EIU
TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

Practice Assignment – Quarter 3, 2024-2025
Course Name: .Net Programming
Course Code: CSE 443
Student's Full Name:
Student ID:

Practice Assignment 7

LIBRARY MANAGEMENT SYSTEM

Following previous Practice Assignment, further develop the new layout and add model in the MVC framework

----

Exercise 1: Implement Authentication for User Login

Objective: Implement a login system to authenticate users and secure access to the admin panel.

Tasks:

- **Setup ASP.NET Core Identity**: Use **ASP.NET Core Identity** to manage user authentication.
- **Create Login Action**: Add a login action in `AccountController` to authenticate users with `UserManager` and `SignInManager`.
- **Create Login View**: Develop a simple login form where users input their credentials (This was completed in a previous practice assignment.).
- **Redirect for Unauthorized Access**: Configure the application to redirect unauthorized users to the login page if they attempt to access restricted areas.
- **Session Validation**: Set the session timeout in the application configuration. The session will automatically expire based on this setting, redirecting users to the login page if they try to access a restricted area after the session ends.

Expected Outcome: Users can log in to access restricted areas of the application. Unauthorized users are redirected to the login page.

Exercise 2: Implement Role-Based Authorization for Admin Access in MVC

Objective: Set up role-based authorization so that only users with the "Admin" role can access certain sections of the admin panel.

Tasks:

1. **Define Roles:**
   - Create a dynamic method to fetch roles from the database using a `RoleService`.
   - Ensure you can add, modify, or remove roles through the application interface.
2. **Secure Actions Using [Authorize(Roles = "Admin")]:**
   - Apply the `[Authorize]` attribute with roles to specific controller actions, restricting them to admins.
   - Utilize the `RoleService` to check if the current user is in the "Admin" role dynamically.
3. **Update UI Based on Role:**
   - **Server-Side Role Check:**
     - On the server side, retrieve the user's roles during the initial page load.

        - Render specific elements or features in the view based on the user's roles.
    - **Use Razor Syntax:**
        - Use Razor syntax to conditionally display elements based on the user's role.

```
@if (User.IsInRole("Admin"))
{
    <button>Edit User</button>
    <button>Manage Books</button>
}
```

4. **Ensure Secure Views:**
    - Make sure that the views are secured so that unauthorized users cannot access elements meant for admins.
    - Use the role management service to dynamically check roles in views and controllers.
5. **Test the Implementation:**
    - Create test users with different roles (Admin, User) to verify that access control is functioning as expected.
    - Attempt to access restricted areas as different users to ensure that only admins can view and interact with admin features.

Expected Outcome:
Only users with the "Admin" role can see and access designated areas and features in the admin panel. The UI is rendered dynamically based on the user's role, ensuring that unauthorized users do not have access to restricted elements.

Exercise 3: Create API Endpoints for Book Management and Use AJAX to Interact with Them

Objective: Set up CRUD API endpoints for managing books and interact with these endpoints via AJAX on the client-side.
Tasks:

- **API Controller for Books**: Create a `BooksController` in the API folder with endpoints for Create, Read, Update, and Delete.
- **Secure API Endpoints**: Use `[Authorize]` on the API controller to secure these endpoints.
- **AJAX CRUD Implementation**:
    - **Create**: Use an AJAX POST request to send new book data to the API.
    - **Read**: Use an AJAX GET request to fetch and display a list of books.
    - **Update**: Use an AJAX PUT request to update existing book information.
    - **Delete**: Use an AJAX DELETE request to remove a book from the list.
- **Handle AJAX Responses**: Show success or error messages to the user based on the response from the API.

Expected Outcome: Users can manage books (create, read, update, delete) through AJAX interactions with the API, with feedback displayed on the client-side.

Exercise 4: Implement API Endpoints and AJAX CRUD for User Management with Role-Based Authorization

Objective: Implement API endpoints for managing user accounts with AJAX and enforce role-based authorization to restrict these operations to admins.

Tasks:

- **API Controller for Users**: Create a `UsersController` in the API folder with CRUD actions for managing users.
- **Role-Based Authorization**: Use `[Authorize(Roles = "Admin")]` to restrict the API endpoints to admin users only.
- **AJAX CRUD for User Management**:
    - **Create**: Use AJAX to send user registration data to the API.
    - **Read**: Fetch a list of users via AJAX and display them in a grid.
    - **Update**: Send updated user details via AJAX to update user information.
    - **Delete**: Use AJAX to delete users, prompting the admin to confirm before deletion.
- **Conditional UI for Admins**: Ensure that user management features only appear for users with the "Admin" role.

Expected Outcome: Admins can manage user accounts via API calls through AJAX. Non-admin users cannot access or view user management features.

Exercise Guide

This is a general guideline for reference on implementation. You should only use it as a reference to carry out your practice exercises, as there may be differences in each practice.

Exercise 1: Implement Authentication for User Login

Steps to Implement the Login System:

1. Setup ASP.NET Core Identity

a. Install Required NuGet Packages: Ensure you have the necessary packages installed in your project. You can use the following command in the Package Manager Console:

```
Install-Package Microsoft.AspNetCore.Identity.EntityFrameworkCore
```

b. Configure Identity in `Startup.cs`: Update the `ConfigureServices` method to add Identity services.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")))
;

    services.AddIdentity<IdentityUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddControllersWithViews();
}
```

c. Update Database: Ensure that you have set up migrations and updated the database to include Identity tables.

```
Add-Migration InitialCreate
Update-Database
```

2. Create Login Action in `AccountController`

a. Create AccountController: If you haven't already, create a controller named `AccountController`.

```
public class AccountController : Controller
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly SignInManager<IdentityUser> _signInManager;

    public AccountController(UserManager<IdentityUser> userManager,
SignInManager<IdentityUser> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }
```

b. Implement the Login Action:

```
[HttpGet]
public IActionResult Login()
{
    return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await _signInManager.PasswordSignInAsync(model.Email,
model.Password, model.RememberMe, false);
        if (result.Succeeded)
        {
            return RedirectToAction("Index", "Home"); // Redirect to a
protected page
        }
        ModelState.AddModelError(string.Empty, "Invalid login attempt.");
    }
    return View(model);
}
```

3. Create Login View

a. Create a ViewModel for Login:

```
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    public bool RememberMe { get; set; }
}
```

b. Create the Login View (`Login.cshtml`):

```
@model LoginViewModel

<form asp-action="Login" method="post">
    <div>
        <label asp-for="Email"></label>
        <input asp-for="Email" />
        <span asp-validation-for="Email"></span>
    </div>
    <div>
        <label asp-for="Password"></label>
        <input asp-for="Password" />
        <span asp-validation-for="Password"></span>
    </div>
    <div>
        <input asp-for="RememberMe" /> Remember Me
    </div>
    <button type="submit">Login</button>
</form>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

4. Redirect for Unauthorized Access

a. Configure Authorization: In `Startup.cs`, add the following to configure the authentication scheme:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // other configurations...

    app.UseAuthentication();
    app.UseAuthorization();
}
```

b. Apply [Authorize] Attribute: Use the `[Authorize]` attribute on your controllers or actions that need protection.

```
[Authorize]
public class AdminController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

5. Session Validation

a. Configure Session Timeout: You can set the session timeout in the `ConfigureServices` method.

```
services.ConfigureApplicationCookie(options =>
{
    options.LoginPath = "/Account/Login";
    options.LogoutPath = "/Account/Logout";
    options.ExpireTimeSpan = TimeSpan.FromMinutes(30);
    options.SlidingExpiration = true;
});
```

b. Redirect on Expired Session: In your views or controllers, check if the user is authenticated and redirect to the login page if not.

Exercise 2: Implement Role-Based Authorization for Admin Access in MVC

1. Define Roles

- **Create a Dynamic Method to Fetch Roles:**
  - Create a service that interacts with your database to fetch roles. Below is an example of a `RoleService`:

    ```
    csharp
    Copy code
    public interface IRoleService
    {
        Task<List<string>> GetAllRolesAsync();
        Task<bool> IsUserInRoleAsync(string userId, string role);
    }

    public class RoleService : IRoleService
    {
        private readonly RoleManager<IdentityRole> _roleManager;
    ```

```
    public RoleService(RoleManager<IdentityRole> roleManager)
    {
        _roleManager = roleManager;
    }

    public async Task<List<string>> GetAllRolesAsync()
    {
        return await _roleManager.Roles.Select(r =>
r.Name).ToListAsync();
    }

    public async Task<bool> IsUserInRoleAsync(string userId,
string role)
    {
        var user = await _userManager.FindByIdAsync(userId);
        return await _userManager.IsInRoleAsync(user, role);
    }
}
```

- **Ensure Role Management:**
  - Allow admins to add, modify, or remove roles through a UI form. Create a page with a form for role management.
  - For example, you can create an `AdminController` with actions to handle role creation and management.

2. Secure Actions Using [Authorize(Roles = "Admin")]

- **Apply the [Authorize] Attribute:**
  - Use the `[Authorize]` attribute in your controllers to secure actions. For example:

```
csharp
Copy code
[Authorize(Roles = "Admin")]
public IActionResult AdminDashboard()
{
    return View();
}
```

- **Utilize the RoleService:**
  - In any action method, you can check if the current user is an admin using the role service:

```
csharp
Copy code
public class AdminController : Controller
{
    private readonly IRoleService _roleService;

    public AdminController(IRoleService roleService)
    {
        _roleService = roleService;
    }

    public async Task<IActionResult Index()
    {
        var userId =
User.FindFirstValue(ClaimTypes.NameIdentifier);
        var isAdmin = await
_roleService.IsUserInRoleAsync(userId, "Admin");
```

```
                if (!isAdmin)
                {
                    return RedirectToAction("AccessDenied", "Account");
                }

                return View();
            }
        }
```

3. Update UI Based on Role

- **Server-Side Role Check:**
  - During the initial page load, retrieve and check the user's roles. Render UI elements conditionally.
- **Use Razor Syntax:**
  - In your Razor views, conditionally render UI elements based on user roles:

```
html
Copy code
@if (User.IsInRole("Admin"))
{
    <button>Edit User</button>
    <button>Manage Books</button>
}
```

4. Ensure Secure Views

- **Secure Views:**
  - Use the `[Authorize]` attribute for controller actions and ensure views that should only be accessed by admins are not publicly accessible.
  - Always check roles when rendering views:

```
csharp
Copy code
@if (User.IsInRole("Admin"))
{
    <div>
        <!-- Admin-specific content -->
    </div>
}
```

Exercise 3: Create API Endpoints for Book Management and Use AJAX to Interact with Them

Steps to Complete Exercise 3

1. Create API Controller for Books

- **Create the `BooksController`:**
  - In your project, create a new folder called `API` if it doesn't already exist.
  - Add a new controller named `BooksController.cs` in the `API` folder.

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using YourProjectNamespace.Models; // Adjust as necessary
using YourProjectNamespace.Services; // Assuming you have a service for
handling book data
using System.Collections.Generic;
using System.Threading.Tasks;

[Route("api/[controller]")]
[ApiController]
[Authorize] // Secures the entire controller
```

```
public class BooksController : ControllerBase
{
    private readonly IBookService _bookService; // Assume you have an
IBookService interface

    public BooksController(IBookService bookService)
    {
        _bookService = bookService;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<Book>>> GetBooks()
    {
        var books = await _bookService.GetAllBooksAsync();
        return Ok(books);
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<Book>> GetBook(int id)
    {
        var book = await _bookService.GetBookByIdAsync(id);
        if (book == null) return NotFound();
        return Ok(book);
    }

    [HttpPost]
    public async Task<ActionResult<Book>> CreateBook(Book book)
    {
        var createdBook = await _bookService.CreateBookAsync(book);
        return CreatedAtAction(nameof(GetBook), new { id = createdBook.Id },
createdBook);
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateBook(int id, Book book)
    {
        if (id != book.Id) return BadRequest();
        await _bookService.UpdateBookAsync(book);
        return NoContent();
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteBook(int id)
    {
        await _bookService.DeleteBookAsync(id);
        return NoContent();
    }
}
```

2. Secure API Endpoints

- As shown above, the `[Authorize]` attribute is applied to the `BooksController`. This ensures that only authenticated users can access these endpoints.

3. AJAX CRUD Implementation

- **Include jQuery in your view:** Make sure you have jQuery included in your view so you can use it for AJAX calls.
- **AJAX Calls:** You will make AJAX calls for CRUD operations in your JavaScript code.

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<script>
    $(document).ready(function () {
        // AJAX GET request to fetch books
        function loadBooks() {
            $.ajax({
                url: '/api/books',
                type: 'GET',
                success: function (data) {
                    $('#bookList').empty();
                    data.forEach(function (book) {
                        $('#bookList').append('<li>' + book.title + ' <button
onclick="editBook(' + book.id + ')">Edit</button> <button
onclick="deleteBook(' + book.id + ')">Delete</button></li>');
                    });
                }
            });
        }

        // AJAX POST request to create a new book
        $('#createBookForm').submit(function (e) {
            e.preventDefault();
            const bookData = {
                title: $('#title').val(),
                author: $('#author').val(),
                // other fields...
            };
            $.ajax({
                url: '/api/books',
                type: 'POST',
                contentType: 'application/json',
                data: JSON.stringify(bookData),
                success: function () {
                    loadBooks();
                    alert('Book created successfully!');
                },
                error: function () {
                    alert('Error creating book.');
                }
            });
        });

        // AJAX PUT request to update a book
        function editBook(id) {
            // Show a form to edit the book and populate it with existing
data
            // On form submit, make an AJAX PUT request to update the book
            $('#editBookForm').submit(function (e) {
                e.preventDefault();
                const updatedBookData = {
                    id: id,
                    title: $('#editTitle').val(),
                    author: $('#editAuthor').val(),
                    // other fields...
                };
                $.ajax({
                    url: '/api/books/' + id,
                    type: 'PUT',
                    contentType: 'application/json',
                    data: JSON.stringify(updatedBookData),
                    success: function () {
                        loadBooks();
                        alert('Book updated successfully!');
                    },
                    error: function () {
                        alert('Error updating book.');
                    }
                });
            });
        }

        // AJAX DELETE request to delete a book
        function deleteBook(id) {
            $.ajax({
                url: '/api/books/' + id,
                type: 'DELETE',
                success: function () {
                    loadBooks();
                    alert('Book deleted successfully!');
                },
                error: function () {
                    alert('Error deleting book.');
                }
            });
        }

        loadBooks(); // Load books on page load
    });
</script>
```

4. Handle AJAX Responses

- Display success or error messages using `alert()` or another UI method (like showing a notification banner) based on the response from the API.