**Problem Statement: Enterprise Employee Management System (EEMS)**

**Overview:**

You are tasked with designing and developing a robust, scalable, and secure **Enterprise Employee Management System (EEMS)** using **ASP.NET Core MVC on .NET 8.0**. The system will manage employees, departments, roles, and user access, and must follow modern architectural patterns and best practices.

**Project Goals:**

* Implement a layered architecture with clear separation of concerns.
* Use **Entity Framework Core** for data access with **Repository** and **Unit of Work** patterns.
* Apply **Fluent Validation** and **Data Annotations** for input validation.
* Secure the application using **ASP.NET Core Identity**, **Claims-based Authorization**, and **Role-based Access Control**.
* Use **Middleware**, **Action Filters**, and **Dependency Injection** to enhance modularity and maintainability.

**Functional Requirements:**

**1. User Management**

* Register, login, and logout functionality.
* Role-based access: Admin, Manager, Employee.
* Claims-based authorization for fine-grained access control.
* Password reset and email confirmation (optional).

**2. Employee Module**

* CRUD operations for employees.
* Each employee belongs to a department and has a role.
* Display employee details with department and role info.
* Implement **Master-Detail** view: Department → Employees.

**3. Department Module**

* CRUD operations for departments.
* Prevent deletion of departments with assigned employees.

**4. Role Management**

* Admin can create and assign roles.
* Roles are used for authorization and UI control.

**5. Business Logic Layer**

* Implement core business rules such as:
  + Prevent duplicate employee emails.
  + Validate department capacity.
  + Auto-assign default roles on registration.

**Technical Requirements:**

**1. ASP.NET Core MVC Fundamentals**

* Use appsettings.json and environment-specific configuration files.
* Implement configuration loading using IConfiguration.

**2. Entity Framework Core**

* Use DbContext to manage database operations.
* Configure entity relationships and mappings in OnModelCreating.
* Apply **Fluent Validation** for complex rules.
* Use **Migrations** to manage schema changes.

**3. Dependency Injection**

* Register services, repositories, and validators using DI.
* Use AddScoped, AddTransient, and AddSingleton appropriately.

**4. Middleware**

* Create custom middleware to log requests and responses.
* Use built-in middleware for exception handling and static files.

**5. Views and Razor Pages**

* Use **HTML Helpers** and **Tag Helpers** for form generation.
* Display validation messages and bind model data.
* Use partial views for reusable UI components.

**6. Controllers & Routing**

* Use **Attribute Routing** for clean and RESTful endpoints.
* Implement [HttpGet], [HttpPost], [Route], and [FromRoute].

**7. Action Filters**

* Create custom filters for logging, validation, and authorization.
* Apply filters globally and at controller/action level.

**8. Authentication & Authorization**

* Use **ASP.NET Core Identity** for user management.
* Implement **Claims-based Authorization** for granular control.
* Use [Authorize], [AllowAnonymous], and role-based policies.

**Non-Functional Requirements:**

* Responsive UI using Bootstrap or Tailwind CSS.
* Secure password storage and user data protection.
* Error handling and logging using ILogger.
* Unit testing for business logic and data access layers.

**Deliverables:**

1. Source code hosted on GitHub or GitLab.
2. Database schema and migration scripts.
3. Technical documentation:
   * Architecture overview
   * Setup instructions
   * API endpoints
   * Validation rules