

**MASTER OF COMPUTER SCIENCE, NJALA UNIVERSITY SEMESTER  
1 PROJECT: CSP610- OOP AND CSP611-ADVANCE DATA  
STRUCTURES & ALGORITHMS IMPLEMENTATION**

**TUTOR: A.J. FOFANAH**

Group D

# **Electronic Student Registration System**

Technical Documentation

Group D Members

Joseph Prince Conteh 54992

Babah Saccoh 17080

Umarr Kamara

8-29-2024

Link to the source code on github

[https://github.com/Zenofume/GroupD\\_StudentRegistration\\_System](https://github.com/Zenofume/GroupD_StudentRegistration_System)

## **Project Proposal**

### **Title: Electronic Student Registration System**

#### **Abstract :** Electronic Student Registration System

The Electronic Student Registration System (ESRS) is an innovative and comprehensive solution designed to streamline the student registration process for universities. The system provides a digital platform where university administrators can efficiently manage student enrolment, academic records, and course registrations. By leveraging modern technologies, ESRS eliminates the need for manual paperwork, reduces administrative overhead, and minimizes the potential for errors.

University administrators can use the system to create student profiles, assign students to specific courses and programs, and manage enrolment data in real-time. The system also supports role-based access control, ensuring that only authorized personnel can perform specific tasks such as updating student information, approving registrations, and generating academic reports.

Additionally, the ESRS integrates with existing university systems, providing seamless data exchange and ensuring that student records are consistently up-to-date. The system's user-friendly interface and automated workflows enhance the overall efficiency of the registration process, allowing university staff to focus on more strategic initiatives.

Ultimately, the ESRS aims to enhance the student experience by providing a faster, more reliable registration process, while also improving the university's operational efficiency and data accuracy.

## **OBJECTIVES OF THE ELECTRONIC STUDENT REGISTRATION SYSTEM**

1. Streamline the Registration Process:
  - Automate the student registration process to reduce manual data entry and paperwork.

- Simplify course and program enrolment for students and administrators.
2. Enhance Data Accuracy:
- Minimize errors associated with manual registration by implementing automated data validation and real-time updates.
  - Ensure accurate and consistent student records across the university's systems.
3. Improve Administrative Efficiency:
- Reduce the administrative burden by providing tools for batch processing of registrations, course assignments, and report generation.
  - Enable quick and easy access to student information for authorized university staff.
4. Implement Role-Based Access Control:
- Ensure that only authorized personnel have access to specific functionalities within the system.
  - Protect sensitive student data by restricting access based on user roles.
5. Enhance Student Experience:
- Provide a user-friendly interface for students to manage their course enrollments, view schedules, and track academic progress.
  - Offer real-time feedback and notifications during the registration process.
6. Integrate with Existing University Systems:
- Seamlessly integrate with the university's existing systems, such as Learning Management Systems (LMS) and Student Information Systems (SIS), to ensure data consistency.
  - Facilitate data exchange and synchronization between the ESRS and other institutional databases.
7. Support Scalability and Flexibility:
- Design the system to accommodate the growing needs of the university, including increasing student enrolment and expanding course offerings.
  - Allow for customization to meet specific institutional requirements and policies.
8. Provide Comprehensive Reporting and Analytics:

- Generate detailed reports on student enrolment, academic performance, and course popularity to support data-driven decision-making.
- Offer analytics tools to identify trends and optimize the registration process.

## **Technologies Used in the Electronic Student Registration System**

### **1. Backend: ASP.NET Web API (C#)**

- **Programming Language:** C#
- **Framework:** ASP.NET Web API on .NET 8
  - Leveraging the latest version of the .NET framework to build high-performance, scalable, and secure RESTful APIs.
  - Facilitates the creation of RESTful services to handle HTTP requests (GET, POST, PUT, DELETE) and manage data exchanges between the front end and database.
- **Entity Framework Core:**
  - An ORM (Object-Relational Mapper) that simplifies database interactions by allowing developers to work with data using C# objects rather than SQL queries.
- **Dependency Injection:**
  - Ensures modular and testable code by injecting dependencies into classes rather than creating them inside the classes.
- **Middleware:**
  - Custom middleware components are used for tasks such as authentication, logging, and error handling.
- **JWT Authentication:**
  - Secure token-based authentication to protect API endpoints and ensure that only authorized users can access specific functionalities.

### **2. Frontend: Angular**

- **Programming Language:** TypeScript
- **Framework:** Angular (version 18)
  - A robust framework for building dynamic and responsive single-page applications (SPAs).

- **Angular CLI:**
  - A command-line interface tool that streamlines the development process, offering commands for code scaffolding, testing, and deployment.
- **Reactive Forms:**
  - Utilized for building complex and dynamic forms that handle user input with real-time validation and state management.
- **HttpClientModule:**
  - Manages HTTP communication with the backend API, handling tasks like sending requests, receiving responses, and managing headers and tokens.
- **RouterModule:**
  - Provides navigation and routing capabilities, allowing the creation of multi-page applications with smooth transitions between views.
- **ActivatedRoute & Router:**
  - Services from @angular/router used for navigating between routes and accessing route parameters dynamically.
- **SweetAlert2 (Swal):**
  - A library used for creating attractive and customizable pop-up alerts for confirmation dialogs, success messages, and error notifications.
- **XLSX:**
  - A library for handling Excel files, allowing users to export data to Excel format.
- **FileSaver.js (saveAs):**
  - A utility used to save files on the client side, enabling features like downloading files directly from the application.

### 3. Database: Microsoft SQL Server

- **Database Management System:** Microsoft SQL Server
  - A reliable and scalable RDBMS (Relational Database Management System) used to store and manage the university's student, course, and enrollment data.

- **SQL Server Management Studio (SSMS):**
  - A tool used to manage SQL Server instances, write and execute SQL queries, manage databases, and perform administrative tasks.
- **Stored Procedures (Optional):**
  - Precompiled SQL statements that can be executed repeatedly, offering performance benefits and improved security by encapsulating SQL logic within the database.
- **Database Migration Tools:**
  - Tools integrated with Entity Framework Core to handle schema changes and ensure database consistency across different environments.

#### 4. Version Control: GitHub

- **Platform: GitHub**
  - A version control platform used to manage code changes, collaborate with team members, and maintain a history of all project developments.
- **Git:**
  - A distributed version control system that tracks changes in source code, allowing developers to revert to previous versions, manage branches, and merge code efficiently.

#### Additional Features and Libraries

- **Swagger:**
  - Automatically generates API documentation, making it easier for developers to understand and test the API endpoints.
- **GitHub Actions (Optional):**
  - CI/CD tools for automating the build, test, and deployment processes, ensuring continuous integration and delivery of the application.

## SYSTEM DESIGN

### System Design Overview

#### 1. Class Diagrams

Here are 10 class diagrams that illustrate the OOP design of the application:

## 1. User Class Diagram

- Attributes:
  - UserId: int
  - Username: string
  - PasswordHash: string
  - Role: string
  - RefreshToken: string (nullable)
  - RefreshTokenExpiryTime: DateTime
  - IsPasswordResetRequired: bool
  - StudentId: int (nullable)
- Associations:
  - One-to-one relationship with Student.

## 2. Student Class Diagram

- Attributes:
  - StudentId: int
  - FirstName: string
  - LastName: string
  - DateOfBirth: DateTime
  - Gender: string
  - DepartmentId: int
  - CourseProgramId: int
- Associations:
  - Many-to-one relationship with Department.
  - Many-to-one relationship with CourseProgram.
  - One-to-one relationship with User.

## 3. School Class Diagram

- Attributes:
  - SchoolId: int

- SchoolName: string
- SchoolCode: string
- Associations:
  - One-to-many relationship with Department.

#### 4. Department Class Diagram

- Attributes:
  - DepartmentId: int
  - DepartmentName: string
  - SchoolId: int
- Associations:
  - Many-to-one relationship with School.
  - One-to-many relationship with Student.

#### 5. CourseProgram Class Diagram

- Attributes:
  - CourseProgramId: int
  - CourseProgramName: string
  - CourseLevel: string
  - Semester: string
  - AcademicYear: string
- Associations:
  - One-to-many relationship with Student.

#### 6. ChangePasswordDto Class Diagram

- Attributes:
  - Username: string
  - OldPassword: string
  - NewPassword: string

#### 7. CreateUserRequest Class Diagram

- Attributes:



- Username: string
- Password: string
- Role: string

#### 8. LoginDto Class Diagram

- Attributes:
  - Username: string
  - Password: string

#### 9. ResetPasswordRequest Class Diagram

- Attributes:
  - Username: string
  - NewPassword: string

#### 10. TokenApiDto Class Diagram

- Attributes:
  - AccessToken: string
  - RefreshToken: string

These diagrams reflect the core entities and DTOs in your application, along with their relationships, to ensure a robust and maintainable design (DTO stands for **Data Transfer Object**. It's a design pattern used to transfer data between different layers of an application, especially between the presentation layer).

### 1. Encapsulation in User Class:

- **Hiding Sensitive Data:** PasswordHash is used instead of storing plain text passwords. This ensures that the actual password is not exposed.
- **Controlled Access:** The RefreshToken and RefreshTokenExpiryTime fields can be accessed or modified only through the User class's methods or properties, not directly from outside the class.

This is an illustration of it in one of my classes

```

public class User
{
    [Key]
    public int UserId { get; set; } // Unique identifier for the user

    [Microsoft.Build.Framework.Required]
    public string Username { get; set; } // Username for login

    [Microsoft.Build.Framework.Required]
    public string PasswordHash { get; set; } // Store hashed passwords

    public string Role { get; set; } // Role of the user (Admin or Student)

    public string? RefreshToken { get; set; } // Token used for refreshing JWT
    public DateTime RefreshTokenExpiryTime { get; set; } // Expiry time of the refresh token

    public bool IsPasswordResetRequired { get; set; } = true; // Flag indicating if password reset is required
}

```

Encapsulation ensures that:

- Data is hidden from direct access and modification.
- Data integrity is maintained by providing controlled access.
- Changes to the internal representation do not affect outside code as long as the public interface remains consistent.

In the code snippet, encapsulation is achieved by using private fields, public properties, and methods to control access and modify data safely.

## 2. Inheritance

Inheritance enables one class (the subclass or derived class) to acquire properties and methods from another class (the superclass or base class). This allows subclasses to reuse code from their parent classes and extend or modify that functionality as needed.

In the Code:

```

public class User
{
    [Key]
    public int UserId { get; set; } // Unique identifier for the user

    [Microsoft.Build.Framework.Required]
    public string Username { get; set; } // Username for login

    [Microsoft.Build.Framework.Required]
    public string PasswordHash { get; set; } // Store hashed passwords

    public string Role { get; set; } // Role of the user (Admin or Student)

    public string? RefreshToken { get; set; } // Token used for refreshing JWT
    public DateTime RefreshTokenExpiryTime { get; set; } // Expiry time of the refresh token

    public bool IsPasswordResetRequired { get; set; } = true; // Flag indicating if password reset is required

    public int? StudentId { get; set; } // Foreign key for Student
    public Student Student { get; set; } // Navigation property to Student
}

public class Student : User
{
    [Key]
    public int StudentId { get; set; } // Unique identifier for the student

    [Required]
    public string FirstName { get; set; } // Student's first name
}

```

### Inheritance in Action:

- **Base Class (User):** Contains common properties and methods related to user management, such as Username, PasswordHash, and Role.
- **Derived Class (Student):** Inherits all properties and methods from User and adds its own specific properties like FirstName, LastName, and StudentId.

### Benefits of Inheritance

1. **Code Reuse:** The Student class can use the existing functionality from the User class without needing to redefine those properties and methods.
2. **Extensibility:** Student can extend or override functionality from User, allowing it to have additional properties and behaviors specific to students while maintaining the core user functionality.

### 3. Polymorphism

Polymorphism can be achieved in two main ways:

1. **Compile-time Polymorphism (Method Overloading):** The ability to define multiple methods with the same name but different parameters.
2. **Run-time Polymorphism (Method Overriding):** The ability to redefine a method in a derived class that has already been defined in its base class.

In the Code:

```
namespace StudentRegistrationSystem.Properties.Domain.Services
{
    public interface IUserService
    {
        Task<User> CreateStudentUserAsync(Student student);
        Task ResetPasswordAsync(User user);
        Task<User> AuthenticateAsync(string username, string password);
        Task<bool> ChangePasswordAsync(User user, string newPassword);
    }
}
```

#### Polymorphism with Interfaces:

- **Definition:** IUserService defines a set of methods related to user management.
- **Implementation:** Classes that implement IUserService (like UserService) must provide specific implementations for these methods.
- **Interface Implementation:** UserService implements the IUserService interface, fulfilling the contract defined by the interface.
- **Method Signatures:** UserService provides concrete implementations for the methods defined in IUserService.
- **Benefits of Polymorphism in This Context**
  - **Flexibility:** You can have multiple implementations of IUserService. For example, if you needed a different way of handling users in a different context, you could create another class that implements IUserService.
  - **Substitution:** You can substitute UserService with any other class that implements IUserService without changing the code that relies on IUserService.
  - **Decoupling:** The AuthController (or any other consumer of IUserService) does not need to know

```

namespace StudentRegistrationSystem.Properties.Domain.Services
{
    public class UserService : IUserService
    {
        private readonly StudentRegistrationDbContext _context;
        private readonly IPasswordHasher<User> _passwordHasher;

        public UserService(StudentRegistrationDbContext context, IPasswordHasher<User> passwordHasher)
        {
            _context = context;
            _passwordHasher = passwordHasher;
        }

        public async Task<User> CreateStudentUserAsync(Student student)
        {
            var user = new User
            {
                Username = student.StudentIDNumber,
                Role = "Student",
                StudentId = student.StudentId,
                PasswordHash = _passwordHasher.HashPassword(new User(), "default"),
                IsPasswordResetRequired = true
            };
        }
    }
}

```

#### 4. Abstraction

**Abstraction** is the concept of hiding complex implementation details and showing only the essential features of an object or system. It allows users to interact with an object or system through a simplified interface while hiding the underlying complexity.

In the provided code snippets, abstraction is demonstrated primarily through the use of interfaces and the way certain services are designed. Here's how abstraction is used:

##### 1. Interface Abstraction

**IUserService Interface:**

```

namespace StudentRegistrationSystem.Properties.Domain.Services
{
    public interface IUserService
    {
        Task<User> CreateStudentUserAsync(Student student);
        Task ResetPasswordAsync(User user);
        Task<User> AuthenticateAsync(string username, string password);
        Task<bool> ChangePasswordAsync(User user, string newPassword);
    }
}

```

### Abstraction in Action:

- **Interface Definition:** The IUserService interface defines methods for user-related operations without specifying how these operations are implemented.
- **Client Code:** Classes or controllers that use IUserService only interact with the interface, not the specific implementation. For example, the AuthController uses IUserService to perform user operations.

## 2. Concrete Implementation Hides Complexity

### UserService Class:

```
namespace StudentRegistrationSystem.Properties.Domain.Services
{
    public class UserService : IUserService
    {
        private readonly StudentRegistrationDbContext _context;
        private readonly IPasswordHasher<User> _passwordHasher;

        public UserService(StudentRegistrationDbContext context, IPasswordHasher<User> pas
        {
            _context = context;
            _passwordHasher = passwordHasher;
        }

        public async Task<User> CreateStudentUserAsync(Student student)
        {
            var user = new User
            {
                Username = student.StudentIDNumber,
                Role = "Student",
                StudentId = student.StudentId,
                PasswordHash = _passwordHasher.HashPassword(new User(), "default"),
            };
        }
    }
}
```

## Data Flow Diagrams (DFDs)

Data Flow Diagrams illustrate how data is processed within the application. Here's an outline for creating them:

- **Level 0 (Context Diagram):**
  - Represents the entire system as a single process.
  - **Entities:**
    - Users (Admins, Students)
    - External API (for authentication)
  - **Data Flows:**

- User credentials are sent to the system for authentication.
- The system processes the credentials and interacts with the external API for verification.
- The system returns access tokens and user data based on authentication results.
- **Level 1 (Detailed Processes):**
  - Break down the system into sub-processes:
    1. **User Authentication Process:**
      - Users send credentials to the system.
      - The system interacts with the external API for validation.
      - The API returns a token, which is stored and returned to the user.
    2. **Student Registration Process:**
      - Admin inputs student data.
      - The system processes and stores the student data in the database.
      - Generates a unique `StudentID` and associates it with the user.
    3. **Course Enrollment Process:**
      - Students select courses for enrollment.
      - The system validates course availability and prerequisites.
      - The system stores the enrollment data.
    4. **Password Management Process:**
      - Users submit requests to change or reset passwords.
      - The system validates and updates the password in the database.
- **Level 2 (Data Stores and Interactions):**
  - Focus on how data moves between different tables/entities:
    - **User Table:** Stores user credentials and roles.
    - **Student Table:** Stores student-specific data.
    - **CourseProgram Table:** Manages course offerings.
    - **Department Table:** Manages departments and their relationships with students and courses.

### 3. Algorithm Design

Here are some key algorithms that you might implement in the system:

1. **User Authentication Algorithm:**
  - **Purpose:** Authenticate users by validating their credentials with an external API and manage token generation.
  - **Steps:**
    1. Receive username and password from the user.
    2. Hash the password and compare it with the stored hash.
    3. If valid, request a token from the external API.
    4. Store the token and return it to the user.
2. **Password Hashing Algorithm:**
  - **Purpose:** Securely store passwords by hashing them before saving them to the database.
  - **Steps:**
    1. Accept a plain-text password.
    2. Generate a salt.
    3. Hash the password with the salt using a secure hashing algorithm (SHA-256).
    4. Store the hashed password and salt.

### 3. Course Enrollment Algorithm:

- **Purpose:** Allow students to enroll in courses while validating prerequisites and availability.
- **Steps:**
  1. Accept a list of courses for enrollment.
  2. Check course availability and prerequisites.
  3. If valid, update the student's course list and adjust course availability.

### 4. Token Refresh Algorithm:

- **Purpose:** Refresh an expired token using a refresh token without requiring the user to log in again.
- **Steps:**
  1. Accept the refresh token from the user.
  2. Validate the refresh token against the stored token.
  3. If valid, generate a new access token and update the refresh token.

### 5. Data Validation Algorithm:

- **Purpose:** Ensure that all input data (e.g., student details, course selections) conforms to predefined rules before storing it.
- **Steps:**
  1. Accept input data.
  2. Validate each field according to business rules (e.g., required fields, data formats).
  3. If validation passes, proceed with storage; otherwise, return validation errors.

These algorithms and designs are crucial for ensuring that your application is secure, efficient, and user-friendly.

## Algorithms and Their Purpose in the UserService Class

In the UserService class, various algorithms are implemented for managing user authentication, authorization, and security. Here's an outline of the key algorithms used and their purposes:

### 1. Password Hashing and Verification

**Purpose:** Securely store and validate user passwords.

- **Algorithm Used:** IPasswordHasher<TUser> from ASP.NET Core Identity.
- **How It Works:**
  - **Hashing Passwords:**

```
user.PasswordHash = _passwordHasher.HashPassword(user, "default");
```

- Passwords are hashed before storing them in the database to protect them from being exposed in plaintext.
- **Verifying Passwords:**

```
result = _passwordHasher.VerifyHashedPassword(user, user.PasswordHash, password);
```



When a user attempts to log in, the provided password is hashed and compared with the stored hash.

## 2. JWT Token Creation

**Purpose:** Create JSON Web Tokens (JWT) for authenticating users.

- **Algorithm Used:** HMAC-SHA256.
- **How It Works:**

```
var credentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256);
```

**Token Generation:**

```
var token = jwtTokenHandler.CreateToken(tokenDescriptor);
```

Generates a JWT with claims, expiration, and signing credentials.

- **Token Serialization:**

```
return jwtTokenHandler.WriteToken(token);
```

- - Converts the JWT into a string format for transmission.

## 3. JWT Token Validation

**Purpose:** Validate JWTs to ensure they are authentic and have not expired.

- **Algorithm Used:** HMAC-SHA256.

```
var tokenValidationParameters = new TokenValidationParameters
{
    IssuerSigningKey = new SymmetricSecurityKey(key),
    ValidateIssuerSigningKey = true,
    ValidateLifetime = false // For expired tokens
};
```

**Token Validation:**

```
var principal = tokenHandler.ValidateToken(token, tokenValidationParameters, out s
```

Validates the JWT against the specified parameters (e.g., signing key).

**Data Structures:**

## 1. Arrays

### Use Case: Pagination and Sorting

- Arrays can be used to handle pagination and sorting of student records.

#### Implementation:

```
[HttpGet("paged")]
0 references
public async Task<ActionResult<IEnumerable<Student>>> GetPagedStudents(int page = 1, int pageSize = 10)
{
    var students = await _context.Students
        .OrderBy(s => s.StudentId) // Or any other sorting criteria
        .Skip((page - 1) * pageSize)
        .Take(pageSize)
        .ToListAsync();

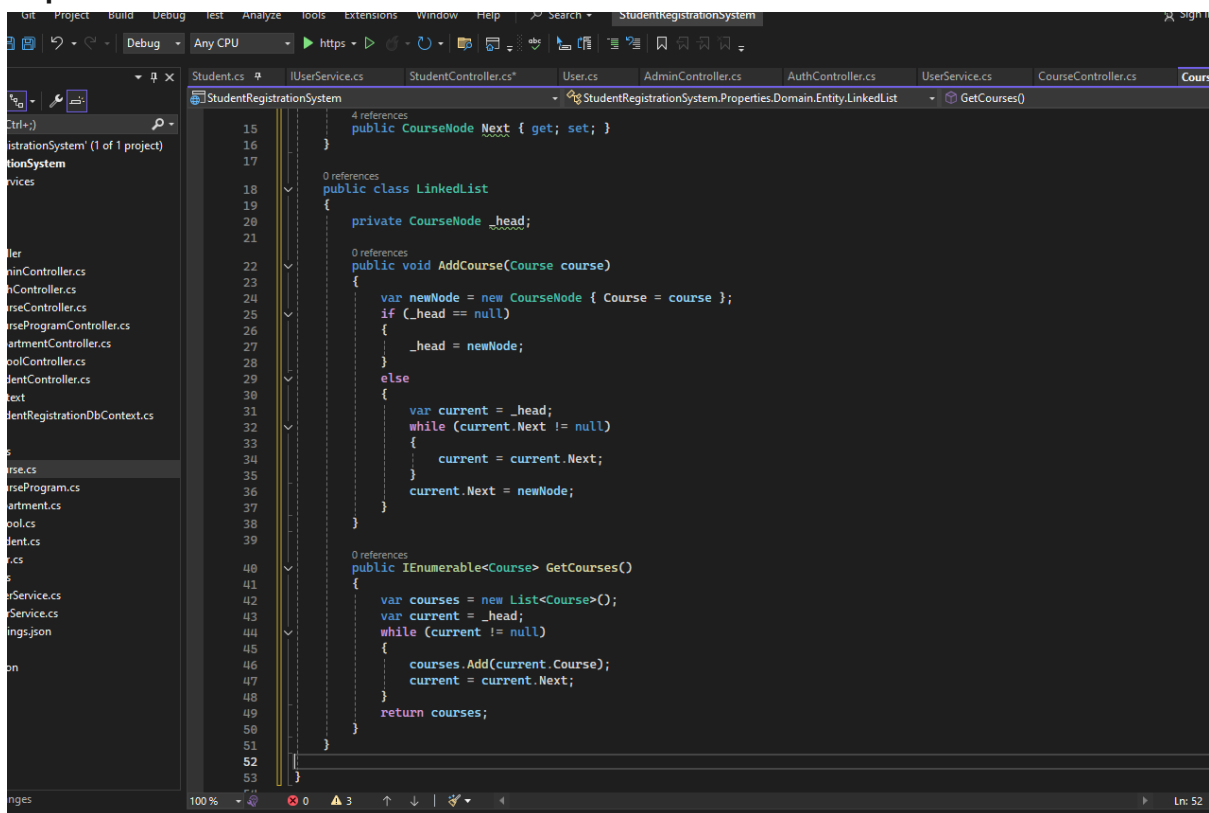
    return Ok(students);
}
```

## 2. Linked Lists

### Use Case: Manage Course Enrollments

- Linked lists can be used to manage a dynamic list of courses a student is enrolled in.

#### Implementation:



The screenshot shows the Visual Studio IDE with the 'StudentRegistrationSystem' project open. The 'CourseController.cs' file is selected, and the 'GetCourses()' method is being implemented. The code defines a 'CourseNode' class with a 'Next' property, a 'LinkedList' class with a '\_head' property, and a 'GetCourses()' method that returns an 'IEnumerable<Course>'.

```
15 public CourseNode Next { get; set; }
16
17
18 0 references
19 public class LinkedList
20 {
21     private CourseNode _head;
22
23     0 references
24     public void AddCourse(Course course)
25     {
26         var newNode = new CourseNode { Course = course };
27         if (_head == null)
28         {
29             _head = newNode;
30         }
31         else
32         {
33             var current = _head;
34             while (current.Next != null)
35             {
36                 current = current.Next;
37             }
38             current.Next = newNode;
39         }
40     }
41
42     0 references
43     public IEnumerable<Course> GetCourses()
44     {
45         var courses = new List<Course>();
46         var current = _head;
47         while (current != null)
48         {
49             courses.Add(current.Course);
50             current = current.Next;
51         }
52         return courses;
53     }
54 }
```

## 3. Stacks

### Use Case: Track Change History

- Stacks can be used to track changes to a student's records for undo functionality.

### Implementation:

```
227     }
228
229     0 references
230     public class StudentHistory
231     {
232         private readonly Stack<UpdateStudentDto> _history = new Stack<UpdateStudentDto>();
233
234         0 references
235         public void PushUpdate(UpdateStudentDto updateStudent)
236         {
237             _history.Push(updateStudent);
238         }
239
240         0 references
241         public UpdateStudentDto PopUpdate()
242         {
243             return _history.Count > 0 ? _history.Pop() : null;
244         }
245     }
246 }
247
```

## 4. Queues

### Use Case: Process Student Registration Requests

- Queues can be used to manage student registration requests in a first-come-first-serve manner.

### Implementation:

```
243     public class RegistrationQueue
244     {
245         private readonly Queue<Student> _queue = new Queue<Student>();
246
247         0 references
248         public void EnqueueStudent(Student student)
249         {
250             _queue.Enqueue(student);
251         }
252
253         0 references
254         public Student DequeueStudent()
255         {
256             return _queue.Count > 0 ? _queue.Dequeue() : null;
257         }
258     }
259 }
260
```

Incorporating data structures such as arrays, linked lists, stacks, queues, trees, and graphs into the StudentsController can enhance functionality and address specific requirements. Here's how you might integrate these structures:

### 1. Arrays

### Use Case: Pagination and Sorting

- Arrays can be used to handle pagination and sorting of student records.

#### Implementation:

csharp

Copy code

```
[HttpGet("paged")]
```

```
public async Task<ActionResult<IEnumerable<Student>>> GetPagedStudents(int page  
= 1, int pageSize = 10)
```

```
{
```

```
    var students = await _context.Students
```

```
        .OrderBy(s => s.StudentId) // Or any other sorting criteria
```

```
        .Skip((page - 1) * pageSize)
```

```
        .Take(pageSize)
```

```
        .ToListAsync();
```

```
    return Ok(students);
```

```
}
```

## 2. Linked Lists

### Use Case: Manage Course Enrollments

- Linked lists can be used to manage a dynamic list of courses a student is enrolled in.

#### Implementation:

csharp

Copy code

```
public class CourseNode
```

```
{
```

```
    public Course Course { get; set; }
```

```
    public CourseNode Next { get; set; }
```

```
}
```

```

public class LinkedList
{
    private CourseNode _head;

    public void AddCourse(Course course)
    {
        var newNode = new CourseNode { Course = course };
        if (_head == null)
        {
            _head = newNode;
        }
        else
        {
            var current = _head;
            while (current.Next != null)
            {
                current = current.Next;
            }
            current.Next = newNode;
        }
    }

    public IEnumerable<Course> GetCourses()
    {
        var courses = new List<Course>();
        var current = _head;
        while (current != null)
        {

```

```

        courses.Add(current.Course);

        current = current.Next;
    }

    return courses;
}
}

```

### 3. Stacks

#### Use Case: Track Change History

- Stacks can be used to track changes to a student's records for undo functionality.

#### Implementation:

csharp

Copy code

```

public class StudentHistory
{
    private readonly Stack<UpdateStudentDto> _history = new
    Stack<UpdateStudentDto>();

    public void PushUpdate(UpdateStudentDto updateStudent)
    {
        _history.Push(updateStudent);
    }

    public UpdateStudentDto PopUpdate()
    {
        return _history.Count > 0 ? _history.Pop() : null;
    }
}

```

### 4. Queues

#### Use Case: Process Student Registration Requests

- Queues can be used to manage student registration requests in a first-come-first-serve manner.

### **Implementation:**

csharp

Copy code

```
public class RegistrationQueue
{
    private readonly Queue<Student> _queue = new Queue<Student>();

    public void EnqueueStudent(Student student)
    {
        _queue.Enqueue(student);
    }

    public Student DequeueStudent()
    {
        return _queue.Count > 0 ? _queue.Dequeue() : null;
    }
}
```

## **5. Trees**

### **Use Case: Hierarchical Data Management (e.g., Departments, Schools)**

- Trees can be used to manage hierarchical relationships such as schools and departments.

### **Implementation:**

```
18 1 reference
19 public class Tree
20 {
21     private readonly DepartmentNode _root;
22
23     0 references
24     public Tree(Department rootDepartment)
25     {
26         _root = new DepartmentNode { Department = rootDepartment };
27     }
28
29     0 references
30     public void AddChild(Department parentDepartment, Department childDepartment)
31     {
32         var parentNode = FindNode(_root, parentDepartment);
33         if (parentNode != null)
34         {
35             parentNode.Children.Add(new DepartmentNode { Department = childDepartment });
36         }
37     }
38
39     2 references
40     private DepartmentNode FindNode(DepartmentNode root, Department department)
41     {
42         if (root.Department == department)
43         {
44             return root;
45         }
46
47         foreach (var child in root.Children)
48         {
49             var result = FindNode(child, department);
50             if (result != null)
51             {
52                 return result;
53             }
54         }
55         return null;
56     }
57 }
```

## 6. Graphs

### Use Case: Course Prerequisites Management

- Graphs can be used to manage and visualize course prerequisites.

### Implementation:



```
257 5 references
258 public class CourseNode
259 {
260     3 references
261     public Course Course { get; set; }
262     1 reference
263     public List<CourseNode> Prerequisites { get; set; } = new List<CourseNode>();
264 }
265
266 0 references
267 public class CourseGraph
268 {
269     private readonly List<CourseNode> _courses = new List<CourseNode>();
270
271     0 references
272     public void AddCourse(Course course)
273     {
274         _courses.Add(new CourseNode { Course = course });
275     }
276
277     0 references
278     public void AddPrerequisite(Course course, Course prerequisite)
279     {
280         var courseNode = _courses.FirstOrDefault(c => c.Course == course);
281         var prerequisiteNode = _courses.FirstOrDefault(c => c.Course == prerequisite);
282
283         if (courseNode != null && prerequisiteNode != null)
284         {
285             courseNode.Prerequisites.Add(prerequisiteNode);
286         }
287     }
288 }
```

## Error Handling

Implementing robust error handling and validation for user inputs is crucial for ensuring your application is secure and reliable are through out the application for improved error handling and validation:

1. Validation: Ensure that your data models and request objects are properly validated using data annotations.
2. Error Handling: Handle various error scenarios such as invalid requests, user already exists, and roles validation.

## Testing Strategy

### Unit Testing

**Objective:** Ensure the correctness of individual components and functions.

- **Backend (ASP.NET Web API):** Write unit tests for critical services, controllers, and business logic. Focus on methods handling student registration, account management, and data validation. Use testing frameworks like xUnit or NUnit to create test cases that cover different scenarios and edge cases.
- **Frontend (Angular):** Test components, services, and directives to verify that they function correctly in isolation. Use Angular's testing utilities like Jasmine and Karma to test UI interactions, data binding, and component lifecycle methods.

### Integration Testing

**Objective:** Verify that various parts of the application work together as intended.

- **Backend:** Conduct integration tests to ensure the API endpoints interact correctly with the database and other services. Test scenarios such as student registration workflows, data retrieval, and role-based access control. Use tools like Postman or integration testing frameworks to simulate interactions and validate the integrated components.
- **Frontend:** Test interactions between Angular components and backend APIs. Ensure that form submissions, data fetching, and error handling work seamlessly. Utilize tools like Protractor or Cypress for end-to-end testing, simulating user interactions and validating system behavior.

### Performance Testing

**Objective:** Evaluate the efficiency of algorithms and data handling.

- **Backend:** Perform performance testing to assess the response times and resource usage of critical API endpoints. Use profiling tools to identify bottlenecks in algorithms and database queries. Focus on high-load scenarios, such as multiple simultaneous student registrations or data retrieval operations.
- **Frontend:** Test the performance of Angular applications, focusing on rendering times and responsiveness. Evaluate the efficiency of data processing and interaction handling. Use tools like Lighthouse or WebPageTest to measure load times and identify performance issues.

By following this testing strategy, you can ensure that your student registration system is robust, efficient, and performs well under various conditions.