

[Range(0, 100)]

public decimal? MinPercentage { get; set; }

[Range(0, 100)]

public decimal? MaxPercentage { get; set; }

[StringLength(100)]

public string? Description { get; set; }

public int? SortOrder { get; set; }

}

public class GradeCalculationDto

{

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public string Description { get; set; } = string.Empty;

public decimal Percentage { get; set; }

}

}

// DTOs/ResultDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ResultDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public decimal Score { get; set; }

public decimal TotalMarks { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public int GradeUnit { get; set; }

public string AssessmentType { get; set; } = string.Empty;

public string Comments { get; set; } = string.Empty;

public DateTime AssessmentDate { get; set; }

public bool IsAiVerified { get; set; }

public decimal AiConfidenceScore { get; set; }

}

public class CreateResultDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

[Range(0, double.MaxValue)]

public decimal Score { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal TotalMarks { get; set; }

[Required]

[StringLength(50)]

public string AssessmentType { get; set; } = string.Empty;

[StringLength(500)]

public string Comments { get; set; } = string.Empty;

[Required]

public DateTime AssessmentDate { get; set; }

}

public class UpdateResultDto

{

[Range(0, double.MaxValue)]

public decimal? Score { get; set; }

[Range(0.01, double.MaxValue)]

public decimal? TotalMarks { get; set; }

[StringLength(50)]

public string? AssessmentType { get; set; }

[StringLength(500)]

public string? Comments { get; set; }

public DateTime? AssessmentDate { get; set; }

}

public class StudentReportCardDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

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

public List<AttendanceSummaryDto> AttendanceSummary { get; set; } = new();

public DateTime GeneratedAt { get; set; }

public decimal OverallAverage { get; set; }

public string OverallGrade { get; set; } = string.Empty;

public int TotalSubjects { get; set; }

public int PassedSubjects { get; set; }

}

public class ClassResultSummaryDto

{

public GradeDto Grade { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<SubjectResultSummaryDto> SubjectSummaries { get; set; } = new();

public int TotalStudents { get; set; }

public decimal OverallAverage { get; set; }

}

public class SubjectResultSummaryDto

{

public SubjectDto Subject { get; set; } = null!;

public int TotalStudents { get; set; }

public decimal AverageScore { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public int PassCount { get; set; }

public int FailCount { get; set; }

public decimal PassRate { get; set; }

}

}

// DTOs/AttendanceDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class AttendanceDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

public string Remarks { get; set; } = string.Empty;

}

public class CreateAttendanceDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public DateTime Date { get; set; }

[Required]

public AttendanceStatus Status { get; set; }

[StringLength(200)]

public string Remarks { get; set; } = string.Empty;

}

public class UpdateAttendanceDto

{

public AttendanceStatus? Status { get; set; }

[StringLength(200)]

public string? Remarks { get; set; }

}

public class AttendanceReportDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<AttendanceSummaryDto> SubjectAttendance { get; set; } = new();

public int TotalClassesTaken { get; set; }

public int TotalClassesAttended { get; set; }

public decimal OverallAttendancePercentage { get; set; }

}

public class AttendanceSummaryDto

{

public Guid SubjectId { get; set; }

public string SubjectName { get; set; } = string.Empty;

public int TotalClasses { get; set; }

public int PresentClasses { get; set; }

public int AbsentClasses { get; set; }

public int LateClasses { get; set; }

public decimal AttendancePercentage { get; set; }

}

}

// DTOs/TimetableDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TimetableDto

{

public Guid Id { get; set; }

public SchoolDto School { get; set; } = null!;

public GradeDto Grade { get; set; } = null!;

public SchoolYearDto SchoolYear { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public string Name { get; set; } = string.Empty;

public bool IsActive { get; set; }

public bool IsGenerated { get; set; }

public DateTime GeneratedAt { get; set; }

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

}

public class TimetableSlotDto

{

public Guid Id { get; set; }

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public string Room { get; set; } = string.Empty;

public int PeriodNumber { get; set; }

public string Notes { get; set; } = string.Empty;

}

public class GenerateTimetableDto

{

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

public TimetableGenerationRulesDto Rules { get; set; } = new();

}

public class TimetableGenerationRulesDto

{

public List<DayOfWeek> WorkingDays { get; set; } = new()

{

DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday,

DayOfWeek.Thursday, DayOfWeek.Friday

};

public TimeSpan SchoolStartTime { get; set; } = new(8, 0, 0);

public TimeSpan SchoolEndTime { get; set; } = new(15, 0, 0);

public TimeSpan PeriodDuration { get; set; } = new(0, 40, 0);

public TimeSpan BreakDuration { get; set; } = new(0, 15, 0);

public TimeSpan LunchDuration { get; set; } = new(1, 0, 0);

public int MaxPeriodsPerDay { get; set; } = 8;

public int MaxConsecutivePeriods { get; set; } = 3;

// Subject-specific rules

public Dictionary<Guid, int> SubjectPeriodsPerWeek { get; set; } = new();

public Dictionary<Guid, List<DayOfWeek>> SubjectPreferredDays { get; set; } = new();

public Dictionary<Guid, List<int>> SubjectPreferredPeriods { get; set; } = new();

// Teacher constraints

public Dictionary<Guid, List<TimeSlotDto>> TeacherUnavailableSlots { get; set; } = new();

// Room assignments

public Dictionary<Guid, string> SubjectRoomPreferences { get; set; } = new();

}

public class TimeSlotDto

{

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

}

public class UpdateTimetableSlotDto

{

public Guid? SubjectId { get; set; }

public Guid? TeacherId { get; set; }

public TimeSpan? StartTime { get; set; }

public TimeSpan? EndTime { get; set; }

public string? Room { get; set; }

public string? Notes { get; set; }

}

public class TimetableConflictDto

{

public bool HasConflicts { get; set; }

public List<ConflictDto> Conflicts { get; set; } = new();

}

public class ConflictDto

{

public ConflictType Type { get; set; }

public string Description { get; set; } = string.Empty;

public List<Guid> AffectedSlotIds { get; set; } = new();

public string Severity { get; set; } = string.Empty;

}

public enum ConflictType

{

TeacherDoubleBooking,

RoomDoubleBooking,

SubjectOverallocation,

InvalidTimeSlot,

ConsecutivePeriodViolation

}

}

// DTOs/NotificationDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class NotificationDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Message { get; set; } = string.Empty;

public NotificationType Type { get; set; }

public NotificationChannel Channel { get; set; }

public bool IsRead { get; set; }

public bool IsSent { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? SentAt { get; set; }

public DateTime? ReadAt { get; set; }

public string ExternalId { get; set; } = string.Empty;

public ParentDto? Parent { get; set; }

public StudentDto? Student { get; set; }

public TeacherDto? Teacher { get; set; }

}

public class CreateNotificationDto

{

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[Required]

public string Message { get; set; } = string.Empty;

[Required]

public NotificationType Type { get; set; }

[Required]

public NotificationChannel Channel { get; set; }

public Guid? ParentId { get; set; }

public Guid? StudentId { get; set; }

public Guid? TeacherId { get; set; }

public string Metadata { get; set; } = string.Empty;

}

public class NotificationStatsDto

{

public int TotalNotifications { get; set; }

public int SentNotifications { get; set; }

public int PendingNotifications { get; set; }

public int ReadNotifications { get; set; }

public Dictionary<NotificationType, int> NotificationsByType { get; set; } = new();

public Dictionary<NotificationChannel, int> NotificationsByChannel { get; set; } = new();

}

}

// DTOs/HandwritingAIDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ProcessHandwritingDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public string ImagePath { get; set; } = string.Empty;

public Guid? SubjectId { get; set; }

public Guid? ResultId { get; set; }

public string? ExpectedAnswer { get; set; }

}

public class HandwritingRecognitionResultDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string RecognizedText { get; set; } = string.Empty;

public decimal OverallConfidence { get; set; }

public bool ProcessedLocally { get; set; }

public bool ProcessedInCloud { get; set; }

public TimeSpan ProcessingTime { get; set; }

public string ExpectedAnswer { get; set; } = string.Empty;

public bool IsCorrect { get; set; }

public decimal PartialCreditScore { get; set; }

public DateTime ProcessedAt { get; set; }

public List<string> AlternativeTexts { get; set; } = new();

public List<decimal> CharacterConfidences { get; set; } = new();

}

public class GenerateTrainingExerciseDto

{

[Required]

public Guid SchoolId { get; set; }

public Guid? GradeId { get; set; }

public Guid? SubjectId { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

[Required]

public ExerciseType Type { get; set; }

[Required]

public DifficultyLevel Difficulty { get; set; }

public int EstimatedMinutes { get; set; } = 10;

}

public class TrainingExerciseDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public ExerciseType Type { get; set; }

public DifficultyLevel Difficulty { get; set; }

public bool IsActive { get; set; }

public bool IsSystemGenerated { get; set; }

public int EstimatedMinutes { get; set; }

public ExerciseContentDto Content { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class StudentTrainingSessionDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingExerciseDto TrainingExercise { get; set; } = null!;

public DateTime StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public SessionStatus Status { get; set; }

public int TotalItems { get; set; }

public int CompletedItems { get; set; }

public int CorrectItems { get; set; }

public decimal OverallAccuracy { get; set; }

public TimeSpan TotalTime { get; set; }

}

public class SubmitTrainingDataDto

{

[Required]

public Guid SessionId { get; set; }

[Required]

public string OriginalContent { get; set; } = string.Empty;

[Required]

public byte[] HandwritingImage { get; set; } = Array.Empty<byte>();

}

public class HandwritingTrainingDataDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingDataType DataType { get; set; }

public string OriginalContent { get; set; } = string.Empty;

public string ImagePath { get; set; } = string.Empty;

public string ProcessedText { get; set; } = string.Empty;

public string VerifiedText { get; set; } = string.Empty;

public decimal ConfidenceScore { get; set; }

public TrainingStatus Status { get; set; }

public bool IsVerified { get; set; }

public bool IsTrainingComplete { get; set; }

public DateTime CapturedAt { get; set; }

public DateTime? VerifiedAt { get; set; }

public DateTime? TrainedAt { get; set; }

public StudentDto Student { get; set; } = null!;

}

public class ModelTrainingJobDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string JobName { get; set; } = string.Empty;

public JobType Type { get; set; }

public JobStatus Status { get; set; }

public int TotalDataPoints { get; set; }

public int ProcessedDataPoints { get; set; }

public decimal Progress { get; set; }

public DateTime QueuedAt { get; set; }

public DateTime? StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public string ErrorMessage { get; set; } = string.Empty;

public string ResultPath { get; set; } = string.Empty;

}

public class HandwritingModelDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string ModelName { get; set; } = string.Empty;

public string Version { get; set; } = string.Empty;

public ModelType Type { get; set; }

public decimal Accuracy { get; set; }

public int TrainingDataCount { get; set; }

public bool IsActive { get; set; }}// DTOs/SchoolDtos.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.DTOs

{

public class SchoolDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string Website { get; set; } = string.Empty;

public string RegistrationNumber { get; set; } = string.Empty;

public string Logo { get; set; } = string.Empty;

public GradingSchemeDto? GradingScheme { get; set; }

public DateTime CreatedAt { get; set; }

}

public class CreateSchoolDto

{

[Required]

[StringLength(200)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Address { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(100)]

public string Website { get; set; } = string.Empty;

[StringLength(50)]

public string RegistrationNumber { get; set; } = string.Empty;

public Guid? GradingSchemeId { get; set; }

}

public class UpdateSchoolDto

{

[StringLength(200)]

public string? Name { get; set; }

[StringLength(500)]

public string? Address { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(100)]

public string? Website { get; set; }

public Guid? GradingSchemeId { get; set; }

}

}

// DTOs/StudentDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class StudentDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string StudentNumber { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public string Gender { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public GradeDto CurrentGrade { get; set; } = null!;

public SchoolDto School { get; set; } = null!;

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

public DateTime CreatedAt { get; set; }

}

public class CreateStudentDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[StringLength(50)]

public string? StudentNumber { get; set; }

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

[StringLength(10)]

public string Gender { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid CurrentGradeId { get; set; }

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

}

public class UpdateStudentDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[StringLength(200)]

public string? Address { get; set; }

public Guid? CurrentGradeId { get; set; }

}

public class StudentTimetableDto

{

public StudentDto Student { get; set; } = null!;

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

public string TimetableName { get; set; } = string.Empty;

}

}

// DTOs/TeacherDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TeacherDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string EmployeeId { get; set; } = string.Empty;

public DateTime DateOfBirth { get; set; }

public DateTime HireDate { get; set; }

public string Qualification { get; set; } = string.Empty;

public string ProfilePicture { get; set; } = string.Empty;

public SchoolDto School { get; set; } = null!;

}

public class CreateTeacherDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[Required]

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[StringLength(50)]

public string EmployeeId { get; set; } = string.Empty;

[Required]

public DateTime DateOfBirth { get; set; }

[Required]

public DateTime HireDate { get; set; }

[StringLength(100)]

public string Qualification { get; set; } = string.Empty;

[Required]

public Guid SchoolId { get; set; }

}

public class UpdateTeacherDto

{

[StringLength(100)]

public string? FirstName { get; set; }

[StringLength(100)]

public string? LastName { get; set; }

[EmailAddress]

[StringLength(100)]

public string? Email { get; set; }

[StringLength(20)]

public string? PhoneNumber { get; set; }

[StringLength(200)]

public string? Address { get; set; }

[StringLength(100)]

public string? Qualification { get; set; }

}

public class AssignTeacherSubjectDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class AssignClassTeacherDto

{

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

}

public class TeacherTimetableDto

{

public TeacherDto Teacher { get; set; } = null!;

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

public TermDto Term { get; set; } = null!;

}

}

// DTOs/GradingDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class GradingSchemeDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public string Country { get; set; } = string.Empty;

public bool IsActive { get; set; }

public bool IsDeployedLocally { get; set; }

public bool IsDeployedCloud { get; set; }

public DateTime TrainingStarted { get; set; }

public DateTime? TrainingCompleted { get; set; }

public DateTime? LastUsed { get; set; }

public string ModelFilePath { get; set; } = string.Empty;

public string CloudModelPath { get; set; } = string.Empty;

public Dictionary<string, object> PerformanceMetrics { get; set; } = new();

}

public class TrainingProgressDto

{

public Guid StudentId { get; set; }

public int TotalTrainingData { get; set; }

public int VerifiedData { get; set; }

public int PendingVerification { get; set; }

public decimal CurrentModelAccuracy { get; set; }

public bool ModelIsDeployedLocally { get; set; }

public List<StudentTrainingSessionDto> RecentSessions { get; set; } = new();

public bool ReadyForTraining { get; set; }

public decimal CompletionPercentage { get; set; }

}

}

// DTOs/CoreDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class GradeDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public int Level { get; set; }

public int MaxStudents { get; set; }

public SchoolDto School { get; set; } = null!;

}

public class SubjectDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public string Code { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public int Credits { get; set; }

public bool IsActive { get; set; }

public SchoolDto School { get; set; } = null!;

}

public class TermDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public int TermNumber { get; set; }

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public bool IsCurrent { get; set; }

public SchoolYearDto SchoolYear { get; set; } = null!;

}

public class SchoolYearDto

{

public Guid Id { get; set; }

public string Name { get; set; } = string.Empty;

public DateTime StartDate { get; set; }

public DateTime EndDate { get; set; }

public bool IsActive { get; set; }

public bool IsCurrent { get; set; }

public SchoolDto School { get; set; } = null!;

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

}

public class ParentDto

{

public Guid Id { get; set; }

public string FirstName { get; set; } = string.Empty;

public string LastName { get; set; } = string.Empty;

public string Email { get; set; } = string.Empty;

public string PhoneNumber { get; set; } = string.Empty;

public string WhatsAppNumber { get; set; } = string.Empty;

public string Address { get; set; } = string.Empty;

public string Relationship { get; set; } = string.Empty;

public string Occupation { get; set; } = string.Empty;

}

public class CreateParentDto

{

[Required]

[StringLength(100)]

public string FirstName { get; set; } = string.Empty;

[Required]

[StringLength(100)]

public string LastName { get; set; } = string.Empty;

[Required]

[EmailAddress]

[StringLength(100)]

public string Email { get; set; } = string.Empty;

[StringLength(20)]

public string PhoneNumber { get; set; } = string.Empty;

[StringLength(20)]

public string WhatsAppNumber { get; set; } = string.Empty;

[StringLength(200)]

public string Address { get; set; } = string.Empty;

[Required]

[StringLength(50)]

public string Relationship { get; set; } = string.Empty;

[StringLength(100)]

public string Occupation { get; set; } = string.Empty;

}

}

// Configuration/AutoMapperProfile.cs

using AutoMapper;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

namespace SchoolManagement.Configuration

{

public class AutoMapperProfile : Profile

{

public AutoMapperProfile()

{

// School mappings

CreateMap<School, SchoolDto>().ReverseMap();

CreateMap<CreateSchoolDto, School>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateSchoolDto, School>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Student mappings

CreateMap<Student, StudentDto>().ReverseMap();

CreateMap<CreateStudentDto, Student>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.Parents, opt => opt.MapFrom(src => src.Parents));

CreateMap<UpdateStudentDto, Student>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Teacher mappings

CreateMap<Teacher, TeacherDto>().ReverseMap();

CreateMap<CreateTeacherDto, Teacher>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateTeacherDto, Teacher>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Parent mappings

CreateMap<Parent, ParentDto>().ReverseMap();

CreateMap<CreateParentDto, Parent>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

// Grading mappings

CreateMap<GradingScheme, GradingSchemeDto>().ReverseMap();

CreateMap<CreateGradingSchemeDto, GradingScheme>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.IsActive, opt => opt.MapFrom(src => true));

CreateMap<UpdateGradingSchemeDto, GradingScheme>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

CreateMap<GradingScale, GradingScaleDto>().ReverseMap();

CreateMap<CreateGradingScaleDto, GradingScale>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateGradingScaleDto, GradingScale>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Core entity mappings

CreateMap<Grade, GradeDto>().ReverseMap();

CreateMap<Subject, SubjectDto>().ReverseMap();

CreateMap<Term, TermDto>().ReverseMap();

CreateMap<SchoolYear, SchoolYearDto>().ReverseMap();

// Result mappings

CreateMap<Result, ResultDto>().ReverseMap();

CreateMap<CreateResultDto, Result>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.Percentage, opt => opt.Ignore())

.ForMember(dest => dest.Grade, opt => opt.Ignore())

.ForMember(dest => dest.GradeUnit, opt => opt.Ignore());

CreateMap<UpdateResultDto, Result>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Attendance mappings

CreateMap<Attendance, AttendanceDto>().ReverseMap();

CreateMap<CreateAttendanceDto, Attendance>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore());

CreateMap<UpdateAttendanceDto, Attendance>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Timetable mappings

CreateMap<Timetable, TimetableDto>().ReverseMap();

CreateMap<TimetableSlot, TimetableSlotDto>().ReverseMap();

CreateMap<UpdateTimetableSlotDto, TimetableSlot>()

.ForAllMembers(opts => opts.Condition((src, dest, srcMember) => srcMember != null));

// Notification mappings

CreateMap<Notification, NotificationDto>().ReverseMap();

CreateMap<CreateNotificationDto, Notification>()

.ForMember(dest => dest.Id, opt => opt.Ignore())

.ForMember(dest => dest.CreatedAt, opt => opt.Ignore())

.ForMember(dest => dest.UpdatedAt, opt => opt.Ignore())

.ForMember(dest => dest.IsRead, opt => opt.MapFrom(src => false))

.ForMember(dest => dest.IsSent, opt => opt.MapFrom(src => false));

// AI Handwriting mappings

CreateMap<HandwritingTrainingData, HandwritingTrainingDataDto>().ReverseMap();

CreateMap<HandwritingRecognitionResult, HandwritingRecognitionResultDto>()

.ForMember(dest => dest.AlternativeTexts, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.AlternativeTexts) ?

new List<string>() :

System.Text.Json.JsonSerializer.Deserialize<List<string>>(src.AlternativeTexts)))

.ForMember(dest => dest.CharacterConfidences, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.CharacterConfidences) ?

new List<decimal>() :

System.Text.Json.JsonSerializer.Deserialize<List<decimal>>(src.CharacterConfidences)));

CreateMap<TrainingExercise, TrainingExerciseDto>()

.ForMember(dest => dest.Content, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.Content) ?

new ExerciseContentDto() :

System.Text.Json.JsonSerializer.Deserialize<ExerciseContentDto>(src.Content)));

CreateMap<StudentTrainingSession, StudentTrainingSessionDto>().ReverseMap();

CreateMap<ModelTrainingJob, ModelTrainingJobDto>().ReverseMap();

CreateMap<HandwritingModel, HandwritingModelDto>()

.ForMember(dest => dest.PerformanceMetrics, opt => opt.MapFrom(src =>

string.IsNullOrEmpty(src.PerformanceMetrics) ?

new Dictionary<string, object>() :

System.Text.Json.JsonSerializer.Deserialize<Dictionary<string, object>>(src.PerformanceMetrics)));

}

}

}

// Services/Interfaces/Supporting/IImageProcessingService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IImageProcessingService

{

Task<ImageProcessingResult> ProcessImageAsync(string imagePath);

Task<TextExtractionResult> ExtractTextAsync(string imagePath);

Task<List<CharacterSegment>> SegmentCharactersAsync(string imagePath);

Task<string> PreprocessImageAsync(string imagePath);

Task<bool> ValidateImageQualityAsync(string imagePath);

}

public class ImageProcessingResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public string BoundingBoxes { get; set; } = string.Empty;

public string CharacterConfidences { get; set; } = string.Empty;

public string AlternativeTexts { get; set; } = string.Empty;

}

public class TextExtractionResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public List<CharacterSegment> Characters { get; set; } = new();

}

public class CharacterSegment

{

public string ImagePath { get; set; } = string.Empty;

public Rectangle BoundingBox { get; set; }

public decimal Confidence { get; set; }

public string FeatureVector { get; set; } = string.Empty;

}

public struct Rectangle

{

public int X { get; set; }

public int Y { get; set; }

public int Width { get; set; }

public int Height { get; set; }

}

}

// Services/Interfaces/Supporting/IMachineLearningService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IMachineLearningService

{

Task<LocalRecognitionResult> RecognizeTextLocallyAsync(string modelPath, string imagePath);

Task<CloudRecognitionResult> RecognizeTextInCloudAsync(string imagePath);

Task<ModelTrainingResult> TrainPersonalModelAsync(List<HandwritingTrainingData> trainingData);

Task<bool> ValidateModelAsync(string modelPath);

Task<ModelMetrics> EvaluateModelAsync(string modelPath, List<HandwritingTrainingData> testData);

Task<string> DeployModelToCloudAsync(string localModelPath);

Task<bool> UpdateModelAsync(string existingModelPath, List<HandwritingTrainingData> newTrainingData);

}

public class LocalRecognitionResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public TimeSpan ProcessingTime { get; set; }

public List<CharacterRecognition> Characters { get; set; } = new();

}

public class CloudRecognitionResult

{

public string Text { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public TimeSpan ProcessingTime { get; set; }

public List<string> Alternatives { get; set; } = new();

}

public class ModelTrainingResult

{

public string CloudPath { get; set; } = string.Empty;

public string LocalPath { get; set; } = string.Empty;

public decimal Accuracy { get; set; }

public ModelMetrics Metrics { get; set; } = new();

public TimeSpan TrainingTime { get; set; }

}

public class ModelMetrics

{

public decimal Accuracy { get; set; }

public decimal Precision { get; set; }

public decimal Recall { get; set; }

public decimal F1Score { get; set; }

public Dictionary<string, decimal> CharacterAccuracies { get; set; } = new();

public int TotalSamples { get; set; }

public int CorrectPredictions { get; set; }

}

public class CharacterRecognition

{

public string Character { get; set; } = string.Empty;

public decimal Confidence { get; set; }

public Rectangle BoundingBox { get; set; }

public List<string> Alternatives { get; set; } = new();

}

}

// Services/Interfaces/Supporting/IFileStorageService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IFileStorageService

{

Task<string> SaveImageAsync(byte[] imageData, string relativePath);

Task<string> SaveFileAsync(byte[] fileData, string relativePath, string contentType);

Task<byte[]> GetFileAsync(string filePath);

Task<bool> DeleteFileAsync(string filePath);

Task<string> DownloadModelAsync(string cloudPath, Guid modelId);

Task<string> UploadModelAsync(string localPath, string cloudPath);

Task<List<string>> GetFilesInDirectoryAsync(string directoryPath);

Task<bool> FileExistsAsync(string filePath);

Task<long> GetFileSizeAsync(string filePath);

}

}

// Services/Interfaces/Supporting/IPdfGenerationService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IPdfGenerationService

{

Task<byte[]> GenerateReportCardPdfAsync(StudentReportCardDto reportCard);

Task<byte[]> GenerateClassResultsPdfAsync(ClassResultSummaryDto classSummary);

Task<byte[]> GenerateTimetablePdfAsync(TimetableDto timetable);

Task<byte[]> GenerateAttendanceReportPdfAsync(AttendanceReportDto attendanceReport);

Task<byte[]> GenerateCertificatePdfAsync(StudentDto student, string certificateType);

Task<byte[]> GenerateTranscriptPdfAsync(StudentDto student, List<ResultDto> results);

}

public class PdfGenerationService : IPdfGenerationService

{

public async Task<byte[]> GenerateReportCardPdfAsync(StudentReportCardDto reportCard)

{

// Implementation using iTextSharp, PdfSharp, or similar library

// This is a placeholder implementation

await Task.Delay(100); // Simulate processing time

return new byte[0]; // Return generated PDF bytes

}

public async Task<byte[]> GenerateClassResultsPdfAsync(ClassResultSummaryDto classSummary)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateTimetablePdfAsync(TimetableDto timetable)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateAttendanceReportPdfAsync(AttendanceReportDto attendanceReport)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateCertificatePdfAsync(StudentDto student, string certificateType)

{

await Task.Delay(100);

return new byte[0];

}

public async Task<byte[]> GenerateTranscriptPdfAsync(StudentDto student, List<ResultDto> results)

{

await Task.Delay(100);

return new byte[0];

}

}

}

// Exceptions/CustomExceptions.cs

namespace SchoolManagement.Core.Exceptions

{

public class NotFoundException : Exception

{

public NotFoundException(string message) : base(message) { }

public NotFoundException(string message, Exception innerException) : base(message, innerException) { }

}

public class ValidationException : Exception

{

public Dictionary<string, List<string>> Errors { get; }

public ValidationException(string message) : base(message)

{

Errors = new Dictionary<string, List<string>>();

}

public ValidationException(Dictionary<string, List<string>> errors) : base("Validation failed")

{

Errors = errors;

}

}

public class BusinessRuleException : Exception

{

public string RuleCode { get; }

public BusinessRuleException(string ruleCode, string message) : base(message)

{

RuleCode = ruleCode;

}

}

public class UnauthorizedException : Exception

{

public UnauthorizedException(string message) : base(message) { }

}

public class ConflictException : Exception

{

public ConflictException(string message) : base(message) { }

}

}

public List<GradingScaleDto> GradingScales { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class CreateGradingSchemeDto

{

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

[StringLength(500)]

public string Description { get; set; } = string.Empty;

[StringLength(50)]

public string Country { get; set; } = string.Empty;

public List<CreateGradingScaleDto> GradingScales { get; set; } = new();

}

public class UpdateGradingSchemeDto

{

[StringLength(100)]

public string? Name { get; set; }

[StringLength(500)]

public string? Description { get; set; }

[StringLength(50)]

public string? Country { get; set; }

public bool? IsActive { get; set; }

}

public class GradingScaleDto

{

public Guid Id { get; set; }

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public decimal MinPercentage { get; set; }

public decimal MaxPercentage { get; set; }

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class CreateGradingScaleDto

{

[Required]

[StringLength(10)]

public string Symbol { get; set; } = string.Empty;

[Required]

public int Unit { get; set; }

[Required]

[Range(0, 100)]

public decimal MinPercentage { get; set; }

[Required]

[Range(0, 100)]

public decimal MaxPercentage { get; set; }

[StringLength(100)]

public string Description { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class UpdateGradingScaleDto

{

[StringLength(10)]

public string? Symbol { get; set; }

public int? Unit { get; set; }

[Range(0, 100)]

public decimal? MinPercentage { get; set; }

[Range(0, 100)]

public decimal? MaxPercentage { get; set; }

[StringLength(100)]

public string? Description { get; set; }

public int? SortOrder { get; set; }

}

public class GradeCalculationDto

{

public string Symbol { get; set; } = string.Empty;

public int Unit { get; set; }

public string Description { get; set; } = string.Empty;

public decimal Percentage { get; set; }

}

}

// DTOs/ResultDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ResultDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public decimal Score { get; set; }

public decimal TotalMarks { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public int GradeUnit { get; set; }

public string AssessmentType { get; set; } = string.Empty;

public string Comments { get; set; } = string.Empty;

public DateTime AssessmentDate { get; set; }

public bool IsAiVerified { get; set; }

public decimal AiConfidenceScore { get; set; }

}

public class CreateResultDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

[Range(0, double.MaxValue)]

public decimal Score { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal TotalMarks { get; set; }

[Required]

[StringLength(50)]

public string AssessmentType { get; set; } = string.Empty;

[StringLength(500)]

public string Comments { get; set; } = string.Empty;

[Required]

public DateTime AssessmentDate { get; set; }

}

public class UpdateResultDto

{

[Range(0, double.MaxValue)]

public decimal? Score { get; set; }

[Range(0.01, double.MaxValue)]

public decimal? TotalMarks { get; set; }

[StringLength(50)]

public string? AssessmentType { get; set; }

[StringLength(500)]

public string? Comments { get; set; }

public DateTime? AssessmentDate { get; set; }

}

public class StudentReportCardDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

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

public List<AttendanceSummaryDto> AttendanceSummary { get; set; } = new();

public DateTime GeneratedAt { get; set; }

public decimal OverallAverage { get; set; }

public string OverallGrade { get; set; } = string.Empty;

public int TotalSubjects { get; set; }

public int PassedSubjects { get; set; }

}

public class ClassResultSummaryDto

{

public GradeDto Grade { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<SubjectResultSummaryDto> SubjectSummaries { get; set; } = new();

public int TotalStudents { get; set; }

public decimal OverallAverage { get; set; }

}

public class SubjectResultSummaryDto

{

public SubjectDto Subject { get; set; } = null!;

public int TotalStudents { get; set; }

public decimal AverageScore { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public int PassCount { get; set; }

public int FailCount { get; set; }

public decimal PassRate { get; set; }

}

}

// DTOs/AttendanceDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class AttendanceDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public DateTime Date { get; set; }

public AttendanceStatus Status { get; set; }

public string Remarks { get; set; } = string.Empty;

}

public class CreateAttendanceDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid TeacherId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public DateTime Date { get; set; }

[Required]

public AttendanceStatus Status { get; set; }

[StringLength(200)]

public string Remarks { get; set; } = string.Empty;

}

public class UpdateAttendanceDto

{

public AttendanceStatus? Status { get; set; }

[StringLength(200)]

public string? Remarks { get; set; }

}

public class AttendanceReportDto

{

public StudentDto Student { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public List<AttendanceSummaryDto> SubjectAttendance { get; set; } = new();

public int TotalClassesTaken { get; set; }

public int TotalClassesAttended { get; set; }

public decimal OverallAttendancePercentage { get; set; }

}

public class AttendanceSummaryDto

{

public Guid SubjectId { get; set; }

public string SubjectName { get; set; } = string.Empty;

public int TotalClasses { get; set; }

public int PresentClasses { get; set; }

public int AbsentClasses { get; set; }

public int LateClasses { get; set; }

public decimal AttendancePercentage { get; set; }

}

}

// DTOs/TimetableDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class TimetableDto

{

public Guid Id { get; set; }

public SchoolDto School { get; set; } = null!;

public GradeDto Grade { get; set; } = null!;

public SchoolYearDto SchoolYear { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public string Name { get; set; } = string.Empty;

public bool IsActive { get; set; }

public bool IsGenerated { get; set; }

public DateTime GeneratedAt { get; set; }

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

}

public class TimetableSlotDto

{

public Guid Id { get; set; }

public SubjectDto Subject { get; set; } = null!;

public TeacherDto Teacher { get; set; } = null!;

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public string Room { get; set; } = string.Empty;

public int PeriodNumber { get; set; }

public string Notes { get; set; } = string.Empty;

}

public class GenerateTimetableDto

{

[Required]

public Guid SchoolId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

[StringLength(100)]

public string Name { get; set; } = string.Empty;

public TimetableGenerationRulesDto Rules { get; set; } = new();

}

public class TimetableGenerationRulesDto

{

public List<DayOfWeek> WorkingDays { get; set; } = new()

{

DayOfWeek.Monday, DayOfWeek.Tuesday, DayOfWeek.Wednesday,

DayOfWeek.Thursday, DayOfWeek.Friday

};

public TimeSpan SchoolStartTime { get; set; } = new(8, 0, 0);

public TimeSpan SchoolEndTime { get; set; } = new(15, 0, 0);

public TimeSpan PeriodDuration { get; set; } = new(0, 40, 0);

public TimeSpan BreakDuration { get; set; } = new(0, 15, 0);

public TimeSpan LunchDuration { get; set; } = new(1, 0, 0);

public int MaxPeriodsPerDay { get; set; } = 8;

public int MaxConsecutivePeriods { get; set; } = 3;

// Subject-specific rules

public Dictionary<Guid, int> SubjectPeriodsPerWeek { get; set; } = new();

public Dictionary<Guid, List<DayOfWeek>> SubjectPreferredDays { get; set; } = new();

public Dictionary<Guid, List<int>> SubjectPreferredPeriods { get; set; } = new();

// Teacher constraints

public Dictionary<Guid, List<TimeSlotDto>> TeacherUnavailableSlots { get; set; } = new();

// Room assignments

public Dictionary<Guid, string> SubjectRoomPreferences { get; set; } = new();

}

public class TimeSlotDto

{

public DayOfWeek DayOfWeek { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

}

public class UpdateTimetableSlotDto

{

public Guid? SubjectId { get; set; }

public Guid? TeacherId { get; set; }

public TimeSpan? StartTime { get; set; }

public TimeSpan? EndTime { get; set; }

public string? Room { get; set; }

public string? Notes { get; set; }

}

public class TimetableConflictDto

{

public bool HasConflicts { get; set; }

public List<ConflictDto> Conflicts { get; set; } = new();

}

public class ConflictDto

{

public ConflictType Type { get; set; }

public string Description { get; set; } = string.Empty;

public List<Guid> AffectedSlotIds { get; set; } = new();

public string Severity { get; set; } = string.Empty;

}

public enum ConflictType

{

TeacherDoubleBooking,

RoomDoubleBooking,

SubjectOverallocation,

InvalidTimeSlot,

ConsecutivePeriodViolation

}

}

// DTOs/NotificationDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class NotificationDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Message { get; set; } = string.Empty;

public NotificationType Type { get; set; }

public NotificationChannel Channel { get; set; }

public bool IsRead { get; set; }

public bool IsSent { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? SentAt { get; set; }

public DateTime? ReadAt { get; set; }

public string ExternalId { get; set; } = string.Empty;

public ParentDto? Parent { get; set; }

public StudentDto? Student { get; set; }

public TeacherDto? Teacher { get; set; }

}

public class CreateNotificationDto

{

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[Required]

public string Message { get; set; } = string.Empty;

[Required]

public NotificationType Type { get; set; }

[Required]

public NotificationChannel Channel { get; set; }

public Guid? ParentId { get; set; }

public Guid? StudentId { get; set; }

public Guid? TeacherId { get; set; }

public string Metadata { get; set; } = string.Empty;

}

public class NotificationStatsDto

{

public int TotalNotifications { get; set; }

public int SentNotifications { get; set; }

public int PendingNotifications { get; set; }

public int ReadNotifications { get; set; }

public Dictionary<NotificationType, int> NotificationsByType { get; set; } = new();

public Dictionary<NotificationChannel, int> NotificationsByChannel { get; set; } = new();

}

}

// DTOs/HandwritingAIDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ProcessHandwritingDto

{

[Required]

public Guid StudentId { get; set; }

[Required]

public string ImagePath { get; set; } = string.Empty;

public Guid? SubjectId { get; set; }

public Guid? ResultId { get; set; }

public string? ExpectedAnswer { get; set; }

}

public class HandwritingRecognitionResultDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string RecognizedText { get; set; } = string.Empty;

public decimal OverallConfidence { get; set; }

public bool ProcessedLocally { get; set; }

public bool ProcessedInCloud { get; set; }

public TimeSpan ProcessingTime { get; set; }

public string ExpectedAnswer { get; set; } = string.Empty;

public bool IsCorrect { get; set; }

public decimal PartialCreditScore { get; set; }

public DateTime ProcessedAt { get; set; }

public List<string> AlternativeTexts { get; set; } = new();

public List<decimal> CharacterConfidences { get; set; } = new();

}

public class GenerateTrainingExerciseDto

{

[Required]

public Guid SchoolId { get; set; }

public Guid? GradeId { get; set; }

public Guid? SubjectId { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

[Required]

public ExerciseType Type { get; set; }

[Required]

public DifficultyLevel Difficulty { get; set; }

public int EstimatedMinutes { get; set; } = 10;

}

public class TrainingExerciseDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public ExerciseType Type { get; set; }

public DifficultyLevel Difficulty { get; set; }

public bool IsActive { get; set; }

public bool IsSystemGenerated { get; set; }

public int EstimatedMinutes { get; set; }

public ExerciseContentDto Content { get; set; } = new();

public DateTime CreatedAt { get; set; }

}

public class StudentTrainingSessionDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingExerciseDto TrainingExercise { get; set; } = null!;

public DateTime StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public SessionStatus Status { get; set; }

public int TotalItems { get; set; }

public int CompletedItems { get; set; }

public int CorrectItems { get; set; }

public decimal OverallAccuracy { get; set; }

public TimeSpan TotalTime { get; set; }

}

public class SubmitTrainingDataDto

{

[Required]

public Guid SessionId { get; set; }

[Required]

public string OriginalContent { get; set; } = string.Empty;

[Required]

public byte[] HandwritingImage { get; set; } = Array.Empty<byte>();

}

public class HandwritingTrainingDataDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public TrainingDataType DataType { get; set; }

public string OriginalContent { get; set; } = string.Empty;

public string ImagePath { get; set; } = string.Empty;

public string ProcessedText { get; set; } = string.Empty;

public string VerifiedText { get; set; } = string.Empty;

public decimal ConfidenceScore { get; set; }

public TrainingStatus Status { get; set; }

public bool IsVerified { get; set; }

public bool IsTrainingComplete { get; set; }

public DateTime CapturedAt { get; set; }

public DateTime? VerifiedAt { get; set; }

public DateTime? TrainedAt { get; set; }

public StudentDto Student { get; set; } = null!;

}

public class ModelTrainingJobDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string JobName { get; set; } = string.Empty;

public JobType Type { get; set; }

public JobStatus Status { get; set; }

public int TotalDataPoints { get; set; }

public int ProcessedDataPoints { get; set; }

public decimal Progress { get; set; }

public DateTime QueuedAt { get; set; }

public DateTime? StartedAt { get; set; }

public DateTime? CompletedAt { get; set; }

public string ErrorMessage { get; set; } = string.Empty;

public string ResultPath { get; set; } = string.Empty;

}

public class HandwritingModelDto

{

public Guid Id { get; set; }

public Guid StudentId { get; set; }

public string ModelName { get; set; } = string.Empty;

public string Version { get; set; } = string.Empty;

public ModelType Type { get; set; }

public decimal Accuracy { get; set; }

public int TrainingDataCount { get; set; }

public bool IsActive { get; set; }// Services/Implementations/TimetableService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class TimetableService : ITimetableService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly ILogger<TimetableService> \_logger;

private readonly IPdfGenerationService \_pdfService;

public TimetableService(

SchoolManagementDbContext context,

IMapper mapper,

ILogger<TimetableService> logger,

IPdfGenerationService pdfService)

{

\_context = context;

\_mapper = mapper;

\_logger = logger;

\_pdfService = pdfService;

}

public async Task<TimetableDto> GenerateTimetableAsync(GenerateTimetableDto generateDto)

{

try

{

\_logger.LogInformation("Starting timetable generation for Grade {GradeId}", generateDto.GradeId);

// Validate input data

await ValidateGenerationInputAsync(generateDto);

// Get required data

var gradeSubjects = await GetGradeSubjectsAsync(generateDto.GradeId, generateDto.SchoolYearId);

var teachers = await GetAvailableTeachersAsync(generateDto.SchoolId, generateDto.SchoolYearId);

// Create timetable entity

var timetable = new Timetable

{

SchoolId = generateDto.SchoolId,

GradeId = generateDto.GradeId,

SchoolYearId = generateDto.SchoolYearId,

TermId = generateDto.TermId,

Name = generateDto.Name,

IsActive = true,

IsGenerated = true,

GeneratedAt = DateTime.UtcNow

};

\_context.Timetables.Add(timetable);

await \_context.SaveChangesAsync();

// Generate timetable slots

var generatedSlots = await GenerateTimetableSlotsAsync(

timetable.Id,

gradeSubjects,

teachers,

generateDto.Rules);

timetable.TimetableSlots = generatedSlots;

await \_context.SaveChangesAsync();

// Validate the generated timetable

var conflicts = await CheckTimetableConflictsAsync(timetable.Id);

if (conflicts.HasConflicts)

{

\_logger.LogWarning("Generated timetable has conflicts: {ConflictCount}", conflicts.Conflicts.Count);

// Attempt to resolve conflicts

await ResolveConflictsAsync(timetable.Id, conflicts);

}

// Create student timetable assignments

await CreateStudentTimetableAssignmentsAsync(timetable.Id, generateDto.GradeId);

var result = await GetTimetableWithDetailsAsync(timetable.Id);

\_logger.LogInformation("Timetable generation completed for Grade {GradeId}", generateDto.GradeId);

return \_mapper.Map<TimetableDto>(result);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error generating timetable for Grade {GradeId}", generateDto.GradeId);

throw;

}

}

public async Task<TimetableDto> GetGradeTimetableAsync(Guid gradeId, Guid termId)

{

var timetable = await \_context.Timetables

.Include(t => t.School)

.Include(t => t.Grade)

.Include(t => t.SchoolYear)

.Include(t => t.Term)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Subject)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Teacher)

.FirstOrDefaultAsync(t => t.GradeId == gradeId && t.TermId == termId && t.IsActive);

if (timetable == null)

throw new NotFoundException($"No active timetable found for grade {gradeId} in term {termId}");

return \_mapper.Map<TimetableDto>(timetable);

}

public async Task<StudentTimetableDto> GetStudentTimetableAsync(Guid studentId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.School)

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

if (student == null)

throw new NotFoundException($"Student {studentId} not found");

var currentTerm = await \_context.Terms

.Where(t => t.SchoolYear.SchoolId == student.SchoolId && t.IsCurrent)

.FirstOrDefaultAsync();

if (currentTerm == null)

throw new NotFoundException("No current term found");

var timetable = await GetGradeTimetableAsync(student.CurrentGradeId, currentTerm.Id);

// Get student-specific assignments (for elective subjects)

var studentSpecificSlots = await \_context.StudentTimetables

.Where(st => st.StudentId == studentId && st.IsActive)

.Include(st => st.TimetableSlot)

.ThenInclude(ts => ts.Subject)

.Include(st => st.TimetableSlot)

.ThenInclude(ts => ts.Teacher)

.Select(st => st.TimetableSlot)

.ToListAsync();

// Filter timetable based on student's subjects

var studentSubjects = await \_context.StudentSubjects

.Where(ss => ss.StudentId == studentId && ss.SchoolYearId == timetable.SchoolYear.Id && ss.IsActive)

.Select(ss => ss.SubjectId)

.ToListAsync();

var filteredSlots = timetable.TimetableSlots

.Where(slot => studentSubjects.Contains(slot.Subject.Id))

.Concat(studentSpecificSlots.Select(slot => \_mapper.Map<TimetableSlotDto>(slot)))

.Distinct()

.OrderBy(slot => slot.DayOfWeek)

.ThenBy(slot => slot.PeriodNumber)

.ToList();

return new StudentTimetableDto

{

Student = \_mapper.Map<StudentDto>(student),

TimetableSlots = filteredSlots,

TimetableName = timetable.Name

};

}

public async Task<TeacherTimetableDto> GetTeacherTimetableAsync(Guid teacherId, Guid termId)

{

var teacher = await \_context.Teachers

.Include(t => t.School)

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

if (teacher == null)

throw new NotFoundException($"Teacher {teacherId} not found");

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

if (term == null)

throw new NotFoundException($"Term {termId} not found");

var teacherSlots = await \_context.TimetableSlots

.Where(ts => ts.TeacherId == teacherId && ts.Timetable.TermId == termId)

.Include(ts => ts.Subject)

.Include(ts => ts.Teacher)

.Include(ts => ts.Timetable)

.ThenInclude(t => t.Grade)

.OrderBy(ts => ts.DayOfWeek)

.ThenBy(ts => ts.PeriodNumber)

.ToListAsync();

return new TeacherTimetableDto

{

Teacher = \_mapper.Map<TeacherDto>(teacher),

Term = \_mapper.Map<TermDto>(term),

TimetableSlots = \_mapper.Map<List<TimetableSlotDto>>(teacherSlots)

};

}

public async Task<TimetableSlotDto> UpdateTimetableSlotAsync(Guid slotId, UpdateTimetableSlotDto updateDto)

{

var slot = await \_context.TimetableSlots.FindAsync(slotId);

if (slot == null)

throw new NotFoundException($"Timetable slot {slotId} not found");

// Check for conflicts before updating

if (updateDto.TeacherId.HasValue || updateDto.StartTime.HasValue || updateDto.EndTime.HasValue)

{

var tempSlot = new TimetableSlot

{

Id = slot.Id,

TimetableId = slot.TimetableId,

TeacherId = updateDto.TeacherId ?? slot.TeacherId,

SubjectId = updateDto.SubjectId ?? slot.SubjectId,

DayOfWeek = slot.DayOfWeek,

StartTime = updateDto.StartTime ?? slot.StartTime,

EndTime = updateDto.EndTime ?? slot.EndTime,

Room = updateDto.Room ?? slot.Room,

PeriodNumber = slot.PeriodNumber

};

var conflicts = await ValidateSlotConflictsAsync(tempSlot);

if (conflicts.Any())

{

throw new ConflictException($"Update would create conflicts: {string.Join(", ", conflicts)}");

}

}

\_mapper.Map(updateDto, slot);

await \_context.SaveChangesAsync();

return \_mapper.Map<TimetableSlotDto>(slot);

}

public async Task<bool> ValidateTimetableAsync(Guid timetableId)

{

var conflicts = await CheckTimetableConflictsAsync(timetableId);

return !conflicts.HasConflicts;

}

public async Task<TimetableConflictDto> CheckTimetableConflictsAsync(Guid timetableId)

{

var conflicts = new List<ConflictDto>();

var slots = await \_context.TimetableSlots

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

.Include(ts => ts.Teacher)

.Include(ts => ts.Subject)

.ToListAsync();

// Check teacher double booking

var teacherConflicts = slots

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

.Where(g => g.Count() > 1)

.Select(g => new ConflictDto

{

Type = ConflictType.TeacherDoubleBooking,

Description = $"Teacher {g.Key.TeacherId} is double-booked on {g.Key.DayOfWeek} from {g.Key.StartTime} to {g.Key.EndTime}",

AffectedSlotIds = g.Select(s => s.Id).ToList(),

Severity = "High"

});

conflicts.AddRange(teacherConflicts);

// Check room conflicts

var roomConflicts = slots

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

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

.Where(g => g.Count() > 1)

.Select(g => new ConflictDto

{

Type = ConflictType.RoomDoubleBooking,

Description = $"Room {g.Key.Room} is double-booked on {g.Key.DayOfWeek} from {g.Key.StartTime} to {g.Key.EndTime}",

AffectedSlotIds = g.Select(s => s.Id).ToList(),

Severity = "Medium"

});

conflicts.AddRange(roomConflicts);

// Check consecutive period violations

var consecutiveViolations = CheckConsecutivePeriodViolations(slots);

conflicts.AddRange(consecutiveViolations);

return new TimetableConflictDto

{

HasConflicts = conflicts.Any(),

Conflicts = conflicts

};

}

public async Task<byte[]> PrintTimetableAsync(Guid timetableId)

{

var timetable = await GetTimetableWithDetailsAsync(timetableId);

var timetableDto = \_mapper.Map<TimetableDto>(timetable);

return await \_pdfService.GenerateTimetablePdfAsync(timetableDto);

}

// Private helper methods

private async Task ValidateGenerationInputAsync(GenerateTimetableDto generateDto)

{

var school = await \_context.Schools.FindAsync(generateDto.SchoolId);

if (school == null)

throw new NotFoundException($"School {generateDto.SchoolId} not found");

var grade = await \_context.Grades.FindAsync(generateDto.GradeId);

if (grade == null)

throw new NotFoundException($"Grade {generateDto.GradeId} not found");

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

if (term == null)

throw new NotFoundException($"Term {generateDto.TermId} not found");

// Check if timetable already exists

var existingTimetable = await \_context.Timetables

.FirstOrDefaultAsync(t => t.GradeId == generateDto.GradeId &&

t.TermId == generateDto.TermId &&

t.IsActive);

if (existingTimetable != null)

{

throw new ConflictException($"Active timetable already exists for this grade and term");

}

}

private async Task<List<GradeSubject>> GetGradeSubjectsAsync(Guid gradeId, Guid schoolYearId)

{

return await \_context.GradeSubjects

.Where(gs => gs.GradeId == gradeId)

.Include(gs => gs.Subject)

.Include(gs => gs.Grade)

.ToListAsync();

}

private async Task<List<Teacher>> GetAvailableTeachersAsync(Guid schoolId, Guid schoolYearId)

{

return await \_context.Teachers

.Where(t => t.SchoolId == schoolId)

.Include(t => t.SubjectTeachers.Where(st => st.SchoolYearId == schoolYearId))

.ThenInclude(st => st.Subject)

.ToListAsync();

}

private async Task<List<TimetableSlot>> GenerateTimetableSlotsAsync(

Guid timetableId,

List<GradeSubject> gradeSubjects,

List<Teacher> teachers,

TimetableGenerationRulesDto rules)

{

var slots = new List<TimetableSlot>();

var random = new Random();

// Calculate periods per day

var totalMinutesPerDay = (int)(rules.SchoolEndTime - rules.SchoolStartTime).TotalMinutes;

var breakMinutes = (int)rules.BreakDuration.TotalMinutes;

var lunchMinutes = (int)rules.LunchDuration.TotalMinutes;

var periodMinutes = (int)rules.PeriodDuration.TotalMinutes;

var availableMinutes = totalMinutesPerDay - breakMinutes - lunchMinutes;

var periodsPerDay = Math.Min(availableMinutes / periodMinutes, rules.MaxPeriodsPerDay);

// Generate time slots for each day

var timeSlots = GenerateTimeSlots(rules, periodsPerDay);

foreach (var gradeSubject in gradeSubjects)

{

// Get assigned teacher for this subject

var assignedTeacher = teachers

.FirstOrDefault(t => t.SubjectTeachers

.Any(st => st.SubjectId == gradeSubject.SubjectId));

if (assignedTeacher == null)

{

\_logger.LogWarning("No teacher assigned to subject {SubjectId}", gradeSubject.SubjectId);

continue;

}

// Determine periods per week for this subject

var periodsPerWeek = rules.SubjectPeriodsPerWeek.GetValueOrDefault(gradeSubject.SubjectId, 3);

// Get preferred days and periods for this subject

var preferredDays = rules.SubjectPreferredDays.GetValueOrDefault(

gradeSubject.SubjectId,

rules.WorkingDays);

var preferredPeriods = rules.SubjectPreferredPeriods.GetValueOrDefault(

gradeSubject.SubjectId,

Enumerable.Range(1, periodsPerDay).ToList());

// Schedule periods for this subject

var scheduledPeriods = 0;

var attempts = 0;

const int maxAttempts = 100;

while (scheduledPeriods < periodsPerWeek && attempts < maxAttempts)

{

attempts++;

// Select random day and period from preferred options

var day = preferredDays[random.Next(preferredDays.Count)];

var period = preferredPeriods[random.Next(preferredPeriods.Count)];

var timeSlot = timeSlots[day][period - 1];

// Check if slot is available

if (IsSlotAvailable(slots, assignedTeacher.Id, day, timeSlot.StartTime, timeSlot.EndTime))

{

var room = rules.SubjectRoomPreferences.GetValueOrDefault(gradeSubject.SubjectId, "");

var slot = new TimetableSlot

{

TimetableId = timetableId,

SubjectId = gradeSubject.SubjectId,

TeacherId = assignedTeacher.Id,

DayOfWeek = day,

StartTime = timeSlot.StartTime,

EndTime = timeSlot.EndTime,

Room = room,

PeriodNumber = period

};

slots.Add(slot);

scheduledPeriods++;

}

}

if (scheduledPeriods < periodsPerWeek)

{

\_logger.LogWarning("Could only schedule {Scheduled}/{Required} periods for subject {SubjectId}",

scheduledPeriods, periodsPerWeek, gradeSubject.SubjectId);

}

}

return slots;

}

private Dictionary<DayOfWeek, List<TimeSlotDto>> GenerateTimeSlots(

TimetableGenerationRulesDto rules,

int periodsPerDay)

{

var timeSlots = new Dictionary<DayOfWeek, List<TimeSlotDto>>();

foreach (var day in rules.WorkingDays)

{

var daySlots = new List<TimeSlotDto>();

var currentTime = rules.SchoolStartTime;

for (int period = 1; period <= periodsPerDay; period++)

{

// Add break after certain periods

if (period == 3) // After 2nd period

{

currentTime = currentTime.Add(rules.BreakDuration);

}

else if (period == 6) // After 5th period (lunch)

{

currentTime = currentTime.Add(rules.LunchDuration);

}

var slot = new TimeSlotDto

{

DayOfWeek = day,

StartTime = currentTime,

EndTime = currentTime.Add(rules.PeriodDuration)

};

daySlots.Add(slot);

currentTime = slot.EndTime;

}

timeSlots[day] = daySlots;

}

return timeSlots;

}

private bool IsSlotAvailable(

List<TimetableSlot> existingSlots,

Guid teacherId,

DayOfWeek day,

TimeSpan startTime,

TimeSpan endTime)

{

return !existingSlots.Any(slot =>

slot.TeacherId == teacherId &&

slot.DayOfWeek == day &&

((startTime >= slot.StartTime && startTime < slot.EndTime) ||

(endTime > slot.StartTime && endTime <= slot.EndTime) ||

(startTime <= slot.StartTime && endTime >= slot.EndTime)));

}

private async Task<List<string>> ValidateSlotConflictsAsync(TimetableSlot slot)

{

var conflicts = new List<string>();

// Check teacher conflicts

var teacherConflict = await \_context.TimetableSlots

.AnyAsync(ts => ts.Id != slot.Id &&

ts.TeacherId == slot.TeacherId &&

ts.DayOfWeek == slot.DayOfWeek &&

((slot.StartTime >= ts.StartTime && slot.StartTime < ts.EndTime) ||

(slot.EndTime > ts.StartTime && slot.EndTime <= ts.EndTime) ||

(slot.StartTime <= ts.StartTime && slot.EndTime >= ts.EndTime)));

if (teacherConflict)

conflicts.Add("Teacher scheduling conflict");

// Check room conflicts

if (!string.IsNullOrEmpty(slot.Room))

{

var roomConflict = await \_context.TimetableSlots

.AnyAsync(ts => ts.Id != slot.Id &&

ts.Room == slot.Room &&

ts.DayOfWeek == slot.DayOfWeek &&

((slot.StartTime >= ts.StartTime && slot.StartTime < ts.EndTime) ||

(slot.EndTime > ts.StartTime && slot.EndTime <= ts.EndTime) ||

(slot.StartTime <= ts.StartTime && slot.EndTime >= ts.EndTime)));

if (roomConflict)

conflicts.Add("Room scheduling conflict");

}

return conflicts;

}

private List<ConflictDto> CheckConsecutivePeriodViolations(List<TimetableSlot> slots)

{

var violations = new List<ConflictDto>();

const int maxConsecutive = 3; // Maximum consecutive periods

var teacherSlots = slots

.GroupBy(s => s.TeacherId)

.ToDictionary(g => g.Key, g => g.ToList());

foreach (var teacherGroup in teacherSlots)

{

var teacherId = teacherGroup.Key;

var teacherPeriods = teacherGroup.Value

.GroupBy(s => s.DayOfWeek)

.ToDictionary(g => g.Key, g => g.OrderBy(s => s.PeriodNumber).ToList());

foreach (var dayGroup in teacherPeriods)

{

var dayPeriods = dayGroup.Value;

var consecutiveCount = 1;

for (int i = 1; i < dayPeriods.Count; i++)

{

if (dayPeriods[i].PeriodNumber == dayPeriods[i - 1].PeriodNumber + 1)

{

consecutiveCount++;

if (consecutiveCount > maxConsecutive)

{

violations.Add(new ConflictDto

{

Type = ConflictType.ConsecutivePeriodViolation,

Description = $"Teacher has {consecutiveCount} consecutive periods on {dayGroup.Key}",

AffectedSlotIds = dayPeriods.Skip(i - consecutiveCount + 1).Take(consecutiveCount).Select(s => s.Id).ToList(),

Severity = "Medium"

});

break;

}

}

else

{

consecutiveCount = 1;

}

}

}

}

return violations;

}

private async Task ResolveConflictsAsync(Guid timetableId, TimetableConflictDto conflicts)

{

// Implement basic conflict resolution logic

foreach (var conflict in conflicts.Conflicts.Where(c => c.Severity == "High"))

{

try

{

await AttemptConflictResolutionAsync(conflict);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to resolve conflict: {ConflictDescription}", conflict.Description);

}

}

}

private async Task AttemptConflictResolutionAsync(ConflictDto conflict)

{

if (conflict.Type == ConflictType.TeacherDoubleBooking)

{

// Try to reschedule one of the conflicting slots

var conflictingSlots = await \_context.TimetableSlots

.Where(ts => conflict.AffectedSlotIds.Contains(ts.Id))

.ToListAsync();

if (conflictingSlots.Count >= 2)

{

var slotToReschedule = conflictingSlots.Last(); // Reschedule the last one

await TryRescheduleSlotAsync(slotToReschedule);

}

}

}

private async Task<bool> TryRescheduleSlotAsync(TimetableSlot slot)

{

var timetable = await \_context.Timetables.FindAsync(slot.TimetableId);

if (timetable == null) return false;

// Try to find an alternative time slot

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

var periods = Enumerable.Range(1, 8).ToList();

foreach (var day in workingDays)

{

foreach (var period in periods)

{

var startTime = new TimeSpan(8, 0, 0).Add(TimeSpan.FromMinutes((period - 1) \* 45)); // 45-minute periods

var endTime = startTime.Add(TimeSpan.FromMinutes(40));

var tempSlot = new TimetableSlot

{

Id = slot.Id,

TimetableId = slot.TimetableId,

TeacherId = slot.TeacherId,

SubjectId = slot.SubjectId,

DayOfWeek = day,

StartTime = startTime,

EndTime = endTime,

Room = slot.Room,

PeriodNumber = period

};

var conflicts = await ValidateSlotConflictsAsync(tempSlot);

if (!conflicts.Any())

{

// Update the slot

slot.DayOfWeek = day;

slot.StartTime = startTime;

slot.EndTime = endTime;

slot.PeriodNumber = period;

await \_context.SaveChangesAsync();

return true;

}

}

}

return false;

}

private async Task CreateStudentTimetableAssignmentsAsync(Guid timetableId, Guid gradeId)

{

var students = await \_context.Students

.Where(s => s.CurrentGradeId == gradeId)

.ToListAsync();

var timetableSlots = await \_context.TimetableSlots

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

.ToListAsync();

var studentTimetables = new List<StudentTimetable>();

foreach (var student in students)

{

// Get student's enrolled subjects

var studentSubjects = await \_context.StudentSubjects

.Where(ss => ss.StudentId == student.Id && ss.IsActive)

.Select(ss => ss.SubjectId)

.ToListAsync();

// Assign relevant timetable slots to student

var relevantSlots = timetableSlots

.Where(slot => studentSubjects.Contains(slot.SubjectId))

.ToList();

foreach (var slot in relevantSlots)

{

studentTimetables.Add(new StudentTimetable

{

StudentId = student.Id,

TimetableSlotId = slot.Id,

IsActive = true

});

}

}

\_context.StudentTimetables.AddRange(studentTimetables);

await \_context.SaveChangesAsync();

}

private async Task<Timetable> GetTimetableWithDetailsAsync(Guid timetableId)

{

return await \_context.Timetables

.Include(t => t.School)

.Include(t => t.Grade)

.Include(t => t.SchoolYear)

.Include(t => t.Term)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Subject)

.Include(t => t.TimetableSlots)

.ThenInclude(ts => ts.Teacher)

.FirstOrDefaultAsync(t => t.Id == timetableId)

?? throw new NotFoundException($"Timetable {timetableId} not found");

}

}

}// Services/Implementations/GradingService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class GradingService : IGradingService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

public GradingService(SchoolManagementDbContext context, IMapper mapper)

{

\_context = context;

\_mapper = mapper;

}

public async Task<IEnumerable<GradingSchemeDto>> GetAllGradingSchemesAsync()

{

var schemes = await \_context.GradingSchemes

.Include(gs => gs.GradingScales.OrderBy(scale => scale.SortOrder))

.Where(gs => gs.IsActive)

.ToListAsync();

return \_mapper.Map<IEnumerable<GradingSchemeDto>>(schemes);

}

public async Task<GradingSchemeDto> CreateGradingSchemeAsync(CreateGradingSchemeDto createDto)

{

var scheme = \_mapper.Map<GradingScheme>(createDto);

\_context.GradingSchemes.Add(scheme);

await \_context.SaveChangesAsync();

// Add grading scales

if (createDto.GradingScales.Any())

{

var scales = createDto.GradingScales.Select(scaleDto =>

{

var scale = \_mapper.Map<GradingScale>(scaleDto);

scale.GradingSchemeId = scheme.Id;

return scale;

}).ToList();

\_context.GradingScales.AddRange(scales);

await \_context.SaveChangesAsync();

}

return await GetGradingSchemeByIdAsync(scheme.Id);

}

public async Task<GradingSchemeDto> UpdateGradingSchemeAsync(Guid schemeId, UpdateGradingSchemeDto updateDto)

{

var scheme = await \_context.GradingSchemes.FindAsync(schemeId);

if (scheme == null)

throw new NotFoundException($"Grading scheme {schemeId} not found");

\_mapper.Map(updateDto, scheme);

await \_context.SaveChangesAsync();

return await GetGradingSchemeByIdAsync(schemeId);

}

public async Task<bool> DeleteGradingSchemeAsync(Guid schemeId)

{

var scheme = await \_context.GradingSchemes.FindAsync(schemeId);

if (scheme == null)

return false;

// Check if scheme is being used by any schools

var isInUse = await \_context.Schools.AnyAsync(s => s.GradingSchemeId == schemeId);

if (isInUse)

{

throw new ConflictException("Cannot delete grading scheme that is in use by schools");

}

scheme.IsActive = false;

await \_context.SaveChangesAsync();

return true;

}

public async Task<GradingScaleDto> AddGradingScaleAsync(Guid schemeId, CreateGradingScaleDto createDto)

{

var scheme = await \_context.GradingSchemes.FindAsync(schemeId);

if (scheme == null)

throw new NotFoundException($"Grading scheme {schemeId} not found");

var scale = \_mapper.Map<GradingScale>(createDto);

scale.GradingSchemeId = schemeId;

\_context.GradingScales.Add(scale);

await \_context.SaveChangesAsync();

return \_mapper.Map<GradingScaleDto>(scale);

}

public async Task<GradingScaleDto> UpdateGradingScaleAsync(Guid scaleId, UpdateGradingScaleDto updateDto)

{

var scale = await \_context.GradingScales.FindAsync(scaleId);

if (scale == null)

throw new NotFoundException($"Grading scale {scaleId} not found");

\_mapper.Map(updateDto, scale);

await \_context.SaveChangesAsync();

return \_mapper.Map<GradingScaleDto>(scale);

}

public async Task<bool> DeleteGradingScaleAsync(Guid scaleId)

{

var scale = await \_context.GradingScales.FindAsync(scaleId);

if (scale == null)

return false;

\_context.GradingScales.Remove(scale);

await \_context.SaveChangesAsync();

return true;

}

public async Task<GradeCalculationDto> CalculateGradeAsync(Guid gradingSchemeId, decimal percentage)

{

var gradingScale = await \_context.GradingScales

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

percentage >= gs.MinPercentage &&

percentage <= gs.MaxPercentage)

.FirstOrDefaultAsync();

if (gradingScale == null)

{

// Return default grade if no matching scale found

return new GradeCalculationDto

{

Symbol = "F",

Unit = 0,

Description = "Fail",

Percentage = percentage

};

}

return new GradeCalculationDto

{

Symbol = gradingScale.Symbol,

Unit = gradingScale.Unit,

Description = gradingScale.Description,

Percentage = percentage

};

}

public async Task<IEnumerable<GradingScaleDto>> GetGradingScalesAsync(Guid schemeId)

{

var scales = await \_context.GradingScales

.Where(gs => gs.GradingSchemeId == schemeId)

.OrderBy(gs => gs.SortOrder)

.ToListAsync();

return \_mapper.Map<IEnumerable<GradingScaleDto>>(scales);

}

private async Task<GradingSchemeDto> GetGradingSchemeByIdAsync(Guid schemeId)

{

var scheme = await \_context.GradingSchemes

.Include(gs => gs.GradingScales.OrderBy(scale => scale.SortOrder))

.FirstOrDefaultAsync(gs => gs.Id == schemeId);

if (scheme == null)

throw new NotFoundException($"Grading scheme {schemeId} not found");

return \_mapper.Map<GradingSchemeDto>(scheme);

}

}

}

// Services/Implementations/AttendanceService.cs

namespace SchoolManagement.Services.Implementations

{

public class AttendanceService : IAttendanceService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly INotificationService \_notificationService;

public AttendanceService(

SchoolManagementDbContext context,

IMapper mapper,

INotificationService notificationService)

{

\_context = context;

\_mapper = mapper;

\_notificationService = notificationService;

}

public async Task<AttendanceDto> MarkAttendanceAsync(CreateAttendanceDto createAttendanceDto)

{

// Check if attendance already exists for this date

var existingAttendance = await \_context.Attendances

.FirstOrDefaultAsync(a => a.StudentId == createAttendanceDto.StudentId &&

a.SubjectId == createAttendanceDto.SubjectId &&

a.Date.Date == createAttendanceDto.Date.Date);

if (existingAttendance != null)

{

// Update existing attendance

existingAttendance.Status = createAttendanceDto.Status;

existingAttendance.Remarks = createAttendanceDto.Remarks;

await \_context.SaveChangesAsync();

return \_mapper.Map<AttendanceDto>(existingAttendance);

}

// Create new attendance record

var attendance = \_mapper.Map<Attendance>(createAttendanceDto);

\_context.Attendances.Add(attendance);

await \_context.SaveChangesAsync();

// Send notification to parents if absent

if (attendance.Status == AttendanceStatus.Absent)

{

await NotifyParentsAboutAbsenceAsync(attendance);

}

return \_mapper.Map<AttendanceDto>(attendance);

}

public async Task<IEnumerable<AttendanceDto>> GetAttendanceByClassAsync(Guid gradeId, DateTime date)

{

var attendances = await \_context.Attendances

.Where(a => a.Student.CurrentGradeId == gradeId && a.Date.Date == date.Date)

.Include(a => a.Student)

.Include(a => a.Subject)

.Include(a => a.Teacher)

.Include(a => a.Term)

.ToListAsync();

return \_mapper.Map<IEnumerable<AttendanceDto>>(attendances);

}

public async Task<IEnumerable<AttendanceDto>> GetStudentAttendanceAsync(Guid studentId, Guid termId)

{

var attendances = await \_context.Attendances

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

.Include(a => a.Student)

.Include(a => a.Subject)

.Include(a => a.Teacher)

.Include(a => a.Term)

.OrderBy(a => a.Date)

.ToListAsync();

return \_mapper.Map<IEnumerable<AttendanceDto>>(attendances);

}

public async Task<AttendanceReportDto> GetAttendanceReportAsync(Guid studentId, Guid termId)

{

var student = await \_context.Students

.Include(s => s.CurrentGrade)

.Include(s => s.School)

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

if (student == null)

throw new NotFoundException($"Student {studentId} not found");

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

if (term == null)

throw new NotFoundException($"Term {termId} not found");

var attendances = await GetStudentAttendanceAsync(studentId, termId);

var subjectAttendance = attendances

.GroupBy(a => new { a.Subject.Id, a.Subject.Name })

.Select(g => new AttendanceSummaryDto

{

SubjectId = g.Key.Id,

SubjectName = g.Key.Name,

TotalClasses = g.Count(),

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

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

LateClasses = g.Count(a => a.Status == AttendanceStatus.Late),

AttendancePercentage = g.Count() > 0 ?

(decimal)g.Count(a => a.Status == AttendanceStatus.Present) / g.Count() \* 100 : 0

})

.ToList();

var totalClasses = subjectAttendance.Sum(s => s.TotalClasses);

var totalPresent = subjectAttendance.Sum(s => s.PresentClasses);

return new AttendanceReportDto

{

Student = \_mapper.Map<StudentDto>(student),

Term = \_mapper.Map<TermDto>(term),

SubjectAttendance = subjectAttendance,

TotalClassesTaken = totalClasses,

TotalClassesAttended = totalPresent,

OverallAttendancePercentage = totalClasses > 0 ? (decimal)totalPresent / totalClasses \* 100 : 0

};

}

public async Task<bool> BulkMarkAttendanceAsync(List<CreateAttendanceDto> attendanceList)

{

var attendances = new List<Attendance>();

var notifications = new List<CreateNotificationDto>();

foreach (var createDto in attendanceList)

{

// Check for existing attendance

var existing = await \_context.Attendances

.FirstOrDefaultAsync(a => a.StudentId == createDto.StudentId &&

a.SubjectId == createDto.SubjectId &&

a.Date.Date == createDto.Date.Date);

if (existing != null)

{

existing.Status = createDto.Status;

existing.Remarks = createDto.Remarks;

}

else

{

var attendance = \_mapper.Map<Attendance>(createDto);

attendances.Add(attendance);

// Queue absence notifications

if (attendance.Status == AttendanceStatus.Absent)

{

var student = await \_context.Students

.Include(s => s.Parents)

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

if (student?.Parents.Any() == true)

{

foreach (var parent in student.Parents)

{

notifications.Add(new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = student.Id,

Title = "Student Absence",

Message = $"{student.FirstName} {student.LastName} was absent on {createDto.Date:dd/MM/yyyy}",

Type = NotificationType.Attendance,

Channel = NotificationChannel.WhatsApp

});

}

}

}

}

}

if (attendances.Any())

\_context.Attendances.AddRange(attendances);

await \_context.SaveChangesAsync();

// Send notifications

foreach (var notification in notifications)

{

try

{

await \_notificationService.CreateNotificationAsync(notification);

}

catch (Exception ex)

{

// Log but don't fail the attendance marking

// \_logger.LogError(ex, "Failed to send attendance notification");

}

}

return true;

}

public async Task<AttendanceDto> UpdateAttendanceAsync(Guid attendanceId, UpdateAttendanceDto updateDto)

{

var attendance = await \_context.Attendances.FindAsync(attendanceId);

if (attendance == null)

throw new NotFoundException($"Attendance record {attendanceId} not found");

\_mapper.Map(updateDto, attendance);

await \_context.SaveChangesAsync();

return \_mapper.Map<AttendanceDto>(attendance);

}

private async Task NotifyParentsAboutAbsenceAsync(Attendance attendance)

{

var student = await \_context.Students

.Include(s => s.Parents)

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

if (student?.Parents.Any() == true)

{

foreach (var parent in student.Parents)

{

var notification = new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = student.Id,

Title = "Student Absence Alert",

Message = $"Dear {parent.FirstName}, {student.FirstName} {student.LastName} was absent from school on {attendance.Date:dd/MM/yyyy}. Please contact the school if this absence was not planned.",

Type = NotificationType.Attendance,

Channel = NotificationChannel.WhatsApp

};

await \_notificationService.CreateNotificationAsync(notification);

}

}

}

}

}

// Services/Implementations/NotificationService.cs

namespace SchoolManagement.Services.Implementations

{

public class NotificationService : INotificationService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly ILogger<NotificationService> \_logger;

private readonly IWhatsAppService \_whatsAppService;

private readonly IEmailService \_emailService;

public NotificationService(

SchoolManagementDbContext context,

IMapper mapper,

ILogger<NotificationService> logger,

IWhatsAppService whatsAppService,

IEmailService emailService)

{

\_context = context;

\_mapper = mapper;

\_logger = logger;

\_whatsAppService = whatsAppService;

\_emailService = emailService;

}

public async Task<NotificationDto> CreateNotificationAsync(CreateNotificationDto createDto)

{

var notification = \_mapper.Map<Notification>(createDto);

\_context.Notifications.Add(notification);

await \_context.SaveChangesAsync();

// Auto-send the notification

await SendNotificationAsync(notification.Id);

return \_mapper.Map<NotificationDto>(notification);

}

public async Task<bool> SendNotificationAsync(Guid notificationId)

{

var notification = await \_context.Notifications

.Include(n => n.Parent)

.Include(n => n.Student)

.Include(n => n.Teacher)

.FirstOrDefaultAsync(n => n.Id == notificationId);

if (notification == null || notification.IsSent)

return false;

try

{

bool sent = false;

switch (notification.Channel)

{

case NotificationChannel.WhatsApp:

sent = await SendWhatsAppNotificationAsync(notification);

break;

case NotificationChannel.Email:

sent = await SendEmailNotificationAsync(notification);

break;

case NotificationChannel.SMS:

sent = await SendSMSNotificationAsync(notification);

break;

case NotificationChannel.InApp:

sent = true; // In-app notifications are always "sent"

break;

}

if (sent)

{

notification.IsSent = true;

notification.SentAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

}

return sent;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send notification {NotificationId}", notificationId);

return false;

}

}

public async Task<bool> SendBulkNotificationAsync(List<Guid> recipientIds, CreateNotificationDto notificationDto)

{

var notifications = new List<Notification>();

foreach (var recipientId in recipientIds)

{

var notification = \_mapper.Map<Notification>(notificationDto);

// Determine recipient type and set appropriate ID

if (notificationDto.ParentId.HasValue)

notification.ParentId = recipientId;

else if (notificationDto.StudentId.HasValue)

notification.StudentId = recipientId;

else if (notificationDto.TeacherId.HasValue)

notification.TeacherId = recipientId;

notifications.Add(notification);

}

\_context.Notifications.AddRange(notifications);

await \_context.SaveChangesAsync();

// Send all notifications

var sendTasks = notifications.Select(n => SendNotificationAsync(n.Id));

var results = await Task.WhenAll(sendTasks);

return results.All(r => r);

}

public async Task<IEnumerable<NotificationDto>> GetUserNotificationsAsync(Guid userId, string userType)

{

IQueryable<Notification> query = \_context.Notifications;

switch (userType.ToLower())

{

case "parent":

query = query.Where(n => n.ParentId == userId);

break;

case "student":

query = query.Where(n => n.StudentId == userId);

break;

case "teacher":

query = query.Where(n => n.TeacherId == userId);

break;

default:

throw new ArgumentException("Invalid user type");

}

var notifications = await query

.Include(n => n.Parent)

.Include(n => n.Student)

.Include(n => n.Teacher)

.OrderByDescending(n => n.CreatedAt)

.Take(50) // Limit for performance

.ToListAsync();

return \_mapper.Map<IEnumerable<NotificationDto>>(notifications);

}

public async Task<bool> MarkAsReadAsync(Guid notificationId)

{

var notification = await \_context.Notifications.FindAsync(notificationId);

if (notification == null)

return false;

if (!notification.IsRead)

{

notification.IsRead = true;

notification.ReadAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

}

return true;

}

public async Task<bool> SendWhatsAppMessageAsync(string phoneNumber, string message)

{

try

{

return await \_whatsAppService.SendMessageAsync(phoneNumber, message);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send WhatsApp message to {PhoneNumber}", phoneNumber);

return false;

}

}

public async Task<bool> SendEmailNotificationAsync(string email, string subject, string message)

{

try

{

return await \_emailService.SendEmailAsync(email, subject, message);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send email to {Email}", email);

return false;

}

}

public async Task<NotificationStatsDto> GetNotificationStatsAsync(Guid schoolId)

{

var notifications = await \_context.Notifications

.Where(n => (n.Student != null && n.Student.SchoolId == schoolId) ||

(n.Teacher != null && n.Teacher.SchoolId == schoolId))

.ToListAsync();

return new NotificationStatsDto

{

TotalNotifications = notifications.Count,

SentNotifications = notifications.Count(n => n.IsSent),

PendingNotifications = notifications.Count(n => !n.IsSent),

ReadNotifications = notifications.Count(n => n.IsRead),

NotificationsByType = notifications.GroupBy(n => n.Type)

.ToDictionary(g => g.Key, g => g.Count()),

NotificationsByChannel = notifications.GroupBy(n => n.Channel)

.ToDictionary(g => g.Key, g => g.Count())

};

}

private async Task<bool> SendWhatsAppNotificationAsync(Notification notification)

{

string phoneNumber = null;

if (notification.Parent != null && !string.IsNullOrEmpty(notification.Parent.WhatsAppNumber))

phoneNumber = notification.Parent.WhatsAppNumber;

else if (notification.Teacher != null && !string.IsNullOrEmpty(notification.Teacher.PhoneNumber))

phoneNumber = notification.Teacher.PhoneNumber;

if (string.IsNullOrEmpty(phoneNumber))

return false;

var success = await \_whatsAppService.SendMessageAsync(phoneNumber, notification.Message);

if (success)

{

notification.ExternalId = Guid.NewGuid().ToString(); // WhatsApp message ID would go here

}

return success;

}

private async Task<bool> SendEmailNotificationAsync(Notification notification)

{

string email = null;

if (notification.Parent != null && !string.IsNullOrEmpty(notification.Parent.Email))

email = notification.Parent.Email;

else if (notification.Teacher != null && !string.IsNullOrEmpty(notification.Teacher.Email))

email = notification.Teacher.Email;

if (string.IsNullOrEmpty(email))

return false;

return await \_emailService.SendEmailAsync(email, notification.Title, notification.Message);

}

private async Task<bool> SendSMSNotificationAsync(Notification notification)

{

string phoneNumber = null;

if (notification.Parent != null && !string.IsNullOrEmpty(notification.Parent.PhoneNumber))

phoneNumber = notification.Parent.PhoneNumber;

else if (notification.Teacher != null && !string.IsNullOrEmpty(notification.Teacher.PhoneNumber))

phoneNumber = notification.Teacher.PhoneNumber;

if (string.IsNullOrEmpty(phoneNumber))

return false;

// SMS service implementation would go here

await Task.Delay(100); // Placeholder

return true;

}

}

}

// Services/Interfaces/Supporting/IWhatsAppService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IWhatsAppService

{

Task<bool> SendMessageAsync(string phoneNumber, string message);

Task<bool> SendMediaMessageAsync(string phoneNumber, string mediaUrl, string caption);

Task<bool> SendTemplateMessageAsync(string phoneNumber, string templateName, Dictionary<string, string> parameters);

Task<List<WhatsAppMessageStatus>> GetMessageStatusAsync(List<string> messageIds);

}

public class WhatsAppMessageStatus

{

public string MessageId { get; set; } = string.Empty;

public string Status { get; set; } = string.Empty; // sent, delivered, read, failed

public DateTime Timestamp { get; set; }

}

}

// Services/Interfaces/Supporting/IEmailService.cs

namespace SchoolManagement.Services.Interfaces

{

public interface IEmailService

{

Task<bool> SendEmailAsync(string toEmail, string subject, string body);

Task<bool> SendEmailAsync(List<string> toEmails, string subject, string body);

Task<bool> SendEmailWithAttachmentAsync(string toEmail, string subject, string body, byte[] attachment, string attachmentName);

Task<bool> SendTemplateEmailAsync(string toEmail, string templateName, Dictionary<string, object> templateData);

}

}

// Services/Implementations/Supporting/WhatsAppService.cs (Basic Implementation)

namespace SchoolManagement.Services.Implementations

{

public class WhatsAppService : IWhatsAppService

{

private readonly HttpClient \_httpClient;

private readonly IConfiguration \_configuration;

private readonly ILogger<WhatsAppService> \_logger;

public WhatsAppService(

HttpClient httpClient,

IConfiguration configuration,

ILogger<WhatsAppService> logger)

{

\_httpClient = httpClient;

\_configuration = configuration;

\_logger = logger;

}

public async Task<bool> SendMessageAsync(string phoneNumber, string message)

{

try

{

// WhatsApp Business API implementation

var apiUrl = \_configuration["WhatsApp:ApiUrl"];

var accessToken = \_configuration["WhatsApp:AccessToken"];

var phoneNumberId = \_configuration["WhatsApp:PhoneNumberId"];

var payload = new

{

messaging\_product = "whatsapp",

to = phoneNumber,

type = "text",

text = new { body = message }

};

\_httpClient.DefaultRequestHeaders.Authorization =

new System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", accessToken);

var json = System.Text.Json.JsonSerializer.Serialize(payload);

var content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

var response = await \_httpClient.PostAsync($"{apiUrl}/{phoneNumberId}/messages", content);

if (response.IsSuccessStatusCode)

{

\_logger.LogInformation("WhatsApp message sent successfully to {PhoneNumber}", phoneNumber);

return true;

}

else

{

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

\_logger.LogError("Failed to send WhatsApp message to {PhoneNumber}. Error: {Error}", phoneNumber, error);

return false;

}

}

catch (Exception ex)

{

\_logger.LogError(ex, "Exception while sending WhatsApp message to {PhoneNumber}", phoneNumber);

return false;

}

}

public async Task<bool> SendMediaMessageAsync(string phoneNumber, string mediaUrl, string caption)

{

// Implementation for media messages

await Task.Delay(100); // Placeholder

return true;

}

public async Task<bool> SendTemplateMessageAsync(string phoneNumber, string templateName, Dictionary<string, string> parameters)

{

// Implementation for template messages

await Task.Delay(100); // Placeholder

return true;

}

public async Task<List<WhatsAppMessageStatus>> GetMessageStatusAsync(List<string> messageIds)

{

// Implementation for checking message status

await Task.Delay(100); // Placeholder

return new List<WhatsAppMessageStatus>();

}

}

}

// Services/Implementations/Supporting/EmailService.cs (Basic Implementation)

namespace SchoolManagement.Services.Implementations

{

public class EmailService : IEmailService

{

private readonly IConfiguration \_configuration;

private readonly ILogger<EmailService> \_logger;

public EmailService(IConfiguration configuration, ILogger<EmailService> logger)

{

\_configuration = configuration;

\_logger = logger;

}

public async Task<bool> SendEmailAsync(string toEmail, string subject, string body)

{

try

{

// SMTP configuration

var smtpHost = \_configuration["Email:SmtpHost"];

var smtpPort = int.Parse(\_configuration["Email:SmtpPort"] ?? "587");

var username = \_configuration["Email:Username"];

var password = \_configuration["Email:Password"];

var fromEmail = \_configuration["Email:FromEmail"];

using var client = new System.Net.Mail.SmtpClient(smtpHost, smtpPort)

{

Credentials = new System.Net.NetworkCredential(username, password),

EnableSsl = true

};

var message = new System.Net.Mail.MailMessage(fromEmail, toEmail, subject, body)

{

IsBodyHtml = true

};

await client.SendMailAsync(message);

\_logger.LogInformation("Email sent successfully to {ToEmail}", toEmail);

return true;

}

catch (Exception ex)

{

\_logger.LogError(ex, "Failed to send email to {ToEmail}", toEmail);

return false;

}

}

public async Task<bool> SendEmailAsync(List<string> toEmails, string subject, string body)

{

var tasks = toEmails.Select(email => SendEmailAsync(email, subject, body));

var results = await Task.WhenAll(tasks);

return results.All(r => r);

}

public async Task<bool> SendEmailWithAttachmentAsync(string toEmail, string subject, string body, byte[] attachment, string attachmentName)

{

// Implementation with attachment support

await Task.Delay(100); // Placeholder

return true;

}

public async Task<bool> SendTemplateEmailAsync(string toEmail, string templateName, Dictionary<string, object> templateData)

{

// Template email implementation

await Task.Delay(100); // Placeholder

return true;

}

}

}

// Program.cs - Dependency Injection Setup

namespace SchoolManagement.API

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

// Add services to the container

builder.Services.AddDbContext<SchoolManagementDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// AutoMapper

builder.Services.AddAutoMapper(typeof(AutoMapperProfile));

// Register services

builder.Services.AddScoped<ISchoolService, SchoolService>();

builder.Services.AddScoped<IStudentService, StudentService>();

builder.Services.AddScoped<ITeacherService, TeacherService>();

builder.Services.AddScoped<IResultService, ResultService>();

builder.Services.AddScoped<IAttendanceService, AttendanceService>();

builder.Services.AddScoped<IGradingService, GradingService>();

builder.Services.AddScoped<ITimetableService, TimetableService>();

builder.Services.AddScoped<INotificationService, NotificationService>();

builder.Services.AddScoped<IHandwritingRecognitionService, HandwritingRecognitionService>();

// Supporting services

builder.Services.AddScoped<IImageProcessingService, ImageProcessingService>();

builder.Services.AddScoped<IMachineLearningService, MachineLearningService>();

builder.Services.AddScoped<IFileStorageService, FileStorageService>();

builder.Services.AddScoped<IPdfGenerationService, PdfGenerationService>();

builder.Services.AddScoped<IWhatsAppService, WhatsAppService>();

builder.Services.AddScoped<IEmailService, EmailService>();

// HTTP Client for WhatsApp

builder.Services.AddHttpClient<IWhatsAppService, WhatsAppService>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

}

}

}// Models/Entities/Exam/Exam.cs

using System.ComponentModel.DataAnnotations;

namespace SchoolManagement.Core.Entities

{

public class Exam : BaseEntity

{

public Guid SchoolId { get; set; }

public virtual School School { get; set; } = null!;

public Guid SubjectId { get; set; }

public virtual Subject Subject { get; set; } = null!;

public Guid GradeId { get; set; }

public virtual Grade Grade { get; set; } = null!;

public Guid TermId { get; set; }

public virtual Term Term { get; set; } = null!;

public Guid SchoolYearId { get; set; }

public virtual SchoolYear SchoolYear { get; set; } = null!;

public Guid CreatedByTeacherId { get; set; }

public virtual Teacher CreatedByTeacher { get; set; } = null!;

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

public ExamType Type { get; set; }

public DateTime ExamDate { get; set; }

public TimeSpan Duration { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public decimal TotalMarks { get; set; }

public decimal PassingMarks { get; set; }

[StringLength(100)]

public string Venue { get; set; } = string.Empty;

public ExamStatus Status { get; set; } = ExamStatus.Draft;

public bool IsPublished { get; set; } = false;

public DateTime? PublishedAt { get; set; }

public bool AllowHandwritingRecognition { get; set; } = false;

public bool IsOnline { get; set; } = false;

public bool RequiresSupervision { get; set; } = true;

[StringLength(500)]

public string Instructions { get; set; } = string.Empty;

public string MaterialsRequired { get; set; } = string.Empty; // JSON list

// Navigation Properties

public virtual ICollection<ExamQuestion> Questions { get; set; } = new List<ExamQuestion>();

public virtual ICollection<ExamRegistration> Registrations { get; set; } = new List<ExamRegistration>();

public virtual ICollection<ExamSession> Sessions { get; set; } = new List<ExamSession>();

public virtual ICollection<ExamResult> Results { get; set; } = new List<ExamResult>();

public virtual ICollection<ExamSupervisor> Supervisors { get; set; } = new List<ExamSupervisor>();

}

public enum ExamType

{

Quiz,

Test,

MidTerm,

FinalExam,

Assignment,

Project,

Practical,

Oral,

Continuous

}

public enum ExamStatus

{

Draft,

Published,

InProgress,

Completed,

Cancelled,

Postponed

}

}

// Models/Entities/Exam/ExamQuestion.cs

namespace SchoolManagement.Core.Entities

{

public class ExamQuestion : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public int QuestionNumber { get; set; }

[Required]

public string QuestionText { get; set; } = string.Empty;

public QuestionType Type { get; set; }

public decimal Marks { get; set; }

public bool IsRequired { get; set; } = true;

public TimeSpan? TimeLimit { get; set; }

// For multiple choice questions

public string Options { get; set; } = string.Empty; // JSON array

public string CorrectAnswer { get; set; } = string.Empty;

// For essay/written questions

public int? WordLimit { get; set; }

public string SampleAnswer { get; set; } = string.Empty;

public string MarkingRubric { get; set; } = string.Empty; // JSON

// For handwriting recognition

public bool EnableHandwritingRecognition { get; set; } = false;

public string ExpectedKeywords { get; set; } = string.Empty; // JSON array

// Media attachments

public string AttachmentPath { get; set; } = string.Empty;

public int SortOrder { get; set; }

// Navigation Properties

public virtual ICollection<ExamAnswer> Answers { get; set; } = new List<ExamAnswer>();

}

public enum QuestionType

{

MultipleChoice,

TrueFalse,

ShortAnswer,

Essay,

FillInTheBlank,

Matching,

Numerical,

Drawing,

FileUpload

}

}

// Models/Entities/Exam/ExamRegistration.cs

namespace SchoolManagement.Core.Entities

{

public class ExamRegistration : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public DateTime RegistrationDate { get; set; } = DateTime.UtcNow;

public RegistrationStatus Status { get; set; } = RegistrationStatus.Registered;

public bool IsPresent { get; set; } = false;

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

[StringLength(100)]

public string SeatNumber { get; set; } = string.Empty;

public bool RequiresSpecialAccommodation { get; set; } = false;

public string SpecialAccommodations { get; set; } = string.Empty;

[StringLength(500)]

public string Remarks { get; set; } = string.Empty;

}

public enum RegistrationStatus

{

Registered,

Present,

Absent,

Late,

Disqualified,

Withdrawn

}

}

// Models/Entities/Exam/ExamSession.cs

namespace SchoolManagement.Core.Entities

{

public class ExamSession : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public DateTime StartTime { get; set; }

public DateTime? EndTime { get; set; }

public TimeSpan ActualDuration { get; set; }

public SessionStatus Status { get; set; } = SessionStatus.NotStarted;

public bool IsSubmitted { get; set; } = false;

public DateTime? SubmittedAt { get; set; }

public bool IsAutoSubmitted { get; set; } = false;

public int TotalQuestions { get; set; }

public int AnsweredQuestions { get; set; }

// Proctoring data

public string DeviceInfo { get; set; } = string.Empty; // JSON

public string IPAddress { get; set; } = string.Empty;

public string BrowserInfo { get; set; } = string.Empty;

// Security flags

public bool HasViolations { get; set; } = false;

public string ViolationDetails { get; set; } = string.Empty; // JSON

public decimal? Score { get; set; }

public decimal? Percentage { get; set; }

// Navigation Properties

public virtual ICollection<ExamAnswer> Answers { get; set; } = new List<ExamAnswer>();

public virtual ICollection<ExamSessionLog> SessionLogs { get; set; } = new List<ExamSessionLog>();

}

public enum SessionStatus

{

NotStarted,

InProgress,

Paused,

Completed,

TimedOut,

Interrupted,

Disqualified

}

}

// Models/Entities/Exam/ExamAnswer.cs

namespace SchoolManagement.Core.Entities

{

public class ExamAnswer : BaseEntity

{

public Guid ExamSessionId { get; set; }

public virtual ExamSession ExamSession { get; set; } = null!;

public Guid ExamQuestionId { get; set; }

public virtual ExamQuestion ExamQuestion { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public string AnswerText { get; set; } = string.Empty;

// For file uploads or handwritten answers

public string AttachmentPath { get; set; } = string.Empty;

public string HandwritingImagePath { get; set; } = string.Empty;

// AI Recognition results

public string RecognizedText { get; set; } = string.Empty;

public decimal AiConfidenceScore { get; set; }

public bool IsAiProcessed { get; set; } = false;

// Marking

public decimal? MarksObtained { get; set; }

public bool IsMarked { get; set; } = false;

public Guid? MarkedByTeacherId { get; set; }

public virtual Teacher? MarkedByTeacher { get; set; }

public DateTime? MarkedAt { get; set; }

public string TeacherFeedback { get; set; } = string.Empty;

// Answer metadata

public DateTime AnsweredAt { get; set; } = DateTime.UtcNow;

public TimeSpan TimeSpent { get; set; }

public int AttemptNumber { get; set; } = 1;

public bool IsFlagged { get; set; } = false;

public string FlagReason { get; set; } = string.Empty;

}

}

// Models/Entities/Exam/ExamResult.cs

namespace SchoolManagement.Core.Entities

{

public class ExamResult : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid StudentId { get; set; }

public virtual Student Student { get; set; } = null!;

public Guid ExamSessionId { get; set; }

public virtual ExamSession ExamSession { get; set; } = null!;

public decimal TotalMarks { get; set; }

public decimal MarksObtained { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public int GradeUnit { get; set; }

public ResultStatus Status { get; set; } = ResultStatus.Pending;

public bool IsPassed { get; set; }

public string Remarks { get; set; } = string.Empty;

// Marking details

public bool IsFullyMarked { get; set; } = false;

public int QuestionsMarked { get; set; }

public int TotalQuestions { get; set; }

public Guid? MarkedByTeacherId { get; set; }

public virtual Teacher? MarkedByTeacher { get; set; }

public DateTime? MarkedAt { get; set; }

// Result publication

public bool IsPublished { get; set; } = false;

public DateTime? PublishedAt { get; set; }

// Analytics

public int Rank { get; set; }

public decimal? ClassAverage { get; set; }

public decimal? HighestScore { get; set; }

public decimal? LowestScore { get; set; }

public string DetailedAnalysis { get; set; } = string.Empty; // JSON

}

public enum ResultStatus

{

Pending,

InProgress,

Completed,

Published,

UnderReview,

Disputed

}

}

// Models/Entities/Exam/ExamSupervisor.cs

namespace SchoolManagement.Core.Entities

{

public class ExamSupervisor : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid TeacherId { get; set; }

public virtual Teacher Teacher { get; set; } = null!;

public SupervisorRole Role { get; set; }

public DateTime AssignedAt { get; set; } = DateTime.UtcNow;

public bool IsPresent { get; set; } = false;

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

public string Responsibilities { get; set; } = string.Empty;

public string Notes { get; set; } = string.Empty;

// Navigation Properties

public virtual ICollection<ExamIncident> ReportedIncidents { get; set; } = new List<ExamIncident>();

}

public enum SupervisorRole

{

ChiefSupervisor,

AssistantSupervisor,

RoomSupervisor,

FloatingSupervisor,

TechnicalSupport

}

}

// Models/Entities/Exam/ExamIncident.cs

namespace SchoolManagement.Core.Entities

{

public class ExamIncident : BaseEntity

{

public Guid ExamId { get; set; }

public virtual Exam Exam { get; set; } = null!;

public Guid? StudentId { get; set; }

public virtual Student? Student { get; set; }

public Guid ReportedByTeacherId { get; set; }

public virtual Teacher ReportedByTeacher { get; set; } = null!;

public IncidentType Type { get; set; }

public IncidentSeverity Severity { get; set; }

[Required]

public string Description { get; set; } = string.Empty;

public DateTime IncidentTime { get; set; }

public string Location { get; set; } = string.Empty;

public string ActionTaken { get; set; } = string.Empty;

public bool RequiresFollowUp { get; set; } = false;

public string FollowUpNotes { get; set; } = string.Empty;

public string AttachmentsPath { get; set; } = string.Empty; // Evidence photos/videos

public IncidentStatus Status { get; set; } = IncidentStatus.Open;

public DateTime? ResolvedAt { get; set; }

public Guid? ResolvedByTeacherId { get; set; }

public virtual Teacher? ResolvedByTeacher { get; set; }

}

public enum IncidentType

{

Cheating,

MobilePhone,

UnauthorizedMaterial,

Disruption,

LateComing,

EarlyLeaving,

TechnicalIssue,

MedicalEmergency,

Misconduct,

Other

}

public enum IncidentSeverity

{

Low,

Medium,

High,

Critical

}

public enum IncidentStatus

{

Open,

UnderInvestigation,

Resolved,

Closed,

Escalated

}

}

// Models/Entities/Exam/ExamSessionLog.cs

namespace SchoolManagement.Core.Entities

{

public class ExamSessionLog : BaseEntity

{

public Guid ExamSessionId { get; set; }

public virtual ExamSession ExamSession { get; set; } = null!;

public LogType Type { get; set; }

public string Event { get; set; } = string.Empty;

public string Details { get; set; } = string.Empty; // JSON

public DateTime Timestamp { get; set; } = DateTime.UtcNow;

public string IPAddress { get; set; } = string.Empty;

public string UserAgent { get; set; } = string.Empty;

public bool IsSuspicious { get; set; } = false;

public string SuspicionReason { get; set; } = string.Empty;

}

public enum LogType

{

SessionStart,

SessionEnd,

QuestionViewed,

AnswerSaved,

AnswerChanged,

PageRefresh,

WindowFocusLost,

WindowFocusGained,

CopyDetected,

PasteDetected,

RightClickDisabled,

DevToolsDetected,

NetworkChange,

SuspiciousActivity,

SystemError

}

}

// DTOs/ExamDtos.cs

namespace SchoolManagement.Core.DTOs

{

public class ExamDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public ExamType Type { get; set; }

public DateTime ExamDate { get; set; }

public TimeSpan Duration { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public decimal TotalMarks { get; set; }

public decimal PassingMarks { get; set; }

public string Venue { get; set; } = string.Empty;

public ExamStatus Status { get; set; }

public bool IsPublished { get; set; }

public bool AllowHandwritingRecognition { get; set; }

public bool IsOnline { get; set; }

public string Instructions { get; set; } = string.Empty;

public SubjectDto Subject { get; set; } = null!;

public GradeDto Grade { get; set; } = null!;

public TermDto Term { get; set; } = null!;

public TeacherDto CreatedByTeacher { get; set; } = null!;

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

public List<ExamSupervisorDto> Supervisors { get; set; } = new();

public int RegisteredStudents { get; set; }

public int CompletedStudents { get; set; }

public DateTime CreatedAt { get; set; }

}

public class CreateExamDto

{

[Required]

public Guid SubjectId { get; set; }

[Required]

public Guid GradeId { get; set; }

[Required]

public Guid TermId { get; set; }

[Required]

public Guid SchoolYearId { get; set; }

[Required]

[StringLength(200)]

public string Title { get; set; } = string.Empty;

[StringLength(1000)]

public string Description { get; set; } = string.Empty;

[Required]

public ExamType Type { get; set; }

[Required]

public DateTime ExamDate { get; set; }

[Required]

public TimeSpan Duration { get; set; }

[Required]

public TimeSpan StartTime { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal TotalMarks { get; set; }

[Required]

[Range(0, double.MaxValue)]

public decimal PassingMarks { get; set; }

[StringLength(100)]

public string Venue { get; set; } = string.Empty;

public bool AllowHandwritingRecognition { get; set; } = false;

public bool IsOnline { get; set; } = false;

public bool RequiresSupervision { get; set; } = true;

[StringLength(500)]

public string Instructions { get; set; } = string.Empty;

public List<string> MaterialsRequired { get; set; } = new();

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

public List<Guid> SupervisorIds { get; set; } = new();

}

public class ExamQuestionDto

{

public Guid Id { get; set; }

public int QuestionNumber { get; set; }

public string QuestionText { get; set; } = string.Empty;

public QuestionType Type { get; set; }

public decimal Marks { get; set; }

public bool IsRequired { get; set; }

public TimeSpan? TimeLimit { get; set; }

public List<string> Options { get; set; } = new();

public string CorrectAnswer { get; set; } = string.Empty;

public int? WordLimit { get; set; }

public bool EnableHandwritingRecognition { get; set; }

public List<string> ExpectedKeywords { get; set; } = new();

public string AttachmentPath { get; set; } = string.Empty;

public int SortOrder { get; set; }

}

public class CreateExamQuestionDto

{

[Required]

public int QuestionNumber { get; set; }

[Required]

public string QuestionText { get; set; } = string.Empty;

[Required]

public QuestionType Type { get; set; }

[Required]

[Range(0.01, double.MaxValue)]

public decimal Marks { get; set; }

public bool IsRequired { get; set; } = true;

public TimeSpan? TimeLimit { get; set; }

// For multiple choice

public List<string> Options { get; set; } = new();

public string CorrectAnswer { get; set; } = string.Empty;

// For written answers

public int? WordLimit { get; set; }

public string SampleAnswer { get; set; } = string.Empty;

public bool EnableHandwritingRecognition { get; set; } = false;

public List<string> ExpectedKeywords { get; set; } = new();

public int SortOrder { get; set; }

}

public class ExamSessionDto

{

public Guid Id { get; set; }

public ExamDto Exam { get; set; } = null!;

public StudentDto Student { get; set; } = null!;

public DateTime StartTime { get; set; }

public DateTime? EndTime { get; set; }

public TimeSpan ActualDuration { get; set; }

public SessionStatus Status { get; set; }

public bool IsSubmitted { get; set; }

public int TotalQuestions { get; set; }

public int AnsweredQuestions { get; set; }

public decimal? Score { get; set; }

public decimal? Percentage { get; set; }

public bool HasViolations { get; set; }

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

}

public class ExamAnswerDto

{

public Guid Id { get; set; }

public Guid ExamQuestionId { get; set; }

public string AnswerText { get; set; } = string.Empty;

public string AttachmentPath { get; set; } = string.Empty;

public string HandwritingImagePath { get; set; } = string.Empty;

public string RecognizedText { get; set; } = string.Empty;

public decimal AiConfidenceScore { get; set; }

public decimal? MarksObtained { get; set; }

public bool IsMarked { get; set; }

public string TeacherFeedback { get; set; } = string.Empty;

public DateTime AnsweredAt { get; set; }

public TimeSpan TimeSpent { get; set; }

public bool IsFlagged { get; set; }

}

public class ExamResultDto

{

public Guid Id { get; set; }

public ExamDto Exam { get; set; } = null!;

public StudentDto Student { get; set; } = null!;

public decimal TotalMarks { get; set; }

public decimal MarksObtained { get; set; }

public decimal Percentage { get; set; }

public string Grade { get; set; } = string.Empty;

public bool IsPassed { get; set; }

public string Remarks { get; set; } = string.Empty;

public bool IsPublished { get; set; }

public int Rank { get; set; }

public decimal? ClassAverage { get; set; }

public Dictionary<string, object> DetailedAnalysis { get; set; } = new();

}

public class StartExamSessionDto

{

[Required]

public Guid ExamId { get; set; }

[Required]

public Guid StudentId { get; set; }

public string DeviceInfo { get; set; } = string.Empty;

public string BrowserInfo { get; set; } = string.Empty;

}

public class SubmitExamAnswerDto

{

[Required]

public Guid ExamSessionId { get; set; }

[Required]

public Guid ExamQuestionId { get; set; }

public string AnswerText { get; set; } = string.Empty;

public byte[]? AttachmentFile { get; set; }

public byte[]? HandwritingImage { get; set; }

}

public class ExamStatisticsDto

{

public Guid ExamId { get; set; }

public int TotalRegistered { get; set; }

public int TotalCompleted { get; set; }

public int TotalPassed { get; set; }

public int TotalFailed { get; set; }

public decimal AverageScore { get; set; }

public decimal HighestScore { get; set; }

public decimal LowestScore { get; set; }

public decimal PassPercentage { get; set; }

public Dictionary<string, int> GradeDistribution { get; set; } = new();

public List<QuestionAnalysisDto> QuestionAnalysis { get; set; } = new();

}

public class QuestionAnalysisDto

{

public Guid QuestionId { get; set; }

public int QuestionNumber { get; set; }

public decimal AverageScore { get; set; }

public int TotalAttempts { get; set; }

public int CorrectAnswers { get; set; }

public decimal DifficultyIndex { get; set; }

public List<string> CommonMistakes { get; set; } = new();

}

}// Services/Interfaces/IExamService.cs

using SchoolManagement.Core.DTOs;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Services.Interfaces

{

public interface IExamService

{

// Exam Management

Task<ExamDto> CreateExamAsync(CreateExamDto createExamDto, Guid teacherId);

Task<ExamDto> UpdateExamAsync(Guid examId, UpdateExamDto updateExamDto);

Task<bool> DeleteExamAsync(Guid examId);

Task<ExamDto> GetExamByIdAsync(Guid examId);

Task<IEnumerable<ExamDto>> GetExamsBySchoolAsync(Guid schoolId);

Task<IEnumerable<ExamDto>> GetExamsBySubjectAsync(Guid subjectId, Guid termId);

Task<IEnumerable<ExamDto>> GetExamsByGradeAsync(Guid gradeId, Guid termId);

Task<IEnumerable<ExamDto>> GetUpcomingExamsAsync(Guid schoolId, DateTime fromDate, DateTime toDate);

// Exam Publication

Task<bool> PublishExamAsync(Guid examId);

Task<bool> UnpublishExamAsync(Guid examId);

// Student Registration

Task<bool> RegisterStudentForExamAsync(Guid examId, Guid studentId);

Task<bool> RegisterStudentsForExamAsync(Guid examId, List<Guid> studentIds);

Task<bool> UnregisterStudentFromExamAsync(Guid examId, Guid studentId);

Task<IEnumerable<ExamRegistrationDto>> GetExamRegistrationsAsync(Guid examId);

// Exam Sessions

Task<ExamSessionDto> StartExamSessionAsync(StartExamSessionDto startSessionDto);

Task<ExamSessionDto> GetExamSessionAsync(Guid sessionId);

Task<bool> EndExamSessionAsync(Guid sessionId);

Task<bool> PauseExamSessionAsync(Guid sessionId);

Task<bool> ResumeExamSessionAsync(Guid sessionId);

// Answer Submission

Task<bool> SubmitAnswerAsync(SubmitExamAnswerDto submitAnswerDto);

Task<bool> SaveAnswerDraftAsync(SubmitExamAnswerDto saveAnswerDto);

Task<bool> SubmitExamAsync(Guid sessionId);

// Marking & Results

Task<bool> MarkAnswerAsync(Guid answerId, decimal marks, string feedback);

Task<bool> AutoMarkExamAsync(Guid examId);

Task<ExamResultDto> GenerateExamResultAsync(Guid sessionId);

Task<bool> PublishResultsAsync(Guid examId);

Task<IEnumerable<ExamResultDto>> GetExamResultsAsync(Guid examId);

Task<ExamResultDto> GetStudentExamResultAsync(Guid examId, Guid studentId);

// Statistics & Analytics

Task<ExamStatisticsDto> GetExamStatisticsAsync(Guid examId);

Task<IEnumerable<QuestionAnalysisDto>> GetQu

Task<IEnumerable<QuestionAnalysisDto>> GetQuestionAnalysisAsync(Guid examId);

Task<byte[]> GenerateExamReportAsync(Guid examId);

Task<byte[]> GenerateStudentExamResultAsync(Guid examResultId);

// Supervision & Security

Task<bool> AssignSupervisorAsync(Guid examId, Guid teacherId, SupervisorRole role);

Task<bool> RemoveSupervisorAsync(Guid examId, Guid teacherId);

Task<bool> CheckInSupervisorAsync(Guid examId, Guid teacherId);

Task<bool> CheckOutSupervisorAsync(Guid examId, Guid teacherId);

Task<bool> ReportIncidentAsync(CreateExamIncidentDto incidentDto);

Task<IEnumerable<ExamIncidentDto>> GetExamIncidentsAsync(Guid examId);

// Student Experience

Task<IEnumerable<ExamDto>> GetStudentUpcomingExamsAsync(Guid studentId);

Task<IEnumerable<ExamResultDto>> GetStudentExamHistoryAsync(Guid studentId);

Task<ExamDto> GetStudentExamDetailsAsync(Guid examId, Guid studentId);

Task<bool> ValidateExamAccessAsync(Guid examId, Guid studentId);

// Handwriting Integration

Task<bool> ProcessHandwrittenAnswersAsync(Guid examId);

Task<bool> ReviewHandwritingRecognitionAsync(Guid answerId, string correctedText);

}

}

// Services/Implementations/ExamService.cs

using Microsoft.EntityFrameworkCore;

using SchoolManagement.Data;

using SchoolManagement.Core.Entities;

using SchoolManagement.Core.DTOs;

using SchoolManagement.Services.Interfaces;

namespace SchoolManagement.Services.Implementations

{

public class ExamService : IExamService

{

private readonly SchoolManagementDbContext \_context;

private readonly IMapper \_mapper;

private readonly ILogger<ExamService> \_logger;

private readonly IHandwritingRecognitionService \_handwritingService;

private readonly IPdfGenerationService \_pdfService;

private readonly INotificationService \_notificationService;

private readonly IGradingService \_gradingService;

public ExamService(

SchoolManagementDbContext context,

IMapper mapper,

ILogger<ExamService> logger,

IHandwritingRecognitionService handwritingService,

IPdfGenerationService pdfService,

INotificationService notificationService,

IGradingService gradingService)

{

\_context = context;

\_mapper = mapper;

\_logger = logger;

\_handwritingService = handwritingService;

\_pdfService = pdfService;

\_notificationService = notificationService;

\_gradingService = gradingService;

}

public async Task<ExamDto> CreateExamAsync(CreateExamDto createExamDto, Guid teacherId)

{

// Validate teacher has permission for this subject and grade

var hasPermission = await \_context.SubjectTeachers

.AnyAsync(st => st.TeacherId == teacherId &&

st.SubjectId == createExamDto.SubjectId &&

st.SchoolYearId == createExamDto.SchoolYearId &&

st.IsActive);

if (!hasPermission)

throw new UnauthorizedException("Teacher not authorized to create exam for this subject");

// Validate exam date is not in the past

if (createExamDto.ExamDate.Date < DateTime.Today)

throw new ValidationException("Exam date cannot be in the past");

var exam = \_mapper.Map<Exam>(createExamDto);

exam.CreatedByTeacherId = teacherId;

exam.EndTime = exam.StartTime.Add(exam.Duration);

// Get school ID from subject

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

exam.SchoolId = subject!.SchoolId;

\_context.Exams.Add(exam);

await \_context.SaveChangesAsync();

// Add questions

if (createExamDto.Questions.Any())

{

var questions = createExamDto.Questions.Select(q =>

{

var question = \_mapper.Map<ExamQuestion>(q);

question.ExamId = exam.Id;

question.Options = q.Options.Any() ? System.Text.Json.JsonSerializer.Serialize(q.Options) : "";

question.ExpectedKeywords = q.ExpectedKeywords.Any() ? System.Text.Json.JsonSerializer.Serialize(q.ExpectedKeywords) : "";

return question;

}).ToList();

\_context.ExamQuestions.AddRange(questions);

}

// Assign supervisors

if (createExamDto.SupervisorIds.Any())

{

var supervisors = createExamDto.SupervisorIds.Select(supervisorId => new ExamSupervisor

{

ExamId = exam.Id,

TeacherId = supervisorId,

Role = SupervisorRole.RoomSupervisor

}).ToList();

\_context.ExamSupervisors.AddRange(supervisors);

}

await \_context.SaveChangesAsync();

return await GetExamByIdAsync(exam.Id);

}

public async Task<ExamDto> UpdateExamAsync(Guid examId, UpdateExamDto updateExamDto)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null)

throw new NotFoundException($"Exam {examId} not found");

if (exam.Status == ExamStatus.InProgress || exam.Status == ExamStatus.Completed)

throw new ConflictException("Cannot update exam that is in progress or completed");

\_mapper.Map(updateExamDto, exam);

if (updateExamDto.Duration.HasValue && updateExamDto.StartTime.HasValue)

{

exam.EndTime = exam.StartTime.Add(exam.Duration);

}

await \_context.SaveChangesAsync();

return await GetExamByIdAsync(examId);

}

public async Task<bool> DeleteExamAsync(Guid examId)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null)

return false;

if (exam.Status == ExamStatus.InProgress)

throw new ConflictException("Cannot delete exam that is in progress");

exam.IsDeleted = true;

await \_context.SaveChangesAsync();

return true;

}

public async Task<ExamDto> GetExamByIdAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.Include(e => e.Questions.OrderBy(q => q.SortOrder))

.Include(e => e.Supervisors)

.ThenInclude(s => s.Teacher)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null)

throw new NotFoundException($"Exam {examId} not found");

var examDto = \_mapper.Map<ExamDto>(exam);

// Get registration counts

examDto.RegisteredStudents = await \_context.ExamRegistrations

.CountAsync(r => r.ExamId == examId);

examDto.CompletedStudents = await \_context.ExamSessions

.CountAsync(s => s.ExamId == examId && s.IsSubmitted);

return examDto;

}

public async Task<IEnumerable<ExamDto>> GetExamsBySchoolAsync(Guid schoolId)

{

var exams = await \_context.Exams

.Where(e => e.SchoolId == schoolId)

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderByDescending(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<IEnumerable<ExamDto>> GetExamsBySubjectAsync(Guid subjectId, Guid termId)

{

var exams = await \_context.Exams

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

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<IEnumerable<ExamDto>> GetExamsByGradeAsync(Guid gradeId, Guid termId)

{

var exams = await \_context.Exams

.Where(e => e.GradeId == gradeId && e.TermId == termId)

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<IEnumerable<ExamDto>> GetUpcomingExamsAsync(Guid schoolId, DateTime fromDate, DateTime toDate)

{

var exams = await \_context.Exams

.Where(e => e.SchoolId == schoolId &&

e.ExamDate >= fromDate &&

e.ExamDate <= toDate &&

e.IsPublished)

.Include(e => e.Subject)

.Include(e => e.Grade)

.Include(e => e.Term)

.Include(e => e.CreatedByTeacher)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(exams);

}

public async Task<bool> PublishExamAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Questions)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null)

return false;

// Validate exam is ready for publication

if (!exam.Questions.Any())

throw new ValidationException("Cannot publish exam without questions");

if (exam.ExamDate <= DateTime.Now)

throw new ValidationException("Cannot publish exam with past date");

exam.IsPublished = true;

exam.PublishedAt = DateTime.UtcNow;

exam.Status = ExamStatus.Published;

await \_context.SaveChangesAsync();

// Auto-register eligible students

await AutoRegisterStudentsAsync(examId);

// Send notifications

await NotifyStudentsAboutExamAsync(examId);

return true;

}

public async Task<bool> UnpublishExamAsync(Guid examId)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null)

return false;

if (exam.Status == ExamStatus.InProgress || exam.Status == ExamStatus.Completed)

throw new ConflictException("Cannot unpublish exam that is in progress or completed");

exam.IsPublished = false;

exam.Status = ExamStatus.Draft;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> RegisterStudentForExamAsync(Guid examId, Guid studentId)

{

// Check if already registered

var existingRegistration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

if (existingRegistration != null)

return true; // Already registered

// Validate student is in the correct grade

var exam = await \_context.Exams

.Include(e => e.Grade)

.FirstOrDefaultAsync(e => e.Id == examId);

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

if (exam == null || student == null)

return false;

if (student.CurrentGradeId != exam.GradeId)

throw new ValidationException("Student is not in the correct grade for this exam");

var registration = new ExamRegistration

{

ExamId = examId,

StudentId = studentId,

RegistrationDate = DateTime.UtcNow,

Status = RegistrationStatus.Registered

};

\_context.ExamRegistrations.Add(registration);

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> RegisterStudentsForExamAsync(Guid examId, List<Guid> studentIds)

{

var tasks = studentIds.Select(studentId => RegisterStudentForExamAsync(examId, studentId));

var results = await Task.WhenAll(tasks);

return results.All(r => r);

}

public async Task<ExamSessionDto> StartExamSessionAsync(StartExamSessionDto startSessionDto)

{

// Validate exam is available

var exam = await \_context.Exams

.Include(e => e.Questions)

.FirstOrDefaultAsync(e => e.Id == startSessionDto.ExamId);

if (exam == null)

throw new NotFoundException("Exam not found");

if (!exam.IsPublished)

throw new ValidationException("Exam is not published");

if (exam.Status != ExamStatus.Published)

throw new ValidationException("Exam is not available for taking");

// Check exam timing

var currentTime = DateTime.Now.TimeOfDay;

var examDate = exam.ExamDate.Date;

if (DateTime.Today != examDate)

throw new ValidationException("Exam is not scheduled for today");

if (currentTime < exam.StartTime)

throw new ValidationException("Exam has not started yet");

if (currentTime > exam.EndTime)

throw new ValidationException("Exam time has ended");

// Check if student is registered

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == startSessionDto.ExamId &&

r.StudentId == startSessionDto.StudentId &&

r.Status == RegistrationStatus.Registered);

if (registration == null)

throw new ValidationException("Student is not registered for this exam");

// Check if session already exists

var existingSession = await \_context.ExamSessions

.FirstOrDefaultAsync(s => s.ExamId == startSessionDto.ExamId &&

s.StudentId == startSessionDto.StudentId);

if (existingSession != null)

{

if (existingSession.IsSubmitted)

throw new ValidationException("Exam has already been submitted");

// Resume existing session

existingSession.Status = SessionStatus.InProgress;

await \_context.SaveChangesAsync();

return \_mapper.Map<ExamSessionDto>(existingSession);

}

// Create new session

var session = new ExamSession

{

ExamId = startSessionDto.ExamId,

StudentId = startSessionDto.StudentId,

StartTime = DateTime.UtcNow,

Status = SessionStatus.InProgress,

TotalQuestions = exam.Questions.Count,

DeviceInfo = startSessionDto.DeviceInfo,

BrowserInfo = startSessionDto.BrowserInfo,

IPAddress = GetClientIPAddress()

};

\_context.ExamSessions.Add(session);

// Mark student as present

registration.IsPresent = true;

registration.CheckInTime = DateTime.UtcNow;

registration.Status = RegistrationStatus.Present;

// Log session start

var log = new ExamSessionLog

{

ExamSessionId = session.Id,

Type = LogType.SessionStart,

Event = "Session Started",

Details = System.Text.Json.JsonSerializer.Serialize(new { Device = startSessionDto.DeviceInfo, Browser = startSessionDto.BrowserInfo }),

IPAddress = session.IPAddress

};

\_context.ExamSessionLogs.Add(log);

await \_context.SaveChangesAsync();

return \_mapper.Map<ExamSessionDto>(session);

}

public async Task<bool> SubmitAnswerAsync(SubmitExamAnswerDto submitAnswerDto)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.FirstOrDefaultAsync(s => s.Id == submitAnswerDto.ExamSessionId);

if (session == null || session.IsSubmitted)

return false;

var question = await \_context.ExamQuestions

.FirstOrDefaultAsync(q => q.Id == submitAnswerDto.ExamQuestionId);

if (question == null)

return false;

// Check if answer already exists

var existingAnswer = await \_context.ExamAnswers

.FirstOrDefaultAsync(a => a.ExamSessionId == submitAnswerDto.ExamSessionId &&

a.ExamQuestionId == submitAnswerDto.ExamQuestionId);

ExamAnswer answer;

if (existingAnswer != null)

{

// Update existing answer

answer = existingAnswer;

answer.AnswerText = submitAnswerDto.AnswerText;

answer.AnsweredAt = DateTime.UtcNow;

answer.AttemptNumber++;

}

else

{

// Create new answer

answer = new ExamAnswer

{

ExamSessionId = submitAnswerDto.ExamSessionId,

ExamQuestionId = submitAnswerDto.ExamQuestionId,

StudentId = session.StudentId,

AnswerText = submitAnswerDto.AnswerText,

AnsweredAt = DateTime.UtcNow,

AttemptNumber = 1

};

\_context.ExamAnswers.Add(answer);

}

// Handle file attachments

if (submitAnswerDto.AttachmentFile != null)

{

var attachmentPath = await SaveAttachmentAsync(

submitAnswerDto.AttachmentFile,

session.StudentId,

submitAnswerDto.ExamSessionId);

answer.AttachmentPath = attachmentPath;

}

// Handle handwriting image

if (submitAnswerDto.HandwritingImage != null && question.EnableHandwritingRecognition)

{

var imagePath = await SaveHandwritingImageAsync(

submitAnswerDto.HandwritingImage,

session.StudentId,

submitAnswerDto.ExamSessionId);

answer.HandwritingImagePath = imagePath;

// Process handwriting asynchronously

\_ = Task.Run(() => ProcessHandwritingAnswerAsync(answer.Id, imagePath, question.ExpectedKeywords));

}

// Auto-mark if possible (multiple choice, true/false)

if (question.Type == QuestionType.MultipleChoice || question.Type == QuestionType.TrueFalse)

{

if (answer.AnswerText.Trim().Equals(question.CorrectAnswer.Trim(), StringComparison.OrdinalIgnoreCase))

{

answer.MarksObtained = question.Marks;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

else

{

answer.MarksObtained = 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

}

// Update session progress

var answeredQuestions = await \_context.ExamAnswers

.CountAsync(a => a.ExamSessionId == submitAnswerDto.ExamSessionId && !string.IsNullOrEmpty(a.AnswerText));

session.AnsweredQuestions = answeredQuestions;

// Log answer submission

var log = new ExamSessionLog

{

ExamSessionId = submitAnswerDto.ExamSessionId,

Type = LogType.AnswerSaved,

Event = "Answer Submitted",

Details = System.Text.Json.JsonSerializer.Serialize(new { QuestionId = submitAnswerDto.ExamQuestionId, AnswerLength = submitAnswerDto.AnswerText.Length }),

IPAddress = GetClientIPAddress()

};

\_context.ExamSessionLogs.Add(log);

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> SubmitExamAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.Include(s => s.Answers)

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

if (session == null || session.IsSubmitted)

return false;

session.IsSubmitted = true;

session.EndTime = DateTime.UtcNow;

session.ActualDuration = session.EndTime.Value - session.StartTime;

session.Status = SessionStatus.Completed;

// Update registration status

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == session.ExamId && r.StudentId == session.StudentId);

if (registration != null)

{

registration.CheckOutTime = DateTime.UtcNow;

}

// Log submission

var log = new ExamSessionLog

{

ExamSessionId = sessionId,

Type = LogType.SessionEnd,

Event = "Exam Submitted",

Details = System.Text.Json.JsonSerializer.Serialize(new { TotalAnswers = session.Answers.Count, Duration = session.ActualDuration }),

IPAddress = GetClientIPAddress()

};

\_context.ExamSessionLogs.Add(log);

await \_context.SaveChangesAsync();

// Start auto-marking process

\_ = Task.Run(() => AutoMarkSessionAsync(sessionId));

return true;

}

public async Task<bool> MarkAnswerAsync(Guid answerId, decimal marks, string feedback)

{

var answer = await \_context.ExamAnswers

.Include(a => a.ExamQuestion)

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

if (answer == null)

return false;

if (marks > answer.ExamQuestion.Marks)

throw new ValidationException($"Marks cannot exceed question maximum of {answer.ExamQuestion.Marks}");

answer.MarksObtained = marks;

answer.TeacherFeedback = feedback;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

// Update session and result if all questions are marked

await UpdateSessionResultAsync(answer.ExamSessionId);

return true;

}

public async Task<ExamResultDto> GenerateExamResultAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.ThenInclude(e => e.Subject)

.Include(s => s.Exam)

.ThenInclude(e => e.Grade)

.Include(s => s.Student)

.Include(s => s.Answers)

.ThenInclude(a => a.ExamQuestion)

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

if (session == null)

throw new NotFoundException("Exam session not found");

var marksObtained = session.Answers.Where(a => a.IsMarked).Sum(a => a.MarksObtained ?? 0);

var totalMarks = session.Exam.TotalMarks;

var percentage = totalMarks > 0 ? (marksObtained / totalMarks) \* 100 : 0;

// Calculate grade using school's grading scheme

var student = await \_context.Students

.Include(s => s.School)

.ThenInclude(sch => sch.GradingScheme)

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

string grade = "F";

if (student?.School?.GradingScheme != null)

{

var gradeCalculation = await \_gradingService.CalculateGradeAsync(

student.School.GradingScheme.Id,

percentage);

grade = gradeCalculation.Symbol;

}

// Check if result already exists

var existingResult = await \_context.ExamResults

.FirstOrDefaultAsync(r => r.ExamSessionId == sessionId);

ExamResult result;

if (existingResult != null)

{

result = existingResult;

}

else

{

result = new ExamResult

{

ExamId = session.ExamId,

StudentId = session.StudentId,

ExamSessionId = sessionId

};

\_context.ExamResults.Add(result);

}

result.TotalMarks = totalMarks;

result.MarksObtained = marksObtained;

result.Percentage = percentage;

result.Grade = grade;

result.IsPassed = percentage >= ((session.Exam.PassingMarks / session.Exam.TotalMarks) \* 100);

result.QuestionsMarked = session.Answers.Count(a => a.IsMarked);

result.TotalQuestions = session.Answers.Count;

result.IsFullyMarked = result.QuestionsMarked == result.TotalQuestions;

if (result.IsFullyMarked)

{

result.Status = ResultStatus.Completed;

}

await \_context.SaveChangesAsync();

return \_mapper.Map<ExamResultDto>(result);

}

public async Task<ExamStatisticsDto> GetExamStatisticsAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Questions)

.Include(e => e.Results)

.ThenInclude(r => r.Student)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null)

throw new NotFoundException("Exam not found");

var totalRegistered = await \_context.ExamRegistrations.CountAsync(r => r.ExamId == examId);

var totalCompleted = exam.Results.Count(r => r.IsFullyMarked);

var totalPassed = exam.Results.Count(r => r.IsPassed);

var totalFailed = totalCompleted - totalPassed;

var completedResults = exam.Results.Where(r => r.IsFullyMarked).ToList();

var averageScore = completedResults.Any() ? completedResults.Average(r => r.Percentage) : 0;

var highestScore = completedResults.Any() ? completedResults.Max(r => r.Percentage) : 0;

var lowestScore = completedResults.Any() ? completedResults.Min(r => r.Percentage) : 0;

var gradeDistribution = completedResults

.GroupBy(r => r.Grade)

.ToDictionary(g => g.Key, g => g.Count());

// Question analysis

var questionAnalysis = new List<QuestionAnalysisDto>();

foreach (var question in exam.Questions)

{

var questionAnswers = await \_context.ExamAnswers

.Where(a => a.ExamQuestionId == question.Id && a.IsMarked)

.ToListAsync();

if (questionAnswers.Any())

{

var analysis = new QuestionAnalysisDto

{

QuestionId = question.Id,

QuestionNumber = question.QuestionNumber,

AverageScore = questionAnswers.Average(a => a.MarksObtained ?? 0),

TotalAttempts = questionAnswers.Count,

CorrectAnswers = questionAnswers.Count(a => (a.MarksObtained ?? 0) == question.Marks),

DifficultyIndex = questionAnswers.Count > 0 ?

questionAnswers.Average(a => (a.MarksObtained ?? 0) / question.Marks) : 0

};

questionAnalysis.Add(analysis);

}

}

return new ExamStatisticsDto

{

ExamId = examId,

TotalRegistered = totalRegistered,

TotalCompleted = totalCompleted,

TotalPassed = totalPassed,

TotalFailed = totalFailed,

AverageScore = averageScore,

HighestScore = highestScore,

LowestScore = lowestScore,

PassPercentage = totalCompleted > 0 ? (decimal)totalPassed / totalCompleted \* 100 : 0,

GradeDistribution = gradeDistribution,

QuestionAnalysis = questionAnalysis

};

}

// Private helper methods

private async Task AutoRegisterStudentsAsync(Guid examId)

{

var exam = await \_context.Exams

.Include(e => e.Grade)

.FirstOrDefaultAsync(e => e.Id == examId);

if (exam == null) return;

// Get all students in the grade who are enrolled in the subject

var eligibleStudents = await \_context.Students

.Where(s => s.CurrentGradeId == exam.GradeId)

.Where(s => s.StudentSubjects.Any(ss => ss.SubjectId == exam.SubjectId &&

ss.SchoolYearId == exam.SchoolYearId &&

ss.IsActive))

.Select(s => s.Id)

.ToListAsync();

await RegisterStudentsForExamAsync(examId, eligibleStudents);

}

private async Task NotifyStudentsAboutExamAsync(Guid examId)

{

var registrations = await \_context.ExamRegistrations

.Where(r => r.ExamId == examId)

.Include(r => r.Student)

.ThenInclude(s => s.Parents)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.ToListAsync();

foreach (var registration in registrations)

{

// Notify student and parents

var exam = registration.Exam;

var student = registration.Student;

var message = $"Exam Alert: {exam.Subject.Name} exam is scheduled for {exam.ExamDate:dd/MM/yyyy} at {exam.StartTime}. Duration: {exam.Duration}. Venue: {exam.Venue}";

foreach (var parent in student.Parents)

{

await \_notificationService.CreateNotificationAsync(new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = student.Id,

Title = $"Exam Notification - {exam.Subject.Name}",

Message = message,

Type = NotificationType.General,

Channel = NotificationChannel.WhatsApp

});

}

}

}

private async Task ProcessHandwritingAnswerAsync(Guid answerId, string imagePath, string expectedKeywordsJson)

{

try

{

var answer = await \_context.ExamAnswers

.Include(a => a.Student)

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

if (answer == null) return;

var processDto = new ProcessHandwritingDto

{

StudentId = answer.StudentId,

ImagePath = imagePath,

ExpectedAnswer = !string.IsNullOrEmpty(expectedKeywordsJson) ?

string.Join(" ", System.Text.Json.JsonSerializer.Deserialize<List<string>>(expectedKeywordsJson) ?? new List<string>()) :

""

};

var result = await \_handwritingService.ProcessHandwritingAsync(processDto);

answer.RecognizedText = result.RecognizedText;

answer.AiConfidenceScore = result.OverallConfidence;

answer.IsAiProcessed = true;

// If confidence is high enough, auto-mark

if (result.OverallConfidence > 0.8m && result.IsCorrect)

{

var question = await \_context.ExamQuestions.FindAsync(answer.ExamQuestionId);

answer.MarksObtained = question?.Marks ?? 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

else if (result.OverallConfidence < 0.6m)

{

// Flag for teacher review

answer.IsFlagged = true;

answer.FlagReason = "Low AI confidence in handwriting recognition";

}

await \_context.SaveChangesAsync();

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error processing handwriting for answer {AnswerId}", answerId);

}

}

private async Task AutoMarkSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Answers)

.ThenInclude(a => a.ExamQuestion)

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

if (session == null) return;

foreach (var answer in session.Answers.Where(a => !a.IsMarked))

{

var question = answer.ExamQuestion;

// Auto-mark objective questions

if (question.Type == QuestionType.MultipleChoice || question.Type == QuestionType.TrueFalse)

{

var isCorrect = answer.AnswerText.Trim().Equals(question.CorrectAnswer.Trim(), StringComparison.OrdinalIgnoreCase);

answer.MarksObtained = isCorrect ? question.Marks : 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

// Auto-mark numerical questions with exact match

else if (question.Type == QuestionType.Numerical)

{

if (decimal.TryParse(answer.AnswerText.Trim(), out var studentAnswer) &&

decimal.TryParse(question.CorrectAnswer.Trim(), out var correctAnswer))

{

var tolerance = 0.01m; // Allow small rounding differences

var isCorrect = Math.Abs(studentAnswer - correctAnswer) <= tolerance;

answer.MarksObtained = isCorrect ? question.Marks : 0;

answer.IsMarked = true;

answer.MarkedAt = DateTime.UtcNow;

}

}

}

await \_context.SaveChangesAsync();

await UpdateSessionResultAsync(sessionId);

}

private async Task UpdateSessionResultAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Answers)

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

if (session == null) return;

var markedAnswers = session.Answers.Where(a => a.IsMarked).ToList();

var totalMarks = markedAnswers.Sum(a => a.MarksObtained ?? 0);

var totalPossible = markedAnswers.Sum(a => a.ExamQuestion?.Marks ?? 0);

session.Score = totalMarks;

session.Percentage = totalPossible > 0 ? (totalMarks / totalPossible) \* 100 : 0;

await \_context.SaveChangesAsync();

// Generate result if all questions are marked

if (session.Answers.All(a => a.IsMarked))

{

await GenerateExamResultAsync(sessionId);

}

}

private async Task<string> SaveAttachmentAsync(byte[] fileData, Guid studentId, Guid sessionId)

{

// Implementation would save to file storage service

var fileName = $"exam\_attachment\_{studentId}\_{sessionId}\_{Guid.NewGuid()}.pdf";

var filePath = $"exam\_attachments/{fileName}";

// Save file logic here

await Task.Delay(10); // Placeholder

return filePath;

}

private async Task<string> SaveHandwritingImageAsync(byte[] imageData, Guid studentId, Guid sessionId)

{

// Implementation would save to file storage service

var fileName = $"handwriting\_{studentId}\_{sessionId}\_{Guid.NewGuid()}.jpg";

var filePath = $"exam\_handwriting/{fileName}";

// Save image logic here

await Task.Delay(10); // Placeholder

return filePath;

}

private string GetClientIPAddress()

{

// Implementation would get actual client IP

return "127.0.0.1"; // Placeholder

}

public async Task<ExamDto> GetStudentExamDetailsAsync(Guid examId, Guid studentId)

{

var exam = await GetExamByIdAsync(examId);

// Check if student is registered

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

if (registration == null)

throw new UnauthorizedException("Student is not registered for this exam");

return exam;

}

public async Task<bool> ValidateExamAccessAsync(Guid examId, Guid studentId)

{

var exam = await \_context.Exams.FindAsync(examId);

if (exam == null || !exam.IsPublished)

return false;

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

return registration != null && registration.Status == RegistrationStatus.Registered;

}

public async Task<IEnumerable<ExamDto>> GetStudentUpcomingExamsAsync(Guid studentId)

{

var upcomingExams = await \_context.ExamRegistrations

.Where(r => r.StudentId == studentId &&

r.Status == RegistrationStatus.Registered &&

r.Exam.ExamDate >= DateTime.Today &&

r.Exam.IsPublished)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.Include(r => r.Exam)

.ThenInclude(e => e.Grade)

.Include(r => r.Exam)

.ThenInclude(e => e.Term)

.Select(r => r.Exam)

.OrderBy(e => e.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamDto>>(upcomingExams);

}

public async Task<IEnumerable<ExamResultDto>> GetStudentExamHistoryAsync(Guid studentId)

{

var results = await \_context.ExamResults

.Where(r => r.StudentId == studentId && r.IsPublished)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.Include(r => r.Exam)

.ThenInclude(e => e.Grade)

.Include(r => r.Student)

.OrderByDescending(r => r.Exam.ExamDate)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamResultDto>>(results);

}

public async Task<bool> ReportIncidentAsync(CreateExamIncidentDto incidentDto)

{

var incident = \_mapper.Map<ExamIncident>(incidentDto);

incident.IncidentTime = DateTime.UtcNow;

\_context.ExamIncidents.Add(incident);

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<ExamIncidentDto>> GetExamIncidentsAsync(Guid examId)

{

var incidents = await \_context.ExamIncidents

.Where(i => i.ExamId == examId)

.Include(i => i.Student)

.Include(i => i.ReportedByTeacher)

.OrderByDescending(i => i.IncidentTime)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamIncidentDto>>(incidents);

}

public async Task<bool> AssignSupervisorAsync(Guid examId, Guid teacherId, SupervisorRole role)

{

var existingSupervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (existingSupervisor != null)

{

existingSupervisor.Role = role;

}

else

{

var supervisor = new ExamSupervisor

{

ExamId = examId,

TeacherId = teacherId,

Role = role,

AssignedAt = DateTime.UtcNow

};

\_context.ExamSupervisors.Add(supervisor);

}

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> PublishResultsAsync(Guid examId)

{

var results = await \_context.ExamResults

.Where(r => r.ExamId == examId && r.IsFullyMarked)

.ToListAsync();

foreach (var result in results)

{

result.IsPublished = true;

result.PublishedAt = DateTime.UtcNow;

result.Status = ResultStatus.Published;

}

await \_context.SaveChangesAsync();

// Notify students and parents

await NotifyResultsPublishedAsync(examId);

return true;

}

private async Task NotifyResultsPublishedAsync(Guid examId)

{

var results = await \_context.ExamResults

.Where(r => r.ExamId == examId && r.IsPublished)

.Include(r => r.Student)

.ThenInclude(s => s.Parents)

.Include(r => r.Exam)

.ThenInclude(e => e.Subject)

.ToListAsync();

foreach (var result in results)

{

var message = $"Exam results published for {result.Exam.Subject.Name}. " +

$"Score: {result.MarksObtained}/{result.TotalMarks} ({result.Percentage:F1}%). " +

$"Grade: {result.Grade}. {(result.IsPassed ? "PASSED" : "FAILED")}";

foreach (var parent in result.Student.Parents)

{

await \_notificationService.CreateNotificationAsync(new CreateNotificationDto

{

ParentId = parent.Id,

StudentId = result.StudentId,

Title = $"Exam Results - {result.Exam.Subject.Name}",

Message = message,

Type = NotificationType.Results,

Channel = NotificationChannel.WhatsApp

});

}

}

}

// Additional interface methods implementation

public async Task<ExamSessionDto> GetExamSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions

.Include(s => s.Exam)

.Include(s => s.Student)

.Include(s => s.Answers)

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

if (session == null)

throw new NotFoundException($"Exam session {sessionId} not found");

return \_mapper.Map<ExamSessionDto>(session);

}

public async Task<bool> EndExamSessionAsync(Guid sessionId)

{

return await SubmitExamAsync(sessionId);

}

public async Task<bool> PauseExamSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions.FindAsync(sessionId);

if (session == null || session.IsSubmitted)

return false;

session.Status = SessionStatus.Paused;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> ResumeExamSessionAsync(Guid sessionId)

{

var session = await \_context.ExamSessions.FindAsync(sessionId);

if (session == null || session.IsSubmitted)

return false;

session.Status = SessionStatus.InProgress;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> SaveAnswerDraftAsync(SubmitExamAnswerDto saveAnswerDto)

{

// Same as SubmitAnswerAsync but mark as draft

return await SubmitAnswerAsync(saveAnswerDto);

}

public async Task<bool> AutoMarkExamAsync(Guid examId)

{

var sessions = await \_context.ExamSessions

.Where(s => s.ExamId == examId && s.IsSubmitted)

.ToListAsync();

foreach (var session in sessions)

{

await AutoMarkSessionAsync(session.Id);

}

return true;

}

public async Task<IEnumerable<ExamResultDto>> GetExamResultsAsync(Guid examId)

{

var results = await \_context.ExamResults

.Where(r => r.ExamId == examId)

.Include(r => r.Student)

.Include(r => r.Exam)

.OrderByDescending(r => r.Percentage)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamResultDto>>(results);

}

public async Task<ExamResultDto> GetStudentExamResultAsync(Guid examId, Guid studentId)

{

var result = await \_context.ExamResults

.Where(r => r.ExamId == examId && r.StudentId == studentId)

.Include(r => r.Student)

.Include(r => r.Exam)

.FirstOrDefaultAsync();

if (result == null)

throw new NotFoundException("Exam result not found");

return \_mapper.Map<ExamResultDto>(result);

}

public async Task<IEnumerable<QuestionAnalysisDto>> GetQuestionAnalysisAsync(Guid examId)

{

var stats = await GetExamStatisticsAsync(examId);

return stats.QuestionAnalysis;

}

public async Task<byte[]> GenerateExamReportAsync(Guid examId)

{

var stats = await GetExamStatisticsAsync(examId);

return await \_pdfService.GenerateExamAnalysisReportAsync(stats);

}

public async Task<byte[]> GenerateStudentExamResultAsync(Guid examResultId)

{

var result = await \_context.ExamResults

.Include(r => r.Student)

.Include(r => r.Exam)

.FirstOrDefaultAsync(r => r.Id == examResultId);

if (result == null)

throw new NotFoundException("Exam result not found");

var resultDto = \_mapper.Map<ExamResultDto>(result);

return await \_pdfService.GenerateStudentExamResultAsync(resultDto);

}

public async Task<bool> RemoveSupervisorAsync(Guid examId, Guid teacherId)

{

var supervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (supervisor == null)

return false;

\_context.ExamSupervisors.Remove(supervisor);

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> CheckInSupervisorAsync(Guid examId, Guid teacherId)

{

var supervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (supervisor == null)

return false;

supervisor.IsPresent = true;

supervisor.CheckInTime = DateTime.UtcNow;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> CheckOutSupervisorAsync(Guid examId, Guid teacherId)

{

var supervisor = await \_context.ExamSupervisors

.FirstOrDefaultAsync(s => s.ExamId == examId && s.TeacherId == teacherId);

if (supervisor == null)

return false;

supervisor.CheckOutTime = DateTime.UtcNow;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> ProcessHandwrittenAnswersAsync(Guid examId)

{

var answers = await \_context.ExamAnswers

.Where(a => a.ExamSession.ExamId == examId &&

!string.IsNullOrEmpty(a.HandwritingImagePath) &&

!a.IsAiProcessed)

.Include(a => a.ExamQuestion)

.ToListAsync();

foreach (var answer in answers)

{

await ProcessHandwritingAnswerAsync(answer.Id, answer.HandwritingImagePath, answer.ExamQuestion.ExpectedKeywords);

}

return true;

}

public async Task<bool> ReviewHandwritingRecognitionAsync(Guid answerId, string correctedText)

{

var answer = await \_context.ExamAnswers.FindAsync(answerId);

if (answer == null)

return false;

answer.RecognizedText = correctedText;

answer.AnswerText = correctedText;

answer.IsFlagged = false;

await \_context.SaveChangesAsync();

return true;

}

public async Task<bool> UnregisterStudentFromExamAsync(Guid examId, Guid studentId)

{

var registration = await \_context.ExamRegistrations

.FirstOrDefaultAsync(r => r.ExamId == examId && r.StudentId == studentId);

if (registration == null)

return false;

registration.Status = RegistrationStatus.Withdrawn;

await \_context.SaveChangesAsync();

return true;

}

public async Task<IEnumerable<ExamRegistrationDto>> GetExamRegistrationsAsync(Guid examId)

{

var registrations = await \_context.ExamRegistrations

.Where(r => r.ExamId == examId)

.Include(r => r.Student)

.Include(r => r.Exam)

.ToListAsync();

return \_mapper.Map<IEnumerable<ExamRegistrationDto>>(registrations);

}

}

}

// Additional DTOs for Exam Management

namespace SchoolManagement.Core.DTOs

{

public class UpdateExamDto

{

[StringLength(200)]

public string? Title { get; set; }

[StringLength(1000)]

public string? Description { get; set; }

public DateTime? ExamDate { get; set; }

public TimeSpan? Duration { get; set; }

public TimeSpan? StartTime { get; set; }

[Range(0.01, double.MaxValue)]

public decimal? TotalMarks { get; set; }

[Range(0, double.MaxValue)]

public decimal? PassingMarks { get; set; }

[StringLength(100)]

public string? Venue { get; set; }

public bool? AllowHandwritingRecognition { get; set; }

public bool? IsOnline { get; set; }

[StringLength(500)]

public string? Instructions { get; set; }

}

public class ExamRegistrationDto

{

public Guid Id { get; set; }

public StudentDto Student { get; set; } = null!;

public ExamDto Exam { get; set; } = null!;

public DateTime RegistrationDate { get; set; }

public RegistrationStatus Status { get; set; }

public bool IsPresent { get; set; }

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

public string SeatNumber { get; set; } = string.Empty;

public bool RequiresSpecialAccommodation { get; set; }

public string SpecialAccommodations { get; set; } = string.Empty;

public string Remarks { get; set; } = string.Empty;

}

public class ExamSupervisorDto

{

public Guid Id { get; set; }

public TeacherDto Teacher { get; set; } = null!;

public SupervisorRole Role { get; set; }

public DateTime AssignedAt { get; set; }

public bool IsPresent { get; set; }

public DateTime? CheckInTime { get; set; }

public DateTime? CheckOutTime { get; set; }

public string Responsibilities { get; set; } = string.Empty;

public string Notes { get; set; } = string.Empty;

}

public class ExamIncidentDto

{

public Guid Id { get; set; }

public StudentDto? Student { get; set; }

public TeacherDto ReportedByTeacher { get; set; } = null!;

public IncidentType Type { get; set; }

public IncidentSeverity Severity { get; set; }

public string Description { get; set; } = string.Empty;

public DateTime IncidentTime { get; set; }

public string Location { get; set; } = string.Empty;

public string ActionTaken { get; set; } = string.Empty;

public bool RequiresFollowUp { get; set; }

public IncidentStatus Status { get; set; }

}

public class CreateExamIncidentDto

{

[Required]

public Guid ExamId { get; set; }

public Guid? StudentId { get; set; }

[Required]

public Guid ReportedByTeacherId { get; set; }

[Required]

public IncidentType Type { get; set; }

[Required]

public IncidentSeverity Severity { get; set; }

[Required]

public string Description { get; set; } = string.Empty;

public string Location { get; set; } = string.Empty;

public string ActionTaken { get; set; } = string.Empty;

public bool RequiresFollowUp { get; set; } = false;

}

}// Services/Interfaces/IExamService.cs

using SchoolManagement.Core.DTOs;

using SchoolManagement.Core.Entities;

namespace SchoolManagement.Services.Interfaces

{

public interface IExamService

{

// Exam Management

Task<ExamDto> CreateExamAsync(CreateExamDto createExamDto, Guid teacherId);

Task<ExamDto> UpdateExamAsync(Guid examId, UpdateExamDto updateExamDto);

Task<bool> DeleteExamAsync(Guid examId);

Task<ExamDto> GetExamByIdAsync(Guid examId);

Task<IEnumerable<ExamDto>> GetExamsBySchoolAsync(Guid schoolId);

Task<IEnumerable<ExamDto>> GetExamsBySubjectAsync(Guid subjectId, Guid termId);

Task<IEnumerable<ExamDto>> GetExamsByGradeAsync(Guid gradeId, Guid termId);

Task<IEnumerable<ExamDto>> GetUpcomingExamsAsync(Guid schoolId, DateTime fromDate, DateTime toDate);

// Exam Publication

Task<bool> PublishExamAsync(Guid examId);

Task<bool> UnpublishExamAsync(Guid examId);

// Student Registration

Task<bool> RegisterStudentForExamAsync(Guid examId, Guid studentId);

Task<bool> RegisterStudentsForExamAsync(Guid examId, List<Guid> studentIds);

Task<bool> UnregisterStudentFromExamAsync(Guid examId, Guid studentId);

Task<IEnumerable<ExamRegistrationDto>> GetExamRegistrationsAsync(Guid examId);

// Exam Sessions

Task<ExamSessionDto> StartExamSessionAsync(StartExamSessionDto startSessionDto);

Task<ExamSessionDto> GetExamSessionAsync(Guid sessionId);

Task<bool> EndExamSessionAsync(Guid sessionId);

Task<bool> PauseExamSessionAsync(Guid sessionId);

Task<bool> ResumeExamSessionAsync(Guid sessionId);

// Answer Submission

Task<bool> SubmitAnswerAsync(SubmitExamAnswerDto submitAnswerDto);

Task<bool> SaveAnswerDraftAsync(SubmitExamAnswerDto saveAnswerDto);

Task<bool> SubmitExamAsync(Guid sessionId);

// Marking & Results

Task<bool> MarkAnswerAsync(Guid answerId, decimal marks, string feedback);

Task<bool> AutoMarkExamAsync(Guid examId);

Task<ExamResultDto> GenerateExamResultAsync(Guid sessionId);

Task<bool> PublishResultsAsync(Guid examId);

Task<IEnumerable<ExamResultDto>> GetExamResultsAsync(Guid examId);

Task<ExamResultDto> GetStudentExamResultAsync(Guid examId, Guid studentId);

// Statistics & Analytics

Task<ExamStatisticsDto> GetExamStatisticsAsync(Guid examId);

Task<IEnumerable<QuestionAnalysisDto>> GetQu