**Task Management System: Comprehensive Project Report**

**Executive Summary**

This report presents the development of a comprehensive Task Management System built using ASP.NET Core Razor Pages. The system implements enterprise-grade features including user authentication, task management, analytics dashboard, recurring tasks, and collaborative task sharing. The application follows modern software development practices with a clean architecture, separation of concerns, and responsive design principles.

**Project Overview**

**Objectives**

* Develop a scalable task management application
* Implement secure user authentication and session management
* Create a functional task CRUD system with filtering and categorization
* Design an intuitive user interface with responsive design
* Implement advanced features: analytics, recurring tasks, and collaboration

**Technologies Utilized**

* **Backend**: ASP.NET Core 9.0, Entity Framework Core
* **Frontend**: HTML5, CSS3, Bootstrap 5, JavaScript
* **Database**: SQLite with EF Core Migrations
* **Development Tools**: Visual Studio, Git

**System Architecture**

The application follows a clear separation of concerns with the following components:

1. **Data Layer**: Entity Framework Core models and DbContext
2. **Service Layer**: Business logic encapsulated in service classes
3. **Presentation Layer**: Razor Pages for UI rendering
4. **Utility Layer**: Helper classes for cross-cutting concerns

// The application's dependency injection setup (Program.cs)

// Register services

builder.Services.AddScoped<IAuthService, AuthService>();

builder.Services.AddScoped<ITaskService, TaskService>();

builder.Services.AddScoped<ITaskAnalyticsService, TaskAnalyticsService>();

builder.Services.AddScoped<IRecurringTaskService, RecurringTaskService>();

builder.Services.AddScoped<ICollaborationService, CollaborationService>();

// Setup a background service to handle recurring tasks

builder.Services.AddHostedService<RecurringTaskBackgroundService>();

**Core Features Implementation**

**User Authentication**

The system implements a secure authentication mechanism with password hashing, salt generation, and session management. The AuthService handles user registration and authentication:

public async Task<User> RegisterUserAsync(string username, string email, string password, string? firstName = null, string? lastName = null)

{

    // Check if username/email already exists

    // Generate salt and hash password

    string salt = PasswordHasher.GenerateSalt();

    string hashedPassword = PasswordHasher.HashPassword(password, salt);

    // Create and store new user

    var user = new User

    {

        Username = username,

        Email = email,

        PasswordHash = hashedPassword,

        PasswordSalt = salt,

        FirstName = firstName,

        LastName = lastName,

        CreatedAt = DateTime.Now,

        IsActive = true

    };

    await \_context.Users.AddAsync(user);

    await \_context.SaveChangesAsync();

    return user;

}

**Task Management**

The system provides comprehensive task management with prioritization, categorization, and filtering capabilities. The TaskService handles task operations:

public async Task<List<UserTask>> GetUserTasksAsync(int userId, bool isCompleted)

{

    return await \_context.Tasks

        .Where(t => t.UserId == userId && t.IsCompleted == isCompleted)

        .OrderByDescending(t => isCompleted ? t.CompletedAt : t.CreatedAt)

        .ToListAsync();

}

public async Task<List<UserTask>> GetUserTasksByPriorityAsync(int userId, Priority priority)

{

    return await \_context.Tasks

        .Where(t => t.UserId == userId && t.Priority == priority)

        .OrderByDescending(t => t.CreatedAt)

        .ToListAsync();

}

## Advanced Features

### Analytics Dashboard

A comprehensive analytics dashboard was implemented to provide users with insights into their tasks. The dashboard displays:

* Tasks distribution by priority
* Tasks distribution by category
* Completion rates and trends
* High-priority and due-soon tasks

// Task analytics by priority (TaskAnalyticsService.cs)

public Dictionary<Priority, int> GetTaskCountByPriority(List<UserTask> tasks)

{

    var priorityCounts = tasks

        .GroupBy(t => t.Priority)

        .ToDictionary(g => g.Key, g => g.Count());

    // Ensure all priority levels are represented

    foreach (Priority priority in Enum.GetValues(typeof(Priority)))

    {

        if (!priorityCounts.ContainsKey(priority))

        {

            priorityCounts[priority] = 0;

        }

    }

    return priorityCounts;

}

**Recurring Tasks System**

The system implements sophisticated recurring task functionality with various recurrence patterns:

// RecurrencePattern.cs - Core model for recurring tasks

public class RecurrencePattern

{

    [Key]

    public int Id { get; set; }

    [Required]

    public int TaskId { get; set; }

    [ForeignKey("TaskId")]

    public UserTask? Task { get; set; }

    [Required]

    public RecurrenceType Type { get; set; } = RecurrenceType.None;

    // Frequency of recurrence (e.g., every 1 day, every 2 weeks)

    [Required]

    public int Interval { get; set; } = 1;

    // For weekly recurrence, the days of the week it occurs (bit flag)

    public int? DaysOfWeek { get; set; }

    // For monthly recurrence, the day of the month or the ordinal week and day

    public int? DayOfMonth { get; set; }

    // For monthly pattern: 1st Monday, 2nd Wednesday, etc.

    public int? WeekOfMonth { get; set; }

    // The date when the recurrence starts

    [Required]

    public DateTime StartDate { get; set; } = DateTime.Today;

    // The date when recurrence ends, null for never

    public DateTime? EndDate { get; set; }

    // Number of occurrences, null for indefinite

    public int? Occurrences { get; set; }

    // Date when the last recurring task was created

    public DateTime? LastGenerated { get; set; }

}

A background service processes and generates recurring tasks based on defined schedules:

// RecurringTaskBackgroundService.cs - Background service that runs daily

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

    \_logger.LogInformation("Recurring Task Background Service is running.");

    // Run once at startup

    await GenerateRecurringTasksAsync();

    // Then run once daily at midnight

    while (!stoppingToken.IsCancellationRequested)

    {

        // Calculate time until next midnight

        var now = DateTime.Now;

        var nextRun = now.Date.AddDays(1);

        var delay = nextRun - now;

        \_logger.LogInformation($"Next recurring task check scheduled for {nextRun.ToString("yyyy-MM-dd HH:mm:ss")}");

        try

        {

            // Wait until next execution time

            await Task.Delay(delay, stoppingToken);

            // Generate recurring tasks

            await GenerateRecurringTasksAsync();

        }

        catch (OperationCanceledException)

        {

            // Normal shutdown

            break;

        }

        catch (Exception ex)

        {

            \_logger.LogError(ex, "Error occurred while generating recurring tasks");

        }

    }

}

### Task Collaboration System

A sophisticated task sharing and collaboration system was implemented to allow users to:

* Share tasks with other users
* Assign different permission levels (Viewer, Editor, Admin)
* Manage collaborators on tasks they own
* View tasks shared with them

The core collaboration model:

// TaskCollaborator.cs - Model that tracks collaborations

public class TaskCollaborator

{

    [Key]

    public int Id { get; set; }

    [Required]

    public int TaskId { get; set; }

    [ForeignKey("TaskId")]

    public UserTask Task { get; set; }

    [Required]

    public int UserId { get; set; }

    [ForeignKey("UserId")]

    public User User { get; set; }

    [Required]

    public CollaboratorRole Role { get; set; }

    public DateTime AddedAt { get; set; } = DateTime.Now;

    public bool CanEdit { get; set; }

}

public enum CollaboratorRole

{

    Viewer,     // Can only view task details

    Editor,     // Can edit task details but not delete

    Admin       // Full control (can delete, add other collaborators)

}

The collaboration service handles permission checks and sharing operations:

// CollaborationService.cs - Core service for collaboration features

public async Task<bool> CanEditTaskAsync(int taskId, int userId)

{

    var task = await \_context.Tasks.FirstOrDefaultAsync(t => t.Id == taskId);

    if (task == null)

        return false;

    // User is the owner

    if (task.UserId == userId)

        return true;

    // User is a collaborator with edit permissions

    var collaborator = await \_context.TaskCollaborators

        .FirstOrDefaultAsync(tc => tc.TaskId == taskId && tc.UserId == userId &&

            (tc.CanEdit || tc.Role == CollaboratorRole.Admin || tc.Role == CollaboratorRole.Editor));

    return collaborator != null;

}

## Database Design

The database schema evolved through multiple migrations as new features were added:

1. Initial migration for users and basic tasks
2. Migration to add recurring tasks support
3. Migration to add collaboration features

The database context configuration ensures proper relationships between entities:

// ApplicationDbContext.cs - Entity relationships configuration

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

    base.OnModelCreating(modelBuilder);

    // Configure User entity

    modelBuilder.Entity<User>()

        .HasIndex(u => u.Email)

        .IsUnique();

    modelBuilder.Entity<User>()

        .HasIndex(u => u.Username)

        .IsUnique();

    // Configure UserTask entity

    modelBuilder.Entity<UserTask>()

        .HasOne(t => t.User)

        .WithMany()

        .HasForeignKey(t => t.UserId)

        .OnDelete(DeleteBehavior.Cascade);

    // Configure RecurrencePattern entity

    modelBuilder.Entity<RecurrencePattern>()

        .HasOne(r => r.Task)

        .WithOne(t => t.RecurrencePattern)

        .HasForeignKey<RecurrencePattern>(r => r.TaskId)

        .OnDelete(DeleteBehavior.Cascade);

    // Configure recurring task instances relationship

    modelBuilder.Entity<UserTask>()

        .HasOne(t => t.ParentTask)

        .WithMany(t => t.RecurrenceInstances)

        .HasForeignKey(t => t.ParentTaskId)

        .OnDelete(DeleteBehavior.Restrict);

    // Configure TaskCollaborator entity

    modelBuilder.Entity<TaskCollaborator>()

        .HasOne(tc => tc.Task)

        .WithMany(t => t.Collaborators)

        .HasForeignKey(tc => tc.TaskId)

        .OnDelete(DeleteBehavior.Cascade);

    modelBuilder.Entity<TaskCollaborator>()

        .HasOne(tc => tc.User)

        .WithMany()

        .HasForeignKey(tc => tc.UserId)

        .OnDelete(DeleteBehavior.Cascade);

}

## Implementation Challenges and Solutions

### Challenge 1: Database Migration Issues

When implementing new models, database migration errors occurred due to pending model changes. This was resolved by properly sequencing migrations and ensuring database synchronization.

### Challenge 2: Nullable Reference Types

Working with C# 9's nullable reference types required careful handling of potentially null values, especially in entity relationships and service methods.

### Challenge 3: Background Services for Recurring Tasks

Implementing the background service for generating recurring tasks required careful consideration of timing, error handling, and database concurrency.

### Challenge 4: Complex Permissions in Collaboration

The task collaboration system needed a sophisticated permission model that required careful design to handle various access scenarios correctly.

## User Interface Design

The user interface was designed with Bootstrap 5 for responsive behavior across devices. Key UI components include:

* A modern navigation system with dropdown menus
* Intuitive task cards with visual priority indicators
* Interactive dashboard with charts and statistics
* Form-based creation and editing of tasks
* Collaboration management interface with role selection

## Conclusion

The Task Management System successfully meets all initial project objectives while incorporating sophisticated features typically found in enterprise applications. The application demonstrates strong software engineering principles including:

* Clean architecture with separation of concerns
* Strong typing and proper error handling
* Responsive and intuitive user interface
* Scalable and maintainable code structure
* Advanced features implementation (analytics, recurrence, collaboration)

The system provides a solid foundation that could be extended for commercial use with minimal additional development. Future enhancements could include integration with external calendars, mobile application development, and AI-powered task suggestions.This project represents a significant achievement in practical software engineering, implementing sophisticated features while maintaining code quality and user experience.