

Development of an Online Platform for Creating and Managing Examination and Evaluation Questionnaires

Ανάπτυξη Διαδικτυακής Πλατφόρμας για τη Δημιουργία και Διαχείριση Ερωτηματολογίων Εξέτασης και Αξιολόγησης

Dissertation

Author

Karakoulak Spyridon

Institution

Hellenic Mediterranean University
School of Engineering
Department of Electronic Engineering

Supervisor

Nikolaos S Petrakis

Date

March 2025

© 2025 Karakoulak Spyridon. All rights reserved.

Table of Contents

Chapter 1: Introduction to the Decenea Examination Platform

- 1.1 Vision and Purpose (p.7)
- 1.2 Key Features and Capabilities (p.7)
- 1.3 Technical Foundation (p.8)
- 1.4 Target Users (p.8)

Chapter 2: Introduction to Modern Software Development with .NET 8 and C#

- 2.1 Fundamentals of Software Development (p.9)
  - 2.1.1 Programming Languages and Compilation (p.9)
- 2.2 Introduction to .NET 8 (p.10)
  - 2.2.1 What is .NET 8? (p.10)
  - 2.2.2 Key Components of .NET 8 (p.10)
- 2.3 Introduction to C# Programming Language (p.11)
  - 2.3.1 Basic C# Concepts (p.11)
  - 2.3.2 Object-Oriented Programming (OOP) in C# (p.12)
- 2.4 Modern Development Features (p.13)
  - 2.4.1 Asynchronous Programming (p.13)
  - 2.4.2 LINQ (Language Integrated Query) (p.13)
  - 2.4.3 Pattern Matching (p.13)
- 2.5 Best Practices and Conventions (p.13)
  - 2.5.1 Coding Standards (p.13)
  - 2.5.2 Project Organization (p.13)
- 2.6 Building Applications with .NET 8 (p.14)
  - 2.6.1 Web Applications (p.14)
  - 2.6.2 Desktop Applications (p.14)
  - 2.6.3 Mobile Development (p.14)
  - 2.6.4 Cloud and Microservices (p.14)
- 2.7 Development Tools and Environment (p.15)
  - 2.7.1 Essential Tools (p.15)
  - 2.7.2 Package Management (p.15)

Chapter 3: Application Architecture and Technology Stack

- 3.1 Clean Architecture Overview (p.16)
  - 3.1.1 Project Structure (p.16)
- 3.2 Core Technologies and Libraries (p.17-37)
  - 3.2.1 Frontend Technologies (p.17)
  - 3.2.2 Backend Architecture (p.20)
  - 3.2.3 Data Persistence (p.23)
  - 3.2.4 Background Processing (p.28)
  - 3.2.5 API Integration (p.28)
  - 3.2.6 Documentation and Testing (p.29)
  - 3.2.7 Validation (p.29)
  - 3.2.8 Logging and Monitoring (p.30)
  - 3.2.9 Authentication and Authorization (p.30)
  - 3.2.10 Domain-Driven Design (p.32)
  - 3.2.11 Error Handling (p.35)
- 3.3 Advanced Architectural Patterns (p.38-43)
  - 3.3.1 Outbox Pattern (p.38)
  - 3.3.2 Options Pattern (p.40)
  - 3.3.3 Database Migrations on Startup (p.42)
  - 3.3.4 Integration Points (p.43)
  - 3.3.5 ID Generation with ULID (p.43)
  - 3.3.6 Entity Versioning, Concurrency and Version Control (p.43)
- 3.4 Database Schema (p.44-49)

Chapter 4: Functionality

- 4.1 Web API Functionality (p.50-55)
  - 4.1.1 Response Format (p.50)
  - 4.1.2 Authentication (p.50)
  - 4.1.3 User Management (p.51)
  - 4.1.4 Group Management (p.51)
  - 4.1.5 Test Management (p.52)
  - 4.1.6 Question Management (p.54)
  - 4.1.7 Answer Management (p.55)
- 4.2 Web Application Functionality (p.56-63)
  - 4.2.1 Login Page (p.56)
  - 4.2.2 Register Page (p.56)
  - 4.2.3 Home Page (p.57)
  - 4.2.4 Contact Form (p.57)
  - 4.2.5 Copyright Page (p.58)
  - 4.2.6 Question Types Page (p.58)
  - 4.2.7 Groups Page (p.59)
  - 4.2.8 Edit Group Page (p.59)
  - 4.2.9 My Test Page (p.60)
  - 4.2.10 Create Test Page (p.60)
  - 4.2.11 Add Question Edit (p.61)
  - 4.2.12 Add Question Preview (p.61)
  - 4.2.13 Update Test Question (p.62)
  - 4.2.14 Update Test User Group (p.62)
  - 4.2.15 Submit Test Page (p.63)

Chapter 5: Conclusion

- 5.1 Summary of Achievements (p.64)
- 5.2 Technical Innovations (p.64)
- 5.3 Limitations and Future Work (p.65)
- 5.4 Lessons Learned (p.65)
- 5.5 Final Remarks (p.66)

Go to Sources (p.67-70)

Abstract

This dissertation presents the development of a comprehensive online platform for creating and managing examination and evaluation questionnaires using modern software development technologies. Built on .NET 8 and C#, the platform implements a clean architecture pattern with domain-driven design principles to ensure scalability, maintainability, and robust security.

The system features a sophisticated web application developed using Blazor WebAssembly, providing a responsive and interactive user interface through the MudBlazor component library. The backend architecture employs the CQRS pattern with FastEndpoints for efficient API development, while PostgreSQL with Entity Framework Core handles data persistence. The platform supports various question types, including multiple-choice, drag-and-drop, and fill-in-the-blank formats, offering flexible assessment capabilities.

Key technical innovations include implementation of the Outbox pattern for reliable event processing, optimistic concurrency control using entity versioning, and comprehensive audit trailing. The system employs JWT-based authentication with refresh token support and implements role-based access control for secure user management. The platform's architecture demonstrates practical application of modern software development practices, including asynchronous programming, LINQ, and pattern matching.

This work contributes to the field of educational technology by providing a robust, scalable solution for online assessment management while showcasing the effective implementation of contemporary software architecture patterns and development practices.

Περίληψη

Η παρούσα διπλωματική εργασία παρουσιάζει την ανάπτυξη μιας ολοκληρωμένης διαδικτυακής πλατφόρμας για τη δημιουργία και διαχείριση ερωτηματολογίων εξέτασης και αξιολόγησης, χρησιμοποιώντας σύγχρονες τεχνολογίες ανάπτυξης λογισμικού. Βασισμένη στο .NET 8 και C#, η πλατφόρμα υλοποιεί ένα μοτίβο καθαρής αρχιτεκτονικής με αρχές σχεδίασης που καθοδηγούνται από τον τομέα (domain-driven design) για τη διασφάλιση της κλιμάκωσης, της συντηρησιμότητας και της ισχυρής ασφάλειας.

Το σύστημα διαθέτει μια εξελιγμένη διαδικτυακή εφαρμογή που αναπτύχθηκε χρησιμοποιώντας το Blazor WebAssembly, παρέχοντας ένα αποκρίσιμο και διαδραστικό περιβάλλον χρήστη μέσω της βιβλιοθήκης στοιχείων MudBlazor. Η αρχιτεκτονική του συστήματος χρησιμοποιεί το μοτίβο CQRS με FastEndpoints για αποτελεσματική ανάπτυξη API, ενώ η PostgreSQL με το Entity Framework Core διαχειρίζεται την παραμονή των δεδομένων. Η πλατφόρμα υποστηρίζει διάφορους τύπους ερωτήσεων, συμπεριλαμβανομένων των μορφών πολλαπλής επιλογής, μεταφοράς και απόθεσης, και συμπλήρωσης κενών, προσφέροντας ευέλικτες δυνατότητες αξιολόγησης.

Βασικές τεχνικές καινοτομίες περιλαμβάνουν την υλοποίηση του μοτίβου Outbox για αξιόπιστη επεξεργασία συμβάντων, έλεγχο αισιόδοξης ταυτοχρονικότητας χρησιμοποιώντας έκδοση οντοτήτων και ολοκληρωμένη παρακολούθηση ελέγχου. Το σύστημα χρησιμοποιεί έλεγχο ταυτότητας βασισμένο σε JWT με υποστήριξη ανανέωσης διακριτικών και υλοποιεί έλεγχο πρόσβασης βάσει ρόλων για ασφαλή διαχείριση χρηστών. Η αρχιτεκτονική της πλατφόρμας επιδεικνύει την πρακτική εφαρμογή σύγχρονων πρακτικών ανάπτυξης λογισμικού, συμπεριλαμβανομένου του ασύγχρονου προγραμματισμού, LINQ και αντιστοίχισης μοτίβων.

Η εργασία αυτή συμβάλλει στον τομέα της εκπαιδευτικής τεχνολογίας παρέχοντας μια ισχυρή, κλιμακώσιμη λύση για τη διαχείριση διαδικτυακής αξιολόγησης, ενώ παράλληλα επιδεικνύει την αποτελεσματική εφαρμογή σύγχρονων μοτίβων αρχιτεκτονικής λογισμικού και πρακτικών ανάπτυξης.

Chapter 1: Introduction to the Decenea Examination Platform

1.1 Vision and Purpose

The Decenea platform was conceived as a comprehensive solution for creating and managing examination and evaluation questionnaires. What began as a personal project to track progress while preparing for certification exams has evolved into a fully-featured assessment platform that addresses the needs of both educators and learners in today's digital education landscape.

At its core, Decenea aims to provide a flexible, user-friendly environment where educators can create diverse assessment materials while learners can engage with these materials in a structured and meaningful way. The platform supports various question types—from multiple choice and fill-in-the-blank to more complex drag-and-drop and ordering questions—enabling educators to design assessments that truly measure understanding rather than just recall.

1.2 Key Features and Capabilities

Decenea distinguishes itself through several key capabilities:

1. **Diverse Question Types**: The platform supports a wide range of question formats, allowing educators to test different cognitive skills and knowledge areas appropriately.
2. **Group-Based Organization**: Users can create and manage groups, facilitating collaboration and enabling targeted assessment distribution to specific cohorts.
3. **Timed Assessments**: Tests can be configured with time limits, simulating the pressure of real examination environments and preparing students for time-constrained scenarios.
4. **Progress Tracking**: Both educators and learners can track performance over time, identifying strengths and areas for improvement.
5. **Secure Authentication**: A robust security model ensures that assessment content remains protected and that user data is handled responsibly.
6. **Responsive Design**: The Blazor WebAssembly frontend provides a seamless experience across devices, making the platform accessible whether in a classroom or studying remotely.

1.3 Technical Foundation

Built on the modern .NET 8 stack with C#, Decenea leverages Clean Architecture principles to ensure maintainability and scalability. The application employs Domain-Driven Design to model complex educational processes and assessment workflows, while the CQRS pattern enables efficient data operations.

The PostgreSQL database provides a reliable foundation for data persistence, while the JWT authentication system ensures secure access to resources. The use of Blazor WebAssembly for the frontend allows for rich, interactive question types that closely mimic the experience of traditional paper assessments while adding the benefits of immediate feedback and automated scoring.

1.4 Target Users

Decenea is designed to serve multiple stakeholders in the educational ecosystem:

- **Educators and Trainers**: Who need to create assessments, manage student groups, and analyze performance data
- **Students and Learners**: Who engage with assessments and track their progress over time
- **Educational Institutions**: That require a centralized platform for managing examinations across courses and departments
- **Professional Certification Bodies**: That need to administer practice and actual certification examinations in a secure environment
- **Corporate Training Departments**: That assess employee knowledge and skill development through structured evaluations

By addressing the needs of these diverse user groups, Decenea aims to become an essential tool in modern education and training environments, bridging the gap between traditional assessment methods and the digital learning landscape.

Chapter 2: Introduction to Modern Software Development with .NET 8 and C#

2.1 Fundamentals of Software Development

Before diving into .NET 8 and C#, it's essential to understand the basic concepts of software development. Software is a collection of instructions that tell a computer how to perform specific tasks. These instructions are written in programming languages, which bridge the gap between human-readable commands and machine code that computers can execute.

2.1.1 Programming Languages and Compilation

Programming languages come in different levels of abstraction:

- Low-level languages work directly with computer hardware
- High-level languages, like C#, use human-readable syntax
- Intermediate languages serve as a bridge between high-level code and machine code

The process of converting human-written code into executable instructions involves:

1. Writing source code in a text editor
2. Compiling the code into intermediate language
3. Converting intermediate language into machine code
4. Executing the resulting program

The compilation process in .NET is a sophisticated multi-step procedure that transforms human-readable code into machine-executable instructions. When developers write C# code, it first gets compiled into an intermediate language (IL) that's platform-independent. This IL code is then compiled again at runtime by the Just-In-Time (JIT) compiler into machine code specific to the target platform. This two-step compilation process is key to .NET's cross-platform capabilities and performance optimization.

2.2 Introduction to .NET 8

.NET 8 is a modern, open-source development platform created by Microsoft. It provides all the tools and libraries needed to build software applications.

2.2.1 What is .NET 8?

.NET 8 represents a complete development ecosystem that includes:

- A runtime environment for executing programs
- A comprehensive set of programming libraries
- Development tools and utilities
- Framework for building different types of applications

Unlike its predecessor (.NET Framework), .NET 8 is:

- Cross-platform: runs on Windows, Linux, and macOS
- Open-source: community can contribute and inspect code
- Modern: designed for current computing needs
- Unified: single platform for all development scenarios

2.2.2 Key Components of .NET 8

The platform consists of several essential components:

1. Runtime Environment
    - Manages program execution
    - Handles memory management
    - Provides security features
    - Manages program threads
    - Performs error handling
2. Base Class Libraries
    - Collections for data management
    - File system operations
    - Network communication
    - Security functions
    - Data access capabilities
3. Development Tools
    - Command-line interface (CLI)
    - Visual Studio IDE
    - Code editors
    - Debugging tools

2.3 Introduction to C# Programming Language

C# is the primary programming language for .NET 8. It's designed to be simple, modern, and versatile.

2.3.1 Basic C# Concepts

Program Structure

The Program.cs file serves as the central configuration and startup point in modern .NET applications, particularly in .NET 8. It implements the minimal hosting model, which reduces boilerplate code while maintaining flexibility. This file is responsible for configuring services, middleware, and routing that your application needs.

Mastering its structure allows you to build efficient, scalable, and robust applications, whether you're creating a simple API, a web app, or using advanced features like background services.

The minimal API approach shown in the example demonstrates how to create lightweight, performance-focused web applications without the overhead of traditional MVC architecture.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// Configure the HTTP request pipeline (optional).
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseHttpsRedirection();

// Define a simple API endpoint.
app.MapGet("/", () => "Hello, World!");

// Run the application.
app.Run();
```

1. Variables and Data Types

```
// Basic data types
int number = 42;                // Whole numbers
double price = 19.99;           // Decimal numbers
string name = "John";           // Text
bool isActive = true;           // True/false values
DateTime today = DateTime.Now;   // Date and time
```

2.3.2 Object-Oriented Programming (OOP) in C#

The Group class demonstrates several key OOP principles (Classes, Objects, Inheritance and Polymorphism) in C#. It uses encapsulation through private fields with public readonly collections, implements the aggregate root pattern from Domain-Driven Design, and shows how to manage entity relationships in a domain model.

```
public class Group : AuditableAggregateRoot
{
    // Constructor
    public Group(string? id)
    {
        Id = id ?? Ulid.NewUlid().ToString();
    }
    // Properties
    public required string Name { get; set; }
    private readonly List<GroupMember> _groupMembers = new ();
    public IReadOnlyCollection<GroupMember> GroupMembers =>
_groupMembers.AsReadOnly();
    private List<TestGroup> _testGroups = new ();
    public IReadOnlyCollection<TestGroup> TestGroups  =>
_testGroups.AsReadOnly();
    public static Group Create(string name, string? id = null)
    {
        return new Group(id) { Name = name };
    }
    // Methods
    public void SyncTests(List<string> testIds)
    {
        _testGroups = testIds.Select(i => new TestGroup()
        {
            TestId = i,
            GroupId = this.Id
        }).ToList();
    }
    public void AddNewGroupMember(string userEmail, string groupId,
GroupRole groupRole, string? alias = null)
    {
        _groupMembers.Add(new GroupMember()
        {
            GroupRole = groupRole,
            GroupUserEmail = userEmail,
            GroupId = groupId,
            Alias = alias
        });
    }
}
```

2.4 Modern Development Features

.NET 8 introduces several modern programming concepts:

2.4.1 Asynchronous Programming

```
public async Task<string> GetDataAsync()
{
    // Simulating network request
    await Task.Delay(1000);
    return "Data retrieved";
}
```

2.4.2 LINQ (Language Integrated Query)

```
var numbers = new List<int> { 1, 2, 3, 4, 5 };
var evenNumbers = numbers.Where(n => n % 2 == 0);
```

2.4.3 Pattern Matching

```
object value = "Hello";
if (value is string message)
{
    Console.WriteLine(message.ToUpper());
}
```

2.5 Best Practices and Conventions

Modern .NET development follows certain conventions:

2.5.1 Coding Standards

- Use PascalCase for class names and public members
- Use camelCase for local variables and parameters
- Prefix interfaces with 'I' (IDisposable)
- Use meaningful and descriptive names
- Follow the Single Responsibility Principle

2.5.2 Project Organization

- Separate concerns into different projects
- Use solution folders for organization
- Maintain clear dependency hierarchy
- Implement dependency injection

2.6 Building Applications with .NET 8

.NET 8 supports various application types:

2.6.1 Web Applications

- ASP.NET Core for web applications
- Web APIs for service development
- Blazor for interactive web UIs
- Minimal APIs for microservices

2.6.2 Desktop Applications

- Windows Forms for traditional desktop apps
- WPF for modern Windows applications
- .NET MAUI for cross-platform desktop apps

2.6.3 Mobile Development

- .NET MAUI for iOS and Android
- Xamarin integration
- Cross-platform mobile development

2.6.4 Cloud and Microservices

- Azure cloud integration
- Docker containerization
- Kubernetes orchestration
- Microservices architecture

2.7 Development Tools and Environment

To start developing with .NET 8:

2.7.1 Essential Tools

1. Visual Studio 2022
    - Integrated development environment
    - Code editor with intelligence
    - Debugging tools
    - Testing framework
    - Source control integration
2. Visual Studio Code
    - Lightweight code editor
    - Cross-platform support
    - Extension marketplace
    - Integrated terminal
3. Command Line Tools

```
dotnet new console     # Create new console application
dotnet build          # Build the project
dotnet run            # Run the application
```

1.7.2 Package Management

NuGet is .NET's package manager:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet restore
```

Chapter 3: Application Architecture and Technology Stack

3.1 Clean Architecture Overview

The application follows Robert C. Martin's Clean Architecture principles, organizing code into concentric layers with dependencies pointing inward. This architectural pattern ensures:

- Independence from frameworks
- Testability
- Readability
- Independence from UI
- Independence from Database
- Independence from external agencies

3.1.1 Project Structure

```
Decenea/
├── Decenea.Domain           # Enterprise Business Rules
├── Decenea.Application      # Application Business Rules
├── Decenea.Infrastructure   # Frameworks & Drivers
├── Decenea.WebAPI          # API Interface
├── Decenea.WebApp          # Blazor WebAssembly UI
└── Decenea.Common          # Shared DTOs & Contracts
```

3.2 Core Technologies and Libraries

3.2.1 Frontend Technologies

Blazor WebAssembly

Blazor WebAssembly (WASM) is a modern web framework by Microsoft that enables developers to build interactive web applications using C# and .NET instead of JavaScript.

1. Browser-based .NET Runtime

- Runs a complete .NET runtime directly in the browser via WebAssembly
- Executes compiled C# code natively in the browser
- Provides near-native performance for computational tasks

2. Development Model

- Component-based architecture similar to React/Angular
- Uses Razor syntax combining HTML with C# code
- Supports two-way data binding
- Built-in dependency injection
- Rich component lifecycle management
- State management through cascading parameters

3. Integration Capabilities

- JavaScript interoperability for using existing JS libraries
- Access to browser APIs through JS interop
- REST API consumption using HttpClient
- SignalR support for real-time communications
- Authentication/Authorization integration with Identity

4. Code Sharing

- Shared business logic between client and server
- Common validation rules
- Shared data models and DTOs
- Reusable components across projects
- Single language (C#) throughout the stack

5. Performance Considerations

- Initial download size includes .NET runtime
- Progressive Web App (PWA) support
- Lazy loading capabilities
- Optimized for subsequent visits through caching
- Prerendering options for better initial load

6. Development Experience

- Full IntelliSense support
- Hot reload capabilities
- Integrated debugging
- Unit testing support
- Access to NuGet package ecosystem

7. Hosting Options

- Static file hosting
- Azure Static Web Apps
- GitHub Pages
- Any static file server
- Can be hosted alongside ASP.NET Core

8. Security Features

- Built-in XSS protection
- CSRF protection
- Authentication providers
- Authorization policies
- Secure cookie handling

Common Use Cases

1. Line-of-business applications
2. Single-page applications (SPAs)
3. Progressive Web Apps (PWAs)
4. Internal enterprise applications
5. Forms-heavy applications

Considerations

- Initial load time due to runtime download
- Not ideal for SEO-critical public websites (unless using prerendering)
- Browser compatibility requirements
- Learning curve for developers new to .NET
- Limited debugging capabilities

MudBlazor

- Material Design component library
- Responsive layout system
- Rich set of pre-built components

The provided timer component demonstrates how Blazor components manage their lifecycle and state. It implements IDisposable for proper resource cleanup, uses parameters for configuration, and shows how to handle real-time updates using Timer. The component demonstrates Blazor's reactive programming model where UI updates are triggered by state changes through StateHasChanged(). This pattern is fundamental to building responsive, real-time web applications in Blazor.

```
@inherit IDisposal

<MudCard>
    <MudCardContent>
        <MudText>Current count: @currentCount</MudText>
    </MudCardContent>
</MudCard>

@code {
    [Parameter] public int TotalTimeInSeconds { get; set; }
    [Parameter] public EventCallback<bool> Finished { get; set; }
    private int ElapsedTime { get; set; }
    private int TimeRemaining => TotalTimeInSeconds - ElapsedTime;
    private TimeSpan TimeSpan => TimeSpan.FromSeconds(TimeRemaining);
    private string FormattedTimeRemaining => string.Format("{0:D2}:
{1:D2}", (int)TimeSpan.TotalMinutes, TimeSpan.Seconds);
    private Timer? Timer { get; set; }
    protected override Task OnInitializedAsync()
    {
        Timer = new Timer(OnTimerElapsed, null, TimeSpan.Zero,
TimeSpan.FromSeconds(1));
        return base.OnInitializedAsync();
    }
    public void Dispose()
    {
        Timer?.Dispose();
    }
    private void OnTimerElapsed(object? state)
    {
        if (TimeRemaining <= 0)
        {
            Finished.InvokeAsync(true);
            Timer?.Dispose();
        }
        else
        {
            ElapsedTime++;
            InvokeAsync(StateHasChanged);
        }
    }
}
```

3.2.2 Backend Architecture

FastEndpoints

FastEndpoints is a High-performance REST API development library built on top of minimal APIs, where each endpoint is self-contained with its request/response models and business logic also known as REPR Pattern (Request-Endpoint-Response).

The GetGroupEndpoint example demonstrates how to implement role-based security, handle JWT claims, and return consistent API responses. This approach reduces cross-cutting concerns and makes endpoints more maintainable compared to traditional controller-based architectures.

```
public class GetGroupEndpoint : Endpoint<GetGroupRequest,
ApiResponseResult<GroupDto>>
{
    public override void Configure()
    {
        Get(RouteConstants.GroupsGet);
        Roles(UserRoleExtensions.GetAuthorizedRoles());
    }

    public override async Task<ApiResponseResult<GroupDto>>
ExecuteAsync(GetGroupRequest req, CancellationToken ct)
    {
        var userEmail =
HttpContext.User.FindFirst(ClaimTypes.Email)?.Value;

        if(userEmail is null)
            return new ApiResponseResult<GroupDto>(null, true, "Invalid
JWT.");

        var result = await new GetGroupQuery()
        {
            UserEmail = userEmail,
            GroupId = req.Id
        }.ExecuteAsync(ct);
        return new ApiResponseResult<GroupDto>(result.Value ,
result.IsError, result.ErrorsOrEmptyList.ToErrorDictionary());
    }
}
```

CQRS (Command Query Responsibility Segregation)

CQRS (Command Query Responsibility Segregation) is an architectural pattern that separates an application's read operations from its write operations. This separation is based on the principle that using the same models for both reading and writing data can lead to unnecessary complexity as applications grow.

> Command Side:

- Commands represent intentions to change the application's state (e.g., CreateOrder, UpdateCustomerAddress)
- They are task-based rather than data-centric
- Commands can be queued and processed asynchronously
- Typically optimized for write operations with simpler data models
- Can incorporate validation, business rules, and domain logic
- Usually returns void or operation status, not data

> Query Side:

- Handles all read operations through specialized data models
- Can be optimized for specific query requirements
- Often uses denormalized views of data for better read performance
- Can utilize different storage technologies optimized for reading
- Returns data but never modifies state
- Can be scaled independently from the command side

> Key Benefits:

- Performance Optimization: Each side can be tuned for its specific purpose. The read side can use denormalized data while the write side maintains data consistency.
- Scalability: Read and write workloads can be scaled independently. This is particularly useful when read operations significantly outnumber writes.
- Security: Easier to implement different security models for reads vs. writes.
- Flexibility: Freedom to evolve the read and write models independently as requirements change.

```
public class GetGroupQuery : ICommand<ErrorOr<GroupDto>>
{
    public string GroupId { get; set; }
    public string UserEmail { get; set; }
}

public class GetGroupQueryHandler : ICommandHandler<GetGroupQuery,
ErrorOr<GroupDto>>
{
    private readonly IDeceneaDbContext _dbContext;

    public GetGroupQueryHandler(IDeceneaDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task<ErrorOr<GroupDto>> ExecuteAsync(GetGroupQuery
command, CancellationToken ct)
    {
        var group = await _dbContext
            .Set<Domain.Aggregates.GroupAggregate.Group>()
            .Include(i => i.GroupMembers)
            .Include(i => i.TestGroups)
            .ThenInclude(i => i.Test)
            .AsSplitQuery()
            .FirstOrDefaultAsync(i => i.Id == command.GroupId
                                      && i.GroupMembers.Any(j =>
j.GroupUserEmail == command.UserEmail), ct);

        if(group is null)
            return Error.NotFound(description: "Group not found.");

        return group.GroupToGroupDto(true);
    }
}
```

3.2.3 Data Persistence

PostgreSQL with Entity Framework Core

- Open-source relational database
- Rich feature set including JSON support
- EF Core for ORM capabilities
- Code-first migrations

The custom DbContext implementation shows advanced Entity Framework Core patterns including audit logging, domain event handling, and optimistic concurrency control. It implements the Unit of Work pattern with transactional consistency for domain events, ensuring that database changes and event publishing either both succeed or both fail. The context also handles automatic audit trail generation for all entity changes, supporting compliance and debugging requirements.

```
internal class DeceneaDbContext : DbContext, IDeceneaDbContext
{
    public string? ModifiedBy { get; set; }
    private Queue<IDomainEvent>? DomainEvents { get; set; }

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

    protected sealed override void
ConfigureConventions(ModelConfigurationBuilder configurationBuilder)
    {
        base.ConfigureConventions(configurationBuilder);
        configurationBuilder.Properties<DateTime>()
            .HaveConversion(typeof(DateTimeToDateTimeUtc));
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
        ApplicationUserSeed.Seed(builder);
        builder.ApplyConfigurations();
    }

    public new DbSet<T> Set<T>() where T : Versioned
    {
        return base.Set<T>();
    }

    public new async Task<ErrorOr<object>>
SaveChangesAsync(CancellationToken cancellationToken = default)
    {
        return await SaveChangesAsync(null, cancellationToken);
    }
```

```
    public async Task<ErrorOr<object>> SaveChangesAsync(string? modifiedBy
= null,
        CancellationToken cancellationToken = default)
    {
        modifiedBy ??= ModifiedBy;
        if (modifiedBy is null)
            return Error.Failure(description: "Unable to save changes.");

        DomainEvents ??= GetDomainEvents();

        if (DomainEvents.Count == 0)
        {
            await ProcessAuditableEntities(modifiedBy);
            try
            {
                await base.SaveChangesAsync(cancellationToken);
            }
            catch (DbUpdateConcurrencyException ex)
            {
                //Handle concurrency exception
                return Error.Conflict(description: "Unable to save changes
due to invalid data.");
            }
        }

        return await HandleDomainEvents(DomainEvents, modifiedBy,
cancellationToken);
    }

    private async Task<ErrorOr<object>>
HandleDomainEvents(Queue<IDomainEvent> domainEvents, string userId,
        CancellationToken cancellationToken = default)
    {
        await using var transaction = await
Database.BeginTransactionAsync(cancellationToken);
        try
        {
            while (domainEvents.TryDequeue(out var nextEvent))
            {
                await nextEvent.PublishAsync(Mode.WaitForAll,
cancellationToken);
            }

            await base.SaveChangesAsync(cancellationToken);

            await transaction.CommitAsync(cancellationToken);
            return "Successful process.";
        }
        catch (Exception ex)
        {
            await transaction.RollbackAsync(cancellationToken);
            await AddDomainEventsAsOutboxMessages(domainEvents, userId,
ex);
```

```
            Log.Error("Something went wrong while publishing events:
{message} . Adding them to the outbox.",
                ex.Message);
            await base.SaveChangesAsync(cancellationToken);
            return Error.Unexpected(description: "Unable to Handle
DomainEvents in DbContext.");
        }
    }

    private Queue<IDomainEvent> GetDomainEvents()
    {
        return new Queue<IDomainEvent>(ChangeTracker
            .Entries<AggregateRoot>()
            .Select(entry => entry.Entity)
            .SelectMany(ar => { return ar.PopDomainEvents(); }));
    }

    private async Task AddDomainEventsAsOutboxMessages(Queue<IDomainEvent>
domainEvents, string userId,
        Exception exception)
    {
        try
        {
            var dateTime = DateTime.UtcNow;
            var outboxMessages = domainEvents
                .Select(domainEvent => new OutboxMessage()
                {
                    OccurredOnUtc = dateTime,
                    Type = domainEvent.GetType().Name,
                    DomainEvent = JsonSerializer.Serialize(domainEvent),
                    CreatedBy = userId,
                    Error = exception.Message
                })
                .ToList();

            await AddRangeAsync(outboxMessages);
        }
        catch (Exception ex)
        {
            Log.Error("There was something wrong while
AddDomainEventsAsOutboxMessages : {ex}", ex);
        }
    }

    private async Task ProcessAuditableEntities(string createdBy)
    {
        try
        {
            var entityEntryList = ChangeTracker.Entries()
                .Where(x => x.State is EntityState.Modified or
EntityState.Added or EntityState.Deleted)
                .ToList();

            var dateTimeUtcNow = DateTime.UtcNow;
```

```
            foreach (var entityEntry in entityEntryList)
            {
                if (entityEntry.Entity is IAuditableEntity auditable)
                {
                    if (entityEntry.State == EntityState.Added)
                    {
                        auditable.CreatedBy = createdBy;
                        auditable.CreatedByTimestampUtc = dateTimeUtcNow;
                    }
                    else if (entityEntry.State == EntityState.Modified)
                    {
                        auditable.Version =
RandomStringGenerator.RandomString(8);
                    }

                    auditable.LastModifiedBy = createdBy;
                    auditable.LastModifiedByTimestampUtc = dateTimeUtcNow;
                }

                var entityId = GetEntityKeyString(entityEntry);
                if (!string.IsNullOrWhiteSpace(entityId))
                {
                    var auditLog = new AuditLog()
                    {
                        EntityId = entityId,
                        EntityType =
entityEntry.Entity.GetType().ToString(),
                        ExecutedOperation =
(ExecutedOperation)entityEntry.State,
                        OperationExecutedAt = dateTimeUtcNow,
                        DataAfterExecutedOperation =
JsonSerializer.Serialize(entityEntry.Entity,
AuditLogJsonSerializerOptions.JsonSerializerOptions)
                    };

                    await AddAsync(auditLog);
                }
                else
                {
                    Log.Error("The entityId for: {entityEntry} was not
found", entityEntry);
                }
            }
        }
        catch (Exception ex)
        {
            Log.Error("There was something wrong while
ProcessAuditableEntities : {ex}", ex);
        }
    }

    private string GetEntityKeyString(EntityEntry entityEntry)
    {
```

```
        var keyProperties = entityEntry
            .Metadata
            .FindPrimaryKey()?
            .Properties;

        if (keyProperties == null || !keyProperties.Any())
            return string.Empty;

        var keyValues = new List<object>();
        foreach (var property in keyProperties)
        {
            var value = entityEntry.Property(property.Name).CurrentValue
                        ??
entityEntry.Property(property.Name).OriginalValue;

            // For new entities, use the local value if available
            if (entityEntry.State == EntityState.Added)
            {
                value = entityEntry.Property(property.Name).CurrentValue;
            }

            keyValues.Add(value ?? "");
        }

        if (keyValues.Count == 1)
            return keyValues[0].ToString() ?? string.Empty;

        return JsonSerializer.Serialize(keyValues);
    }
}

file static class AuditLogJsonSerializerOptions
{
    public static JsonSerializerOptions JsonSerializerOptions = new
JsonSerializerOptions
        {
            ReferenceHandler = ReferenceHandler.Preserve,
            MaxDepth = 64
        };
}
```

3.2.4 Background Processing

Quartz.NET

- Job scheduling framework
- Cron-like scheduling
- Persistent job storage
- Clustering support

```
public class DataCleanupJob : IJob
{
    public async Task Execute(IJobExecutionContext context)
    {
        // Cleanup implementation
    }
}

// Job configuration
services.AddQuartz(q =>
{
    var jobKey = new JobKey("DataCleanup");
    q.AddJob<DataCleanupJob>(opts => opts.WithIdentity(jobKey));
    q.AddTrigger(opts => opts
        .ForJob(jobKey)
        .WithCronSchedule("0 0 12 * * ?"));
});
```

3.2.5 API Integration

Refit

- Type-safe REST library
- Automatic HTTP client generation
- Interface-based API definitions

```
public interface IUserApi
{
    [Get("/api/users/{id}")]
    Task<UserDto> GetUserAsync(Ulid id);

    [Post("/api/users")]
    Task<UserDto> CreateUserAsync([Body] CreateUserRequest request);
}
```

3.2.6 Documentation and Testing

OpenAPI (Swagger)

- API documentation
- Interactive testing interface
- Code generation
- Client SDK generation

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "Decenea API",
        Version = "v1"
    });
});
```

3.2.7 Validation

FluentValidation

- Fluent interface for validation rules
- Complex validation scenarios
- Custom validation rules
- Integration with FastEndpoints

```
internal sealed class RegisterUserCommandValidator :
Validator<RegisterUserCommand>
{
    public RegisterUserCommandValidator()
    {
        RuleFor(c => c.FirstName).NotEmpty()
            .WithMessage("Your first name is required!");

        RuleFor(c => c.LastName).NotEmpty()
            .WithMessage("Your last name is required!");

        RuleFor(c => c.MiddleName).NotEmpty()
            .WithMessage("Your middle name is required!");

        RuleFor(c => c.Email).EmailAddress();

        RuleFor(c => c.Password).NotEmpty()
            .MinimumLength(8);
    }
}
```

3.2.8 Logging and Monitoring

Serilog

- Structured logging
- Multiple output sinks
- Log enrichment
- Level-based filtering

```
Log.Logger = new LoggerConfiguration()
    .WriteTo.Console()
    .WriteTo.PostgreSQL(
        connectionString,
        "Logs",
        needAutoCreateTable: true)
    .CreateLogger();
```

3.2.9 Authentication and Authorization

JWT Authentication

Authentication and Authorization in modern applications encompass several critical security mechanisms that work together to ensure secure access control. JWT (JSON Web Token) Authentication provides a stateless, token-based approach where encoded JSON tokens carry user identity and claims information between the client and server. These tokens enable Token-based authentication, allowing users to access protected resources without repeatedly providing credentials. Claims-based authorization extends this by making access decisions based on token claims (attributes) about the user or entity. Role-based access control (RBAC) implements a hierarchical permission system where access rights are grouped into roles that can be assigned to users. Finally, secure token storage involves protecting these authentication tokens on both client and server sides through encryption, secure HTTP-only cookies, and other safety measures to prevent token theft and unauthorized access.

> A typical flow looks like this:

- User logs in and receives both an access token and refresh token
- Access token is used for API requests until it expires
- When expired, the refresh token is used to obtain a new access token
- This continues until the refresh token expires or is revoked
- At that point, the user needs to log in again

This two-token system balances security (short-lived access tokens) with user experience (not having to frequently re-authenticate).

> Access tokens and refresh tokens work together in a complementary security system:

Access Token:

A short-lived credential (typically valid for minutes to hours) that allows access to protected resources Contains encoded user information and permissions (claims) Sent with each API request in the Authorization header When it expires, the user would normally need to re-authenticate Kept short-lived to minimize the security risk if the token is compromised

Refresh Token:

A longer-lived credential (valid for days to weeks) used to obtain new access tokens Stored securely (usually in an HTTP-only cookie or secure storage) Only sent to specific authentication endpoints, not with regular API requests When the access token expires, the refresh token can be used to get a new access token without requiring re-authentication If compromised, can be revoked server-side to force re-authentication

```
services
            .AddAuthentication(o => o.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme)
            .AddJwtBearer(o =>
            {
                SecurityKey key = new
SymmetricSecurityKey(Encoding.ASCII.GetBytes(configuration["Auth:JWTSignin
gKey"]!));

                //set defaults
                o.TokenValidationParameters.IssuerSigningKey = key;
                o.TokenValidationParameters.ValidateIssuerSigningKey =
true;
                o.TokenValidationParameters.ValidateLifetime = true;
                o.TokenValidationParameters.ClockSkew =
TimeSpan.FromSeconds(60);
                o.TokenValidationParameters.ValidAudience = null;
                o.TokenValidationParameters.ValidateAudience = false;
                o.TokenValidationParameters.ValidIssuer = null;
                o.TokenValidationParameters.ValidateIssuer = false;

                //correct any user mistake
                o.TokenValidationParameters.ValidateAudience =
o.TokenValidationParameters.ValidAudience is not null;
                o.TokenValidationParameters.ValidateIssuer =
o.TokenValidationParameters.ValidIssuer is not null;
            });
```

3.2.10 Domain-Driven Design

Domain-Driven Design is a software development approach introduced by Eric Evans in 2003 that focuses on modeling software to match a domain according to input from domain experts. The fundamental premise is that the primary focus of a software project should be the core domain and domain logic.

Key concepts and principles:

> **Benefits:**

Better alignment between business needs and software implementation Reduced complexity through clear boundaries and contexts Improved communication between technical and non-technical team members More maintainable and flexible codebase Better handling of complex business rules and processes

DDD is particularly valuable for complex domains with sophisticated business rules. However, it requires significant investment in domain modeling and might be overengineering for simple applications or systems with minimal business logic. The approach emphasizes continuous learning and collaboration between technical and domain experts, leading to an evolving model that better serves business needs over time.

> **Ubiquitous Language**

- A common, rigorous language shared between developers and domain experts
- Used consistently in code, documentation, and spoken communication
- Reduces translation between technical and business terminology
- Evolves as the team's understanding of the domain grows

> **Strategic Design**

1. Bounded Contexts

- Clear boundaries between different parts of the model
- Each context has its own ubiquitous language
- Helps manage complexity in large systems

2. Context Mapping

- Defines relationships between bounded contexts
- Includes patterns like Partnership, Customer-Supplier, Conformist, etc.
- Helps teams understand system integration points

Tactical Design Patterns

1. Entities

- Objects with a distinct identity that runs through time
- Identity remains constant even as attributes change
- Example: A customer account with a unique ID

2. Value Objects

- Objects defined by their attributes rather than identity
- Immutable and interchangeable if attributes match
- Example: Money, addresses, or dates

3. Aggregates

- Cluster of domain objects treated as a single unit
- Has a root entity that controls access
- Maintains consistency boundaries
- Example: An order and its line items

4. Domain Events

- Record of something significant that happened in the domain
- Used to trigger reactions in other parts of the system
- Enable loose coupling between bounded contexts

5. Repositories

- Provide collection-like interface for accessing domain objects
- Abstract away data persistence details
- Focus on domain language rather than database concepts

6. Domain Services

- Operations that don't naturally belong to any entity or value object
- Represent domain concepts that are processes rather than things
- Stateless operations involving multiple domain objects

```
public class Group : AuditableAggregateRoot
{
    public Group(string? id)
    {
        Id = id ?? Ulid.NewUlid().ToString();
    }
    public required string Name { get; set; }
    private readonly List<GroupMember> _groupMembers = new ();
    public IReadOnlyCollection<GroupMember> GroupMembers =>
_groupMembers.AsReadOnly();
    private List<TestGroup> _testGroups = new ();
    public IReadOnlyCollection<TestGroup> TestGroups  =>
_testGroups.AsReadOnly();
    public static Group Create(string name, string? id = null)
    {
        var group = new Group(id)
        {
            Name = name
        };

        return group;
    }

    public void SyncTests(List<string> testIds)
    {
        _testGroups = testIds.Select(i => new TestGroup()
        {
            TestId = i,
            GroupId = this.Id
        }).ToList();
    }

    public void AddNewGroupMember(string userEmail, string groupId,
GroupRole groupRole, string? alias = null)
    {
        _groupMembers.Add(new GroupMember()
        {
            GroupRole = groupRole,
            GroupUserEmail = userEmail,
            GroupId = groupId,
            Alias = alias
        });
    }
}
```

3.2.11 Error Handling

ErrorOr

- Functional error handling
- Type-safe error responses
- Error aggregation
- Integration with FastEndpoints

```
public class LoginUserCommandHandler : ICommandHandler<LoginUserCommand,
ErrorOr<LoginUserResponse>>
{
    private readonly IDeceneaDbContext _dbContext;
    private readonly IConfiguration _configuration;
    public LoginUserCommandHandler(IDeceneaDbContext dbContext,
IConfiguration configuration)
    {
        _dbContext = dbContext;
        _configuration = configuration;
    }
    public async Task<ErrorOr<LoginUserResponse>>
ExecuteAsync(LoginUserCommand command, CancellationToken
cancellationToken)
    {
        try
        {
            var loginUserResponse = new LoginUserResponse();

            var user = await
_dbContext.Set<Domain.Aggregates.UserAggregate.User>()
                .FirstOrDefaultAsync(i=> i.Email == command.Email,
cancellationToken);

            if(user is null)
                return Error.NotFound(description: "User not found.");

            if(user.LockoutEnabled)
                return Error.Failure(description: "User is locked.");

            _dbContext.ModifiedBy = user.Id;

            var passCheck = CheckPassword(command.Password,
user.PasswordHash);

            if (passCheck.IsError)
            {
                return passCheck.Errors;
            }

            var accessTokenExpiryTime = DateTime.UtcNow.AddDays(1);
            var jwtToken = JWTBearer.CreateToken(
```

```
                signingKey: _configuration["Auth:JWTSigningKey"],
                expireAt: accessTokenExpiryTime,
                privileges: u =>
                {
                    u.Roles.Add(user.Role.ToString());

                    u.Permissions.AddRange(new[] { "Browse" });

                    u.Claims.Add(new("userName", user.UserName));
                    u.Claims.Add(new(ClaimTypes.Email, user.Email));

                    u["userId"] = user.Id; //indexer based claim setting
                });

            if (command.RememberMe)
            {
                var refreshToken = AuthTokenHelper.GenerateRefreshToken();
                loginUserResponse.AuthTokensResponse.RefreshToken =
refreshToken.RefreshToken;

loginUserResponse.AuthTokensResponse.RefreshTokenExpiryTime =
refreshToken.RefreshTokenExpiryTime;
                user.RefreshToken = refreshToken.RefreshToken;
                user.RefreshTokenExpiryTime =
refreshToken.RefreshTokenExpiryTime;
                _dbContext.Set<Domain.Aggregates.UserAggregate.User>
().Update(user);
            }

            loginUserResponse.AuthTokensResponse.AccessToken = jwtToken;
            loginUserResponse.AuthTokensResponse.AccessTokenExpiryTime =
accessTokenExpiryTime;

            var result = await
_dbContext.SaveChangesAsync(cancellationToken);

            if (result.IsError)
                return result.Errors;

            return loginUserResponse;
        }
        catch (Exception ex)
        {
            Log.Error("Didn't manage to login user: {email} :
{ex}",command.Email,ex);
            return Error.Unexpected(description: $"Didn't manage to login
user: {command.Email}");
        }
    }

    private ErrorOr<bool> CheckPassword(string password,
        string passwordHash)
    {
        var passwordHelper = new
```

```
PasswordHelper(Convert.FromBase64String(_configuration["Auth:Pepper"]));

        if (!passwordHelper.VerifyPassword(password, passwordHash))
        {
            return Error.Failure(description: "Credentials don't match.");
        }

        return true;
    }
}
```

3.3 Advanced Architectural Patterns

3.3.1 Outbox Pattern

The Outbox pattern is a design pattern used in distributed systems to ensure reliable message publishing in distributed transactions.

The core idea is to store messages/events in a local "outbox" table within the same database transaction as your business operations. This guarantees that both the business operation and message publishing are atomic - they either both succeed or both fail together.

Here's the flow:

- Application performs business logic and database updates
- As part of the same transaction, it stores outgoing messages in an outbox table
- A separate background process (often called a "relay" or "publisher") periodically polls the outbox table
- The relay publishes any unpublished messages to the message broker (like Kafka or RabbitMQ)
- After successful publishing, the relay marks messages as published or removes them from the outbox

Key benefits:

- Ensures messages are never lost if the application crashes after committing the transaction but before publishing
- Maintains data consistency between your database and published events
- Prevents duplicate message publishing through idempotency checks
- Preserves message ordering within each aggregate/entity

```
public sealed class OutboxMessage : Entity
{
    public string CreatedBy { get; init; }

    public DateTime OccurredOnUtc { get; init; }

    public string Type { get; init; }

    public string DomainEvent { get; init; }

    public DateTime? ProcessedOnUtc { get; set; }

    public string? Error { get; set; }
}
```

```
[DisallowConcurrentExecution]
internal sealed class ProcessOutboxMessagesJob : IJob
{
    private readonly IDeceneaDbContext _dbContext;

    public ProcessOutboxMessagesJob(IDeceneaDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task Execute(IJobExecutionContext context)
    {
        Log.Information("Beginning to process outbox messages");

        var outboxMessages = await _dbContext
            .Set<OutboxMessage>()
            .Where(i => i.ProcessedOnUtc == null)
            .ToListAsync();

        await using var transaction = await _dbContext
            .Database
            .BeginTransactionAsync();

        foreach (var outboxMessageGroup in outboxMessages.GroupBy(i =>
i.CreatedBy))
        {
            var error = string.Empty;
            await transaction.CreateSavepointAsync("OuterLoop");
            try
            {
                foreach (var outboxMessage in outboxMessageGroup)
                {
                    var domainEvent =
JsonSerializer.Deserialize<IDomainEvent>(outboxMessage.DomainEvent)!;
                    await domainEvent.PublishAsync(Mode.WaitForAll,
context.CancellationToken);
                }
            }
            catch (Exception ex)
            {
                await transaction.RollbackToSavepointAsync("OuterLoop");
                Log.Error(
                    ex,
                    "Exception while batch processing outbox messages with
trace Id {traceId}",
                    outboxMessageGroup.Key);
                error = ex.Message;
            }

            foreach (var outboxMessage in outboxMessageGroup)
            {
                outboxMessage.ProcessedOnUtc = DateTime.UtcNow;
```

```
                outboxMessage.Error = error;
            }
        }

        _dbContext.Set<OutboxMessage>().UpdateRange(outboxMessages);
        await transaction.CommitAsync();
        Log.Information("Completed processing outbox messages");
    }
}
```

3.3.2 Options Pattern

The Options pattern is a software design pattern commonly used in modern application frameworks for configuration and settings management.

The core concept is to use strongly-typed classes to represent groups of related settings. Instead of handling configuration values as loose strings or primitives, you encapsulate them in dedicated classes where:

- Settings are represented as properties with proper types and validation
- Related settings are grouped logically in the same class
- Options can be validated and processed during application startup
- Dependency injection is used to make settings available throughout the app

```
public sealed class OutboxOptions
{
    public int IntervalInSeconds { get; init; }

    public int BatchSize { get; init; }
}


// Registration
services.Configure<OutboxOptions>(configuration.GetSection("Outbox"));

// Usage
public class ProcessOutboxMessagesJobSetup :
IConfigureOptions<QuartzOptions>
{
    private readonly OutboxOptions _outboxOptions;

    public ProcessOutboxMessagesJobSetup(IOptions<OutboxOptions>
outboxOptions)
    {
        _outboxOptions = outboxOptions.Value;
    }

    public void Configure(QuartzOptions options)
    {
        const string jobName = nameof(ProcessOutboxMessagesJob);

        options
            .AddJob<ProcessOutboxMessagesJob>(configure =>
configure.WithIdentity(jobName))
            .AddTrigger(configure =>
                configure
                    .ForJob(jobName)
                    .WithSimpleSchedule(schedule =>

schedule.WithIntervalInSeconds(_outboxOptions.IntervalInSeconds).RepeatFor
ever()));
    }
}
```

3.3.3 Database Migrations on Startup

Automatic database schema updates using Entity Framework Core migrations:

```
#Database migration creation:
dotnet ef migrations add $(name) -s ./Decenea.WebAPI/Decenea.WebAPI.csproj
-p ./Decenea.Infrastructure/Decenea.Infrastructure.csproj
#Latest database migration removal:
dotnet ef migrations remove -s ./Decenea.WebAPI/Decenea.WebAPI.csproj -p
./Decenea.Infrastructure/Decenea.Infrastructure.csproj
#Database migration application:
dotnet ef database update -s ./Decenea.WebAPI/Decenea.WebAPI.csproj -p
./Decenea.Infrastructure/Decenea.Infrastructure.csproj
```

```
public static async Task ExecuteInfrastructureOnStartup(this
WebApplication app)
{
    using var scope = app.Services.CreateScope();
    var db = scope.ServiceProvider.GetRequiredService<DeceneaDbContext>();
    if ((await db.Database.GetPendingMigrationsAsync()).Any())
    {
        await db.Database.MigrateAsync();
    }
}

// Program.cs
var app = builder.Build();
/* Code in between */
await app.ExecuteInfrastructureOnStartup();
```

3.3.4 Integration Points

The complete system integration showing all patterns:

```
// Program.cs
var builder = WebApplication.CreateBuilder();
/* Code in between */
builder.Services
    .AddFastEndpoints()
    .AddInfrastructure(builder.Configuration)
    .AddAuthorization()
    .AddApplication();
```

3.3.5 ID Generation with ULID

The application uses ULIDs (Universally Unique Lexicographically Sortable Identifiers) for entity identification. ULIDs offer several advantages over traditional GUIDs or auto-incrementing integers:

- **Time-ordered**: First 48 bits contain a timestamp
- **Sortable**: Lexicographically sortable by timestamp
- **URL-safe**: Uses Crockford's base32 encoding
- **Unique**: 80 bits of randomness (more than UUIDv4)
- **Compact**: 26 characters vs 36 for UUID

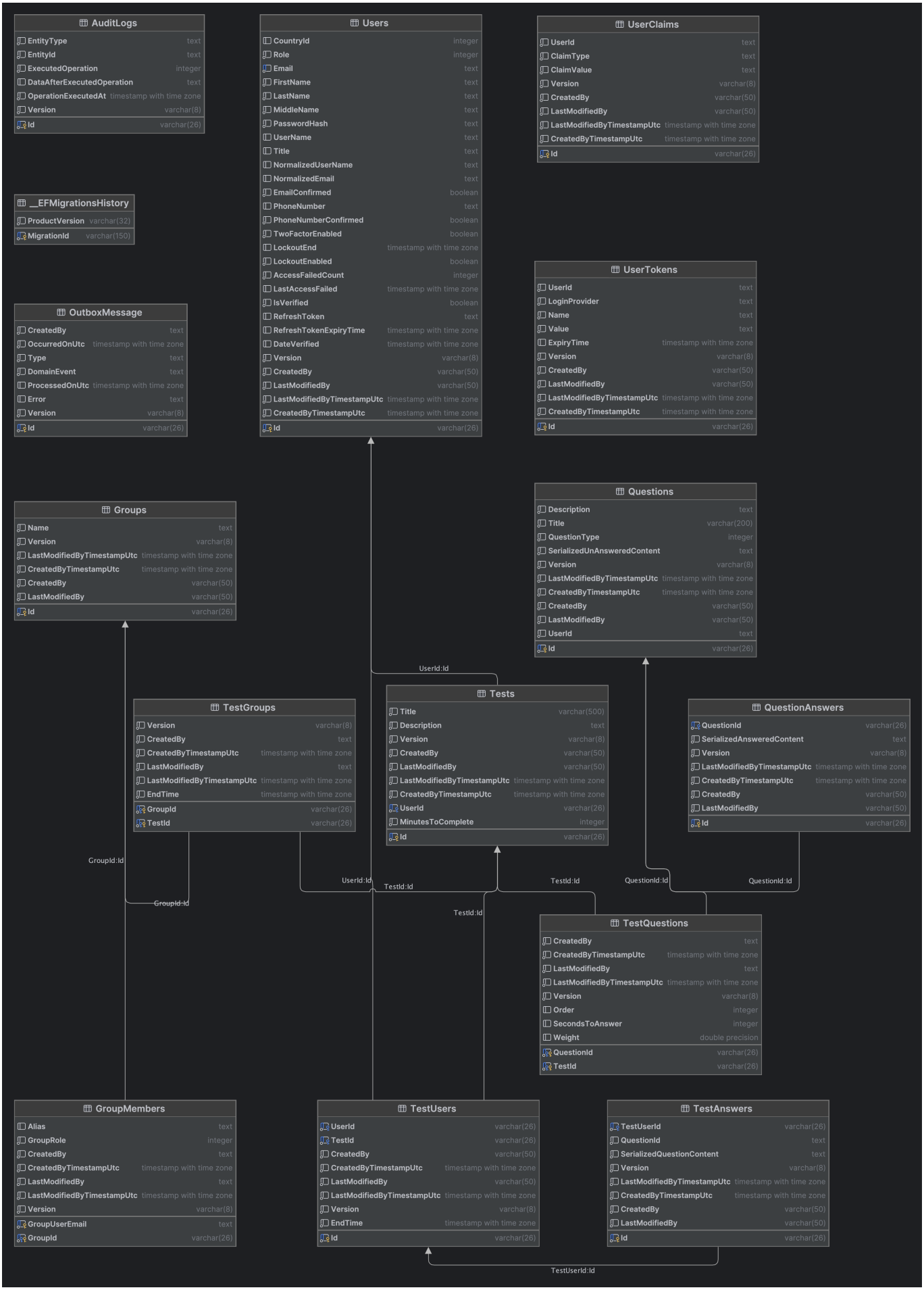
3.3.6 Entity Versioning, Concurrency and Version Control

The application implements optimistic concurrency control using a Version column for all entities. This prevents lost updates in concurrent scenarios.

The application implements automatic audit trails for all entities that inherit from AuditableEntity. This provides a complete history of who modified what and when.

- Automatic tracking of all changes
- Record of who made each change
- Timestamps for creation and modification
- Serialized snapshots of entity state
- Support for compliance and auditing requirements

3.4 Database Schema



AuditLogs

- **EntityType**: text
- **EntityId**: text
- **ExecutedOperation**: text
- **DateExecutedUtc**: timestamp with time zone
- **OperationExecutedAt**: timestamp with time zone
- **Version**: varchar(26)

EFMigrationHistory

- **ProductVersion**: varchar(32)
- **MigrationId**: varchar(150)

OutboxMessage

- **Id**: varchar(26)
- **CreatedUtc**: timestamp with time zone
- **OccurredUtc**: timestamp with time zone
- **TypeName**: text
- **Payload**: text
- **ProcessedUtc**: timestamp with time zone
- **Error**: text
- **Version**: varchar(26)

Users

- **Id**: varchar(26)
- **CountryId**: integer
- **Email**: text
- **FirstName**: text
- **LastName**: text
- **PhoneNumber**: text
- **PasswordHash**: text
- **UserName**: text
- **NormalizeUserName**: text
- **EmailConfirmed**: boolean
- **PhoneNumberConfirmed**: boolean
- **TwoFactorEnabled**: boolean
- **LockoutEnabled**: boolean
- **AccessFailedCount**: integer
- **LastAccessFailed**: timestamp with time zone
- **IsVerified**: boolean
- **RefreshToken**: text
- **RefreshTokenExpiryTime**: timestamp with time zone

- **DateVerified**: timestamp with time zone
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

UserClaims

- **Id**: varchar(26)
- **UserId**: text
- **ClaimType**: text
- **ClaimValue**: text
- **Version**: varchar(50)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

UserTokens

- **Id**: varchar(26)
- **UserId**: text
- **LoginProvider**: text
- **Name**: text
- **Value**: text
- **ExpTime**: timestamp with time zone
- **Version**: varchar(50)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

Groups

- **Id**: varchar(26)
- **Name**: text
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

GroupMembers

- **Id**: varchar(26)
- **GroupId**: varchar(26)
- **UserId**: text
- **Alias**: text
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

Questions

- **Id**: varchar(26)
- **Title**: text
- **Description**: text
- **QuestionType**: text
- **SerializedAnsweredContent**: text
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

QuestionAnswers

- **Id**: varchar(26)
- **QuestionId**: varchar(26)
- **SerializedAnsweredContent**: text
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

Tests

- **Id**: varchar(26)
- **Title**: text
- **Description**: text
- **MinutesToComplete**: integer
- **UserId**: varchar(26)
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone

- **LastModifiedTimestampUtc**: timestamp with time zone

TestGroups

- **Id**: varchar(26)
- **GroupId**: varchar(26)
- **TestId**: varchar(26)
- **EndTime**: timestamp with time zone
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

TestQuestions

- **Id**: varchar(26)
- **TestId**: varchar(26)
- **QuestionId**: varchar(26)
- **Order**: integer
- **SecondsToAnswer**: integer
- **Weight**: double precision
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

TestUsers

- **Id**: varchar(26)
- **TestId**: varchar(26)
- **UserId**: varchar(26)
- **EndTime**: timestamp with time zone
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

TestAnswers

- **Id**: varchar(26)
- **TestUserId**: varchar(26)

- **QuestionId**: varchar(26)
- **SerializedQuestionContent**: text
- **SerializedAnswerContent**: text
- **Version**: varchar(26)
- **CreatedBy**: varchar(50)
- **LastModifiedBy**: varchar(50)
- **CreatedTimestampUtc**: timestamp with time zone
- **LastModifiedTimestampUtc**: timestamp with time zone

Chapter 4: Functionality

4.1 Web API Functionality

4.1.1 Response Format

The standardized API response format ensures consistent error handling and data return across all endpoints. The isError boolean provides immediate status indication, while the messages dictionary allows for multiple contextual messages keyed by their source or type. This format supports both successful data returns and detailed error reporting, making it easier for clients to handle responses consistently. The format also facilitates internationalization as messages can be keyed to support multiple languages.

```
{
    "isError": boolean,
    "messages": {
        "key": "message"
    },
    "data": any
}
```

4.1.2 Authentication

Login

PUT `/api/auth/login`

- Authenticates users with email/password
- Returns JWT access token
- Optional refresh token with "Remember Me"
- Body:

```
{
  "email": "string",
  "password": "string",
  "rememberMe": boolean
}
```

Refresh Token

PUT `/api/auth/regenerate-auth-tokens`

- Refreshes expired access tokens
- Requires valid refresh token

4.1.3 User Management

Register User

POST `/api/users/register`

- Creates new user account
- Required: email, username, first/last name, password
- Optional: middle name, phone number

Update User

POST `/api/users/update`

- Updates user profile
- Requires authentication
- Roles: SuperAdmin, Admin, Member

List Users

GET `/api/users/get-many`

- Lists all users
- Admin role required
- Returns paginated results

4.1.4 Group Management

Create Group

POST `/api/groups/create`

- Creates new groups
- Authenticated endpoint

Update Group

PUT `/api/groups/update`

- Modifies group properties
- Requires version number

Delete Group

DELETE `/api/groups/delete`

- Removes specified groups
- Body includes group IDs

Get Group

GET `/api/groups/get`

- Retrieves single group
- Includes:
  - Members
  - Associated tests
  - Group metadata

List Groups

GET `/api/groups/get-many`

- Lists accessible groups
- Authenticated endpoint

Member Management

- **PUT** `/api/groups/add-group-members`: Add users
- **PUT** `/api/groups/remove-group-members`: Remove users
- **PUT** `/api/groups/update-group-member`: Update roles/aliases
- **PUT** `/api/groups/sync-tests`: Sync group tests

4.1.5 Test Management

Create Test

POST `/api/tests/create`

- Creates new test
- Defines:
  - Title
  - Description
  - Time limit
  - Questions (optional)

Update Test

PUT `/api/tests/update`

- Modifies test properties
- Requires version control
- Can update questions

Delete Test

DELETE `/api/tests/delete`

- Removes specified tests
- Batch deletion supported

Get Test

GET `/api/tests/get`

- Retrieves test details
- Query parameters:
  - `includeQuestions`
  - `includeGroups`
  - `includeUsers`

List Tests

GET `/api/tests/get-many`

- Lists available tests
- Supports:
  - Pagination
  - Detail level control

Active Test

GET `/api/tests/get-active`

- Fetches current test
- Includes answers

Question handling

- **PUT** `/api/tests/add-questions`: Add to test
- **PUT** `/api/tests/remove-questions`: Remove from test

4.1.6 Question Management

Create Questions

POST `/api/questions/create`

- Creates new questions
- Supports question types:
  - DragAndDrop
  - Dropdown
  - FillBlank
  - FillBlankDropdown
  - MultipleChoice
  - MultipleChoiceSingle
  - MultipleYesOrNo
  - OrderingDragAndDrop
  - Ordering

Delete Questions

GET `/api/questions/delete`

- Removes questions
- Query parameter: `questionIds`

Get Question

GET `/api/questions/get`

- Retrieves question details
- Query parameter: `questionId`

List Questions

GET `/api/questions/get-many`

- Lists all questions
- Authenticated endpoint
-

4.1.7 Answer Management

Submit Answers

POST `/api/answers/upsert`

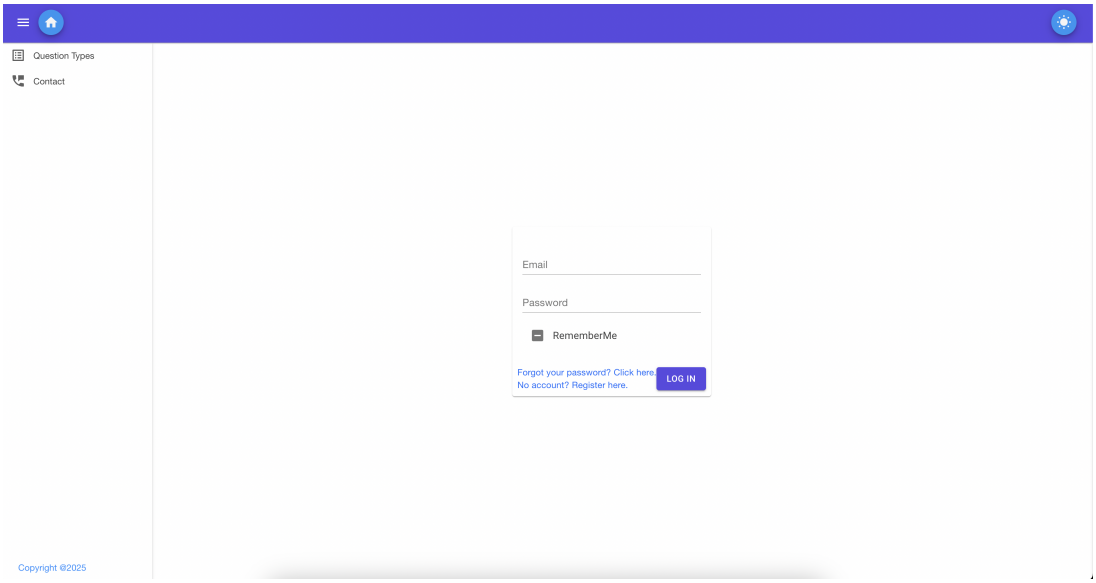
- Creates/updates test answers
- Stores serialized content
- Body:

```
{
  "testId": "string",
  "answers": [
    {
      "questionId": "string",
      "serializedQuestionContent": "string"
    }
  ]
}
```

4.2 Web Application Functionality

4.2.1 Login Page

A clean authentication interface where users can enter their email and password to access the system. The page appears to include "Remember me" functionality and likely includes validation for input fields. The design follows a modern, minimalist approach with clear error messaging and potentially password strength indicators.



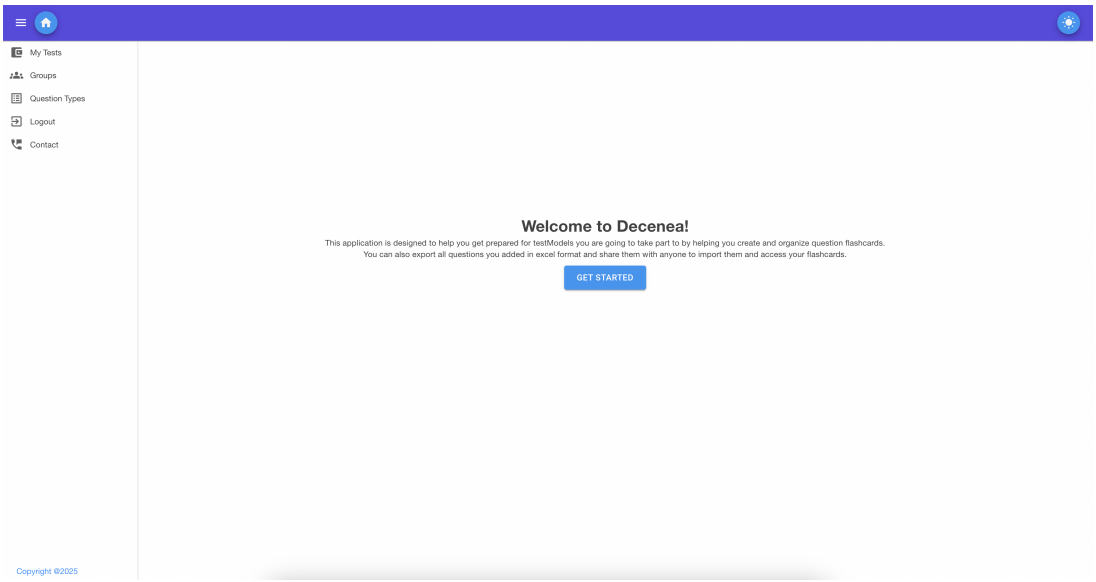
4.2.2 Register Page

A user registration form that collects essential information including email, first name, last name, middle name, username, and password. The form likely includes client-side validation for email format, password strength requirements, and username availability checking. It probably also includes terms of service acceptance.



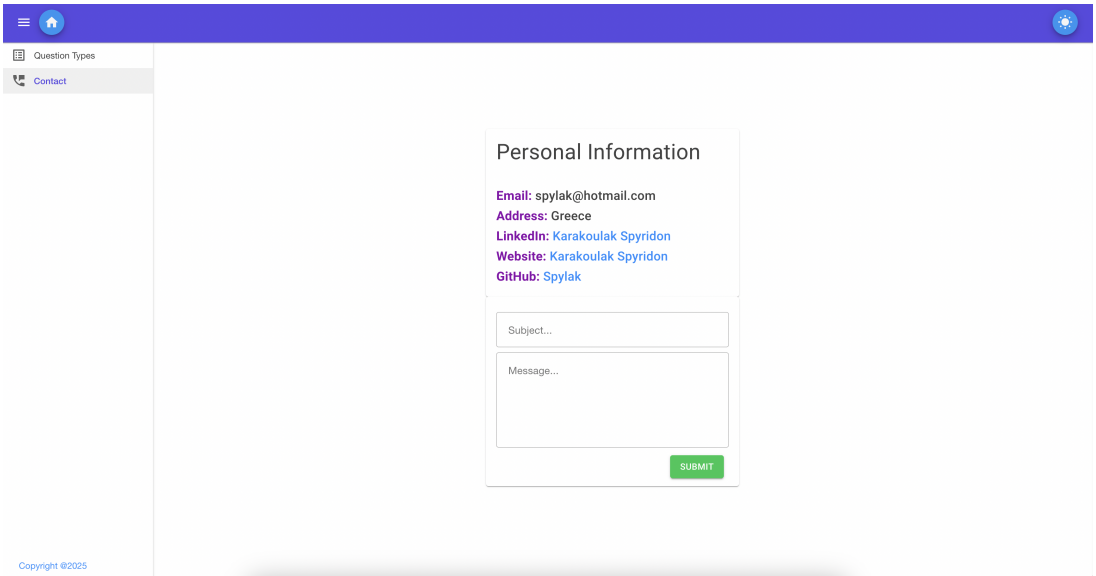
4.2.3 Home Page

The main landing page after authentication, featuring a navigation menu and possibly a dashboard overview of the user's tests, groups, and recent activities. The interface appears to use Material Design principles through MudBlazor components, with a responsive layout that adapts to different screen sizes.



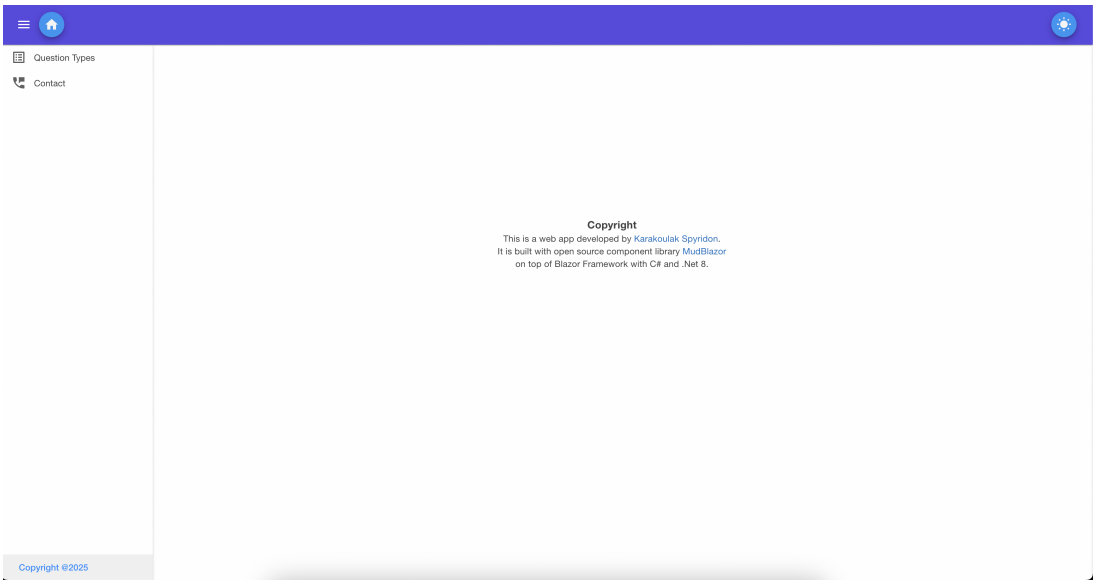
4.2.4 Contact Form

A communication interface that allows users to send messages or inquiries to system administrators. The form likely includes fields for subject, message content, and possibly attachment capabilities. It probably implements email validation and spam prevention measures.



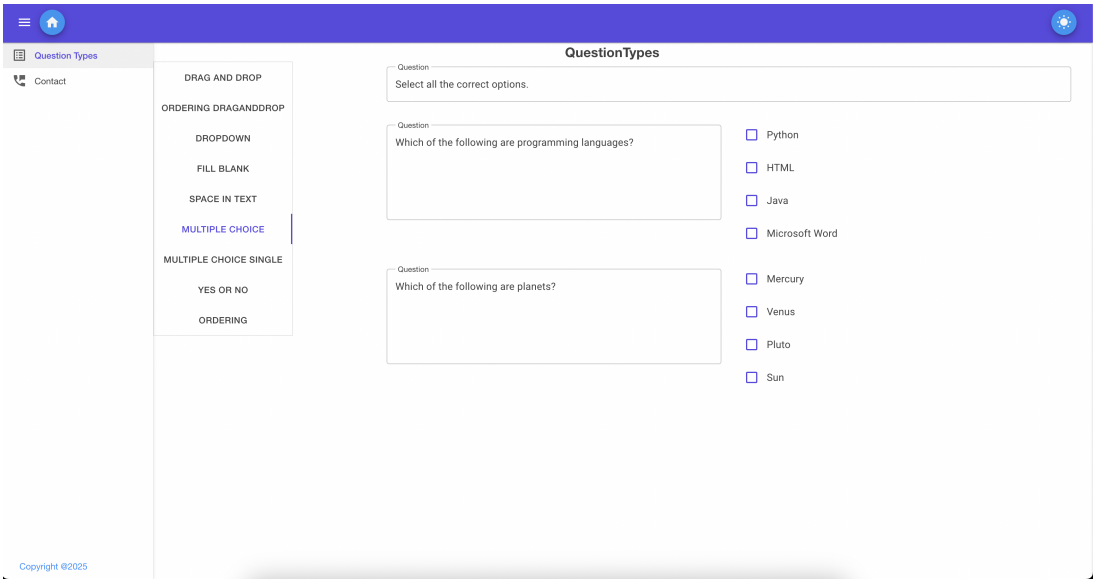
4.2.5 Copyright Page

An informational page displaying legal notices, copyright information, and potentially terms of service and privacy policy details. This page ensures legal compliance and transparently communicates usage rights and restrictions to users.



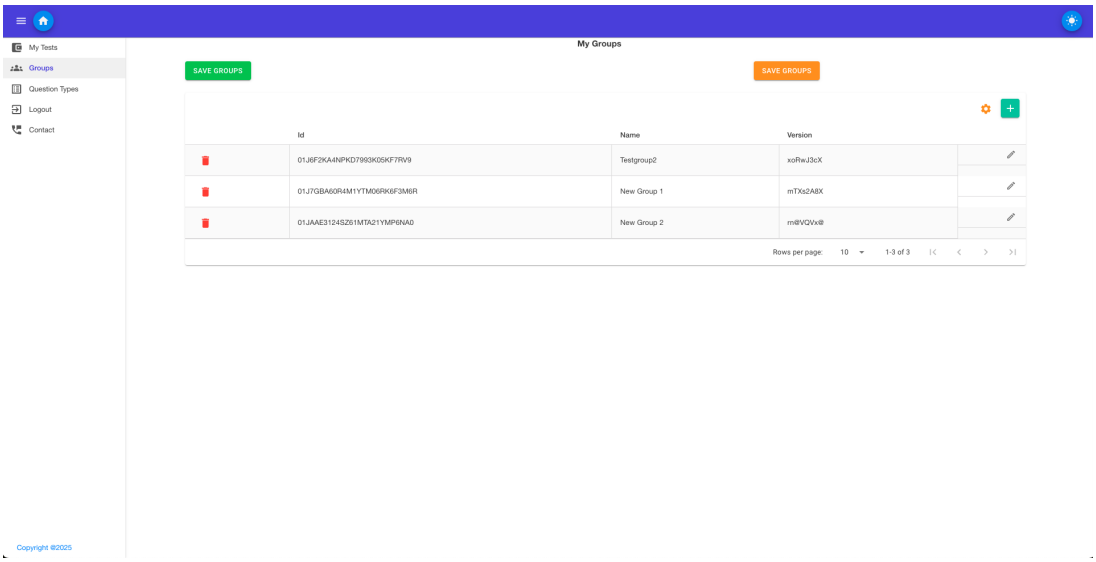
4.2.6 Question Types Page

A reference or selection interface showcasing the various types of questions available in the system, including Multiple Choice, Fill in the Blank, Drag and Drop, and others. This page likely serves as both documentation and a selection interface when creating new questions.



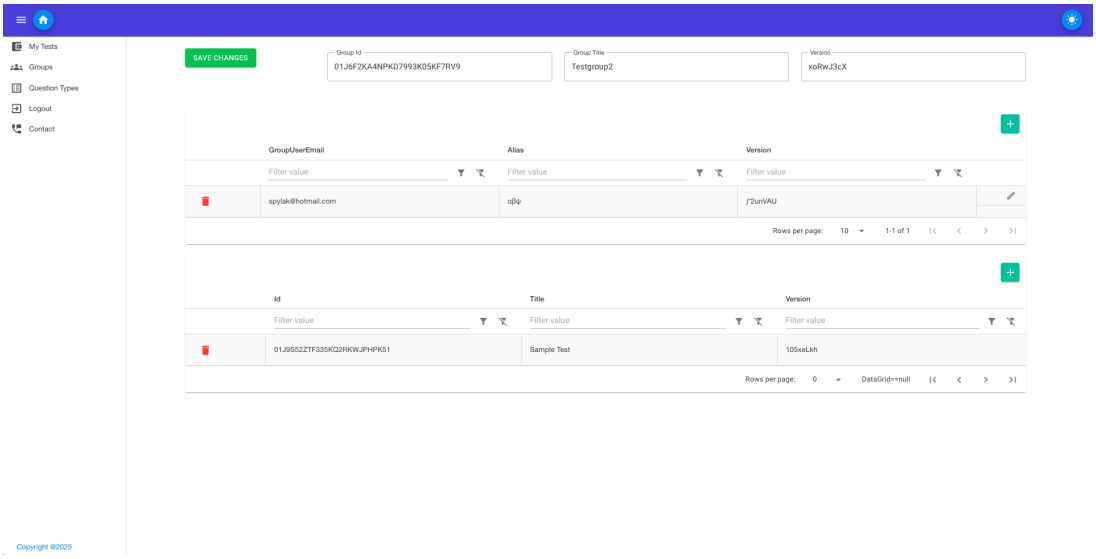
4.2.7 Groups Page

A management interface for viewing and organizing user groups. It likely includes functionality to create new groups, view existing ones, and manage group memberships. The interface probably supports sorting, filtering, and searching capabilities.



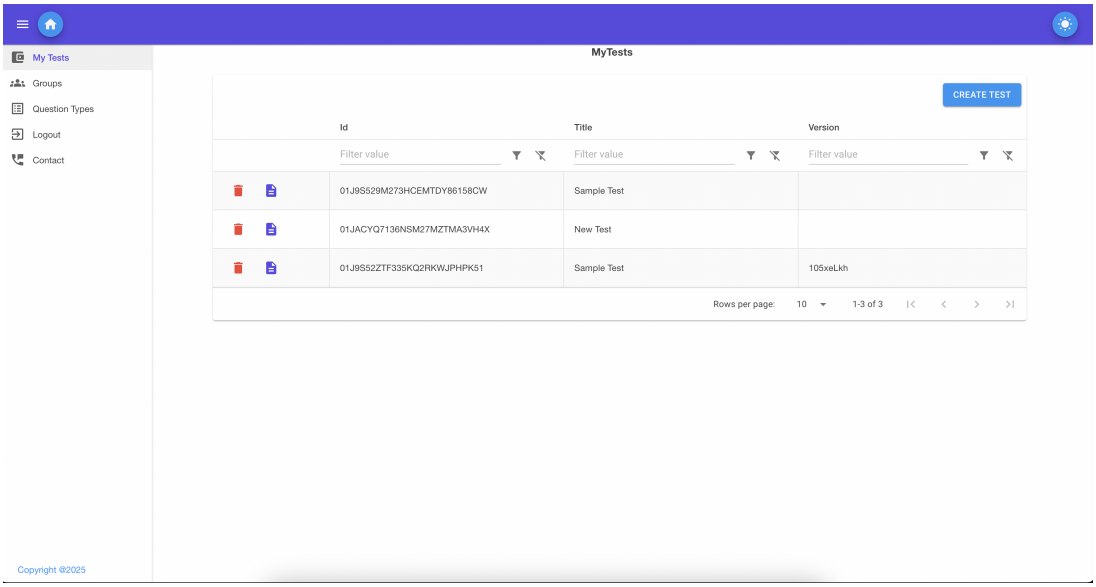
4.2.8 Edit Group Page

An interface for modifying group details, including adding or removing members, changing group settings, and managing group-specific permissions. It likely includes member role assignment and the ability to sync tests with the group.



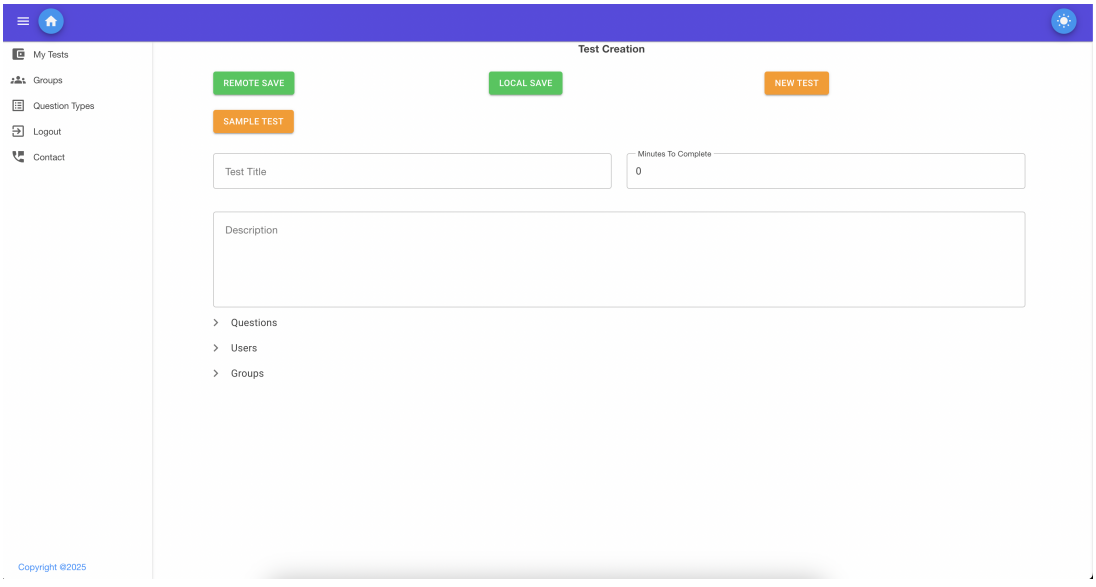
4.2.9 My Test Page

A personal dashboard showing tests assigned to or created by the user. This interface likely includes test status indicators, completion deadlines, and quick actions for managing individual tests.



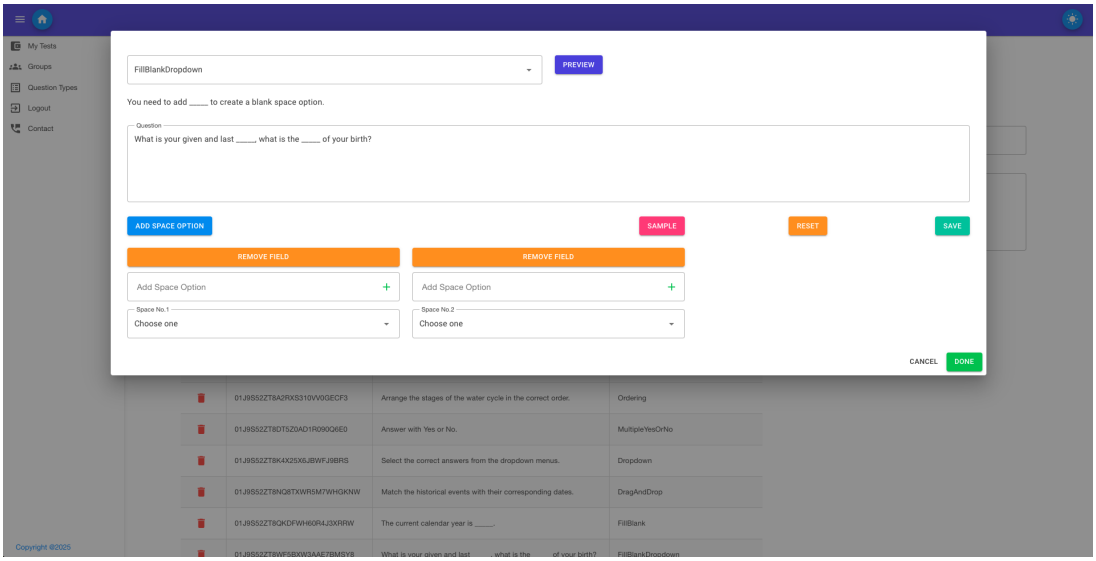
4.2.10 Create Test Page

A test creation interface where instructors can build new assessments. The page likely includes fields for test title, description, time limits, and the ability to add questions. It probably supports test scheduling and group assignment features.



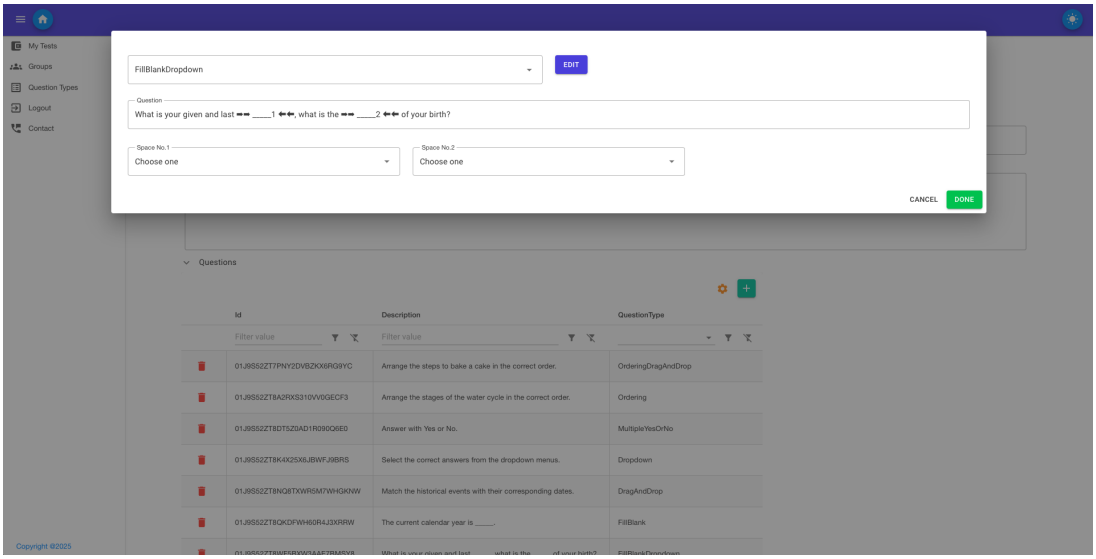
4.2.11 Add Question Edit

An interface for creating and editing test questions. The page appears to support multiple question types and includes options for setting correct answers, assigning points, and potentially adding media content.



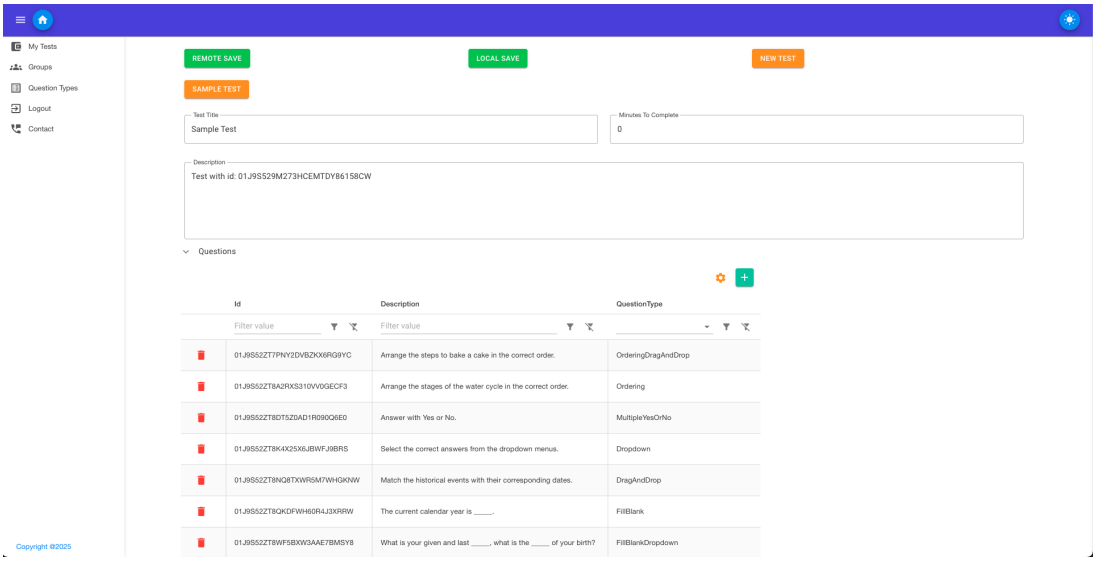
4.2.12 Add Question Preview

A preview mode showing how questions will appear to test takers. This allows question creators to verify the formatting, functionality, and clarity of their questions before publishing.



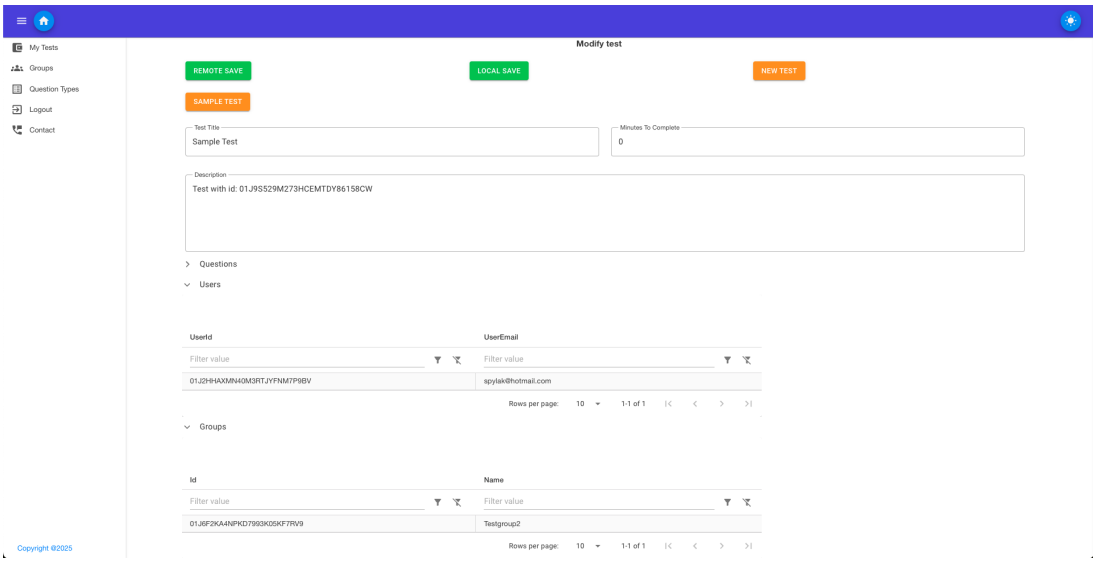
4.2.13 Update Test Question

An interface for modifying existing test questions, including changing content, answers, point values, and other question parameters. It likely includes version control to track changes.



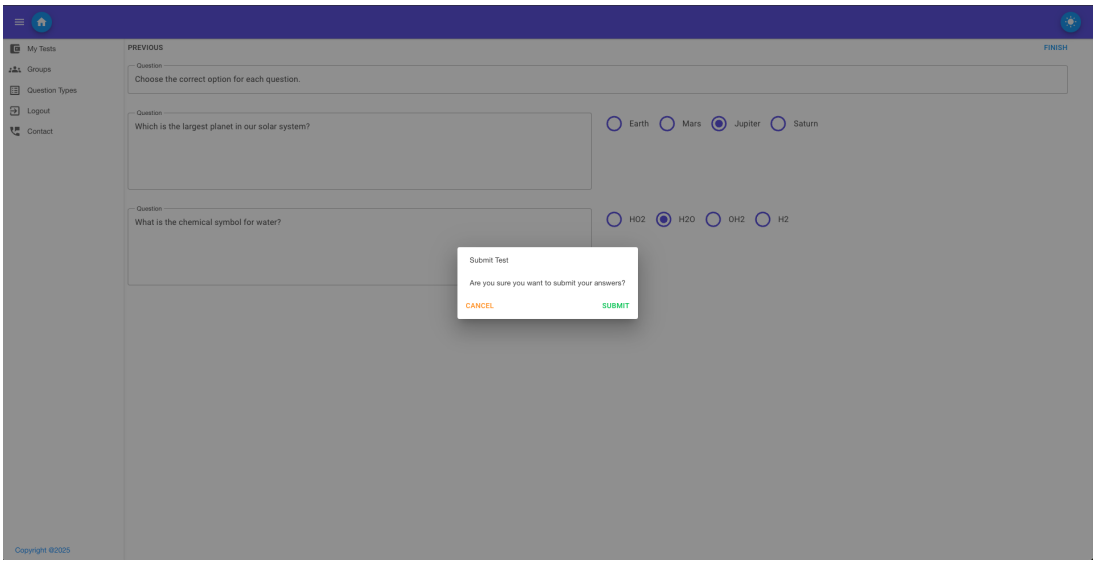
4.2.14 Update Test User Group

A management interface for modifying test assignments and group access. This page likely allows administrators to add or remove users/groups from tests and set specific test parameters for different groups.



4.2.15 Submit Test Page

The interface where students take tests, featuring a timer, question navigation, and answer submission capabilities. The page likely includes auto-save functionality and warns users about unanswered questions before final submission.



Chapter 5: Conclusion

5.1 Summary of Achievements

The development of this online examination and evaluation platform represents a successful implementation of modern software development practices and architectural patterns. The system successfully achieves its primary objectives:

1. **Robust Architecture**: The implementation of clean architecture principles has resulted in a highly maintainable and scalable system, with clear separation of concerns and dependencies flowing inward toward business rules.

2. **Secure Authentication**: The JWT-based authentication system, combined with refresh token functionality and role-based access control, provides a secure and user-friendly authentication experience.

3. **Flexible Question Management**: The platform supports multiple question types and formats, allowing for diverse assessment approaches while maintaining consistency in data handling and user experience.

4. **Reliable Data Handling**: Through the implementation of the Outbox pattern and optimistic concurrency control, the system ensures data consistency and reliable event processing in distributed scenarios.

5.2 Technical Innovations

Several technical innovations stand out in this implementation:

1. **Modern Frontend Development**: The use of Blazor WebAssembly with MudBlazor components demonstrates how C# can be effectively used for frontend development, providing a seamless development experience across the stack.

2. **Efficient API Design**: The adoption of FastEndpoints and the CQRS pattern has resulted in a highly performant and maintainable API structure, improving both development efficiency and runtime performance.

3. **Advanced Data Management**: The combination of PostgreSQL with Entity Framework Core, enhanced by features such as automatic audit trailing and entity versioning, provides a robust foundation for data persistence and tracking.

4. **Event Processing**: The implementation of the Outbox pattern ensures reliable event processing in distributed scenarios, solving common challenges in distributed systems.

5.3 Limitations and Future Work

While the platform successfully meets its core objectives, several areas present opportunities for future enhancement:

1. **Performance Optimization**:
    - Implementation of caching strategies for frequently accessed data
    - Further optimization of initial load time for Blazor WebAssembly
    - Addition of server-side prerendering capabilities
2. **Feature Enhancements**:
    - Integration of real-time collaboration features
    - Implementation of advanced analytics and reporting capabilities
    - Support for multimedia content in questions
    - Addition of AI-powered question generation and grading assistance
3. **Technical Improvements**:
    - Implementation of horizontal scaling capabilities
    - Addition of real-time notification systems
    - Enhancement of offline capabilities
    - Integration with learning management systems (LMS)

5.4 Lessons Learned

The development process revealed several valuable insights:

1. **Architecture Decisions**: The choice of clean architecture with domain-driven design proved beneficial for maintaining code quality and managing complexity, though it required initial investment in setup and training.
2. **Technology Selection**: The selection of .NET 8 and associated technologies provided a robust foundation, with the ecosystem's maturity supporting rapid development while maintaining code quality.
3. **Development Practices**: The importance of:
    - Comprehensive testing strategies
    - Proper error handling and logging
    - Documentation maintenance
    - Version control and change management
    - Security considerations at all levels

5.5 Final Remarks

This project demonstrates the successful application of modern software development practices in creating a comprehensive examination and evaluation platform. The implemented architecture provides a solid foundation for future enhancements while maintaining security, scalability, and maintainability.

The platform not only serves its immediate purpose of managing online examinations but also serves as a reference implementation for several modern software development patterns and practices. The lessons learned and architectural decisions documented here can inform future projects in similar domains.

The combination of robust backend services with a modern, responsive frontend creates a user-friendly experience while maintaining system integrity and security. As educational technology continues to evolve, this platform provides a flexible foundation that can adapt to changing requirements while maintaining its core reliability and functionality.

Sources

.NET and C# Documentation

- Microsoft .NET Documentation
  - Content on .NET 8 architecture, features, and components (Sections 2.1-2.2)
  - C# language specifications and programming concepts (Section 2.3)
  - ASP.NET Core documentation (Sections 2.6.1, 3.2.2)
  - Entity Framework Core documentation (Section 3.2.3)
- Microsoft Learn - .NET 8 Fundamentals
  - Core concepts and architecture of .NET 8 (Section 2.2)
  - Performance improvements and cross-platform capabilities
- Microsoft Learn - C# Programming Guide
  - Modern C# features and programming paradigms (Section 2.3)
  - Language evolution and best practices
- Microsoft Learn - Blazor WebAssembly
  - WebAssembly hosting model (Section 3.2.1)
  - Client-side execution and JavaScript interop
- Microsoft Learn - ASP.NET Core Middleware
  - HTTP pipeline configuration (Section 3.2.2)
  - Request processing architecture

Architecture and Design Patterns

- Clean Architecture (Robert C. Martin)
  - Core principles and layer organization (Section 3.1)
  - Dependency rules and boundaries
- Domain-Driven Design (Eric Evans)
  - Strategic design patterns (Section 3.2.10)
  - Tactical patterns and implementation
- Martin Fowler's Enterprise Patterns
  - CQRS pattern implementation (Section 3.2.2)
  - Unit of Work pattern

Technical Libraries and Frameworks

- FastEndpoints Documentation
    - API endpoint implementation (Section 3.2.2)
    - Request/response handling
- Blazor Documentation
    - WebAssembly architecture (Section 3.2.1)
    - Component lifecycle
    - State management
- MudBlazor Documentation
    - Component library usage
    - Material Design implementation

Database and Infrastructure

- PostgreSQL Documentation
    - Database schema design (Section 3.4)
    - Performance optimization
- Entity Framework Core Documentation
    - Data modeling
    - Migration management
    - Change tracking
- Microsoft Learn - Entity Framework Core Advanced Concepts
    - Entity configuration and relationships (Section 3.2.3)
    - Shadow properties and owned entities
    - Concurrency control implementation
- Microsoft Learn - Database Migration Strategies
    - Migration workflows and best practices (Section 3.3.3)
    - Deployment scenarios and automation

Security and Authentication

- JWT Authentication Specifications
    - Token structure and validation (Section 3.2.9)
    - Security best practices
- OAuth 2.0 and OpenID Connect Standards
    - Authentication flows
    - Authorization implementation
- Microsoft Learn - ASP.NET Core Security
    - Authentication and authorization frameworks (Section 3.2.9)
    - Data protection APIs and implementation
    - HTTPS enforcement and configuration
- Microsoft Learn - Identity Security Concepts
    - User management best practices (Section 4.1.3)
    - Password policies and storage
    - Multi-factor authentication implementation

API Design

- REST API Design Best Practices
    - Response format standardization (Section 4.1.1)
    - Error handling patterns
- OpenAPI Specification
    - API documentation (Section 3.2.6)
    - Schema definition
- Microsoft Learn - API Implementation Patterns
    - Controller vs. Minimal API approaches (Section 3.2.2)
    - Response formatting and content negotiation
    - API versioning strategies
- Microsoft Learn - Advanced Web API Concepts
    - Custom formatters and filters (Section 4.1)
    - Performance considerations and caching
    - API analyzers and conventions

Additional Modern Development Resources

- Microsoft Learn - Async Programming Patterns
    - Task-based asynchronous pattern (Section 2.4.1)
    - Best practices for async/await
    - Parallel processing capabilities
- Microsoft Learn - .NET Application Architecture
    - Microservices design patterns (Section 2.6.4)
    - Containerization with Docker and .NET
    - Cloud-native application principles
- Microsoft Learn - LINQ Programming
    - Query syntax and extension methods (Section 2.4.2)
    - Performance considerations
    - Advanced LINQ techniques

Source code