ASP.NET Core - True Ultimate Guide

Section 25: Identity and Authorization - Notes

ASP.NET Core Identity & Authorization

ASP.NET Core Identity is a robust and flexible membership system that enables you to add authentication and authorization features to your web applications. It provides essential building blocks for managing user accounts, passwords, roles, claims, tokens, and more.

Purpose and When to Use

- **User Authentication:** Verify user identities and control access to protected resources.
- **User Management:** Handle user registration, login, logout, password reset, and profile management.
- **Role-Based Authorization:** Restrict access to specific actions or features based on user roles.
- **Claims-Based Authorization:** Make authorization decisions based on claims (attributes) associated with a user.
- **External Login Providers:** Integrate with external authentication providers (Google, Facebook, etc.).
- **Two-Factor Authentication (2FA):** Add an extra layer of security to your login process.

Key Concepts

- **Identity Models:**
    - **ApplicationUser:** Extends the IdentityUser class to represent your application's users. It can include additional properties like PersonName (in your example).
    - **ApplicationRole:** Extends the IdentityRole class to define roles within your application.
- **UserManager<TUser>:** A core service for managing user accounts (creating, deleting, finding, updating).
- **SignInManager<TUser>:** Handles user sign-in and sign-out operations.
- **RoleManager<TRole>:** Manages roles and their assignments to users.

Detailed Code Explanation

AccountController:

- **Constructor:** The constructor injects the UserManager, SignInManager, and RoleManager services.
- **Register (GET):** Displays the registration form.
- **Register (POST):**
    1. **Model Validation:** Checks if the submitted RegisterDTO is valid.
    2. **User Creation:** Creates an ApplicationUser based on the RegisterDTO data.
    3. **Role Handling:**
        - Creates the Admin or User role if it doesn't exist.
        - Assigns the user to the selected role.
    4. **Sign-In:** Signs the user in upon successful registration.
    5. **Redirect:** Redirects to the PersonsController.Index action.
    6. **Error Handling:** If user creation fails, adds errors to the ModelState and re-renders the Register view.
- **Login (GET):** Displays the login form.
- **Login (POST):**
    1. **Model Validation:** Checks if the submitted LoginDTO is valid.
    2. **Sign-In Attempt:** Attempts to sign in the user using _signInManager.PasswordSignInAsync.
    3. **Redirect (if successful):**
        - Redirects to the Admin area's Home/Index if the user is an admin.
        - Redirects to the ReturnUrl (if provided and valid) or to PersonsController.Index otherwise.
    4. **Error Handling:** If sign-in fails, adds an error to the ModelState and re-renders the Login view.
- **Logout:**
    - **[Authorize]:** Ensures the user is authenticated before accessing this action.
    - **Signs the user out:** Uses _signInManager.SignOutAsync.
    - **Redirect:** Redirects to PersonsController.Index.
- **IsEmailAlreadyRegistered:**
    - **[AllowAnonymous]:** Allows anonymous users to access this action (useful for client-side validation).

- **Checks for Existing User:** Uses _userManager.FindByEmailAsync to see if a user with the given email already exists.
- **Returns JSON:** Returns true if the email is available (no user found) and false otherwise.

Program.cs:

- **AddControllersWithViews():** Enables MVC controllers and views.
- **AddIdentity<ApplicationUser, ApplicationRole>:** Configures ASP.NET Core Identity with your custom user and role types.
  - options.Password: Sets password complexity requirements.
- **AddEntityFrameworkStores<ApplicationDbContext>:** Configures EF Core to store Identity data in your ApplicationDbContext.
- **AddDefaultTokenProviders():** Adds default token providers for features like password reset and two-factor authentication.
- **AddUserStore, AddRoleStore:** Configures specific stores for user and role data.
- **AddAuthorization():** Sets up authorization policies.
  - FallbackPolicy: Default policy applied if no specific policy is specified.
  - AddPolicy("NotAuthorized", ...): Custom policy to allow only unauthenticated users.
- **ConfigureApplicationCookie():** Customizes the cookie authentication options, setting the login path.
- **AddHttpLogging():** Enables HTTP logging with specified fields.

Notes:

- **IdentityUser and IdentityRole:** Extend these base classes for your custom user and role models.
- **UserManager, SignInManager, RoleManager:** Core services for managing users, sign-in/out, and roles.
- **[Authorize]:** Attribute to restrict access to authenticated users.
- **[AllowAnonymous]:** Attribute to allow anonymous access.
- **Authorization Policies:** Use AddAuthorization to define custom policies.
- **Password Complexity:** Configure password requirements in AddIdentity.
- **Tag Helpers:** In views, use tag helpers like asp-controller, asp-action, asp-for, and asp-validation-for.

- **Client-Side Validation:** You can enhance user experience by adding client-side validation using JavaScript libraries.
- **ReturnUrl:** Used to redirect users back to their original destination after logging in.
- **Remote Validation:** Allows for server-side validation of user input on the client-side using AJAX.
- **Areas:** Organize large applications into logical areas, each with its own set of controllers, views, and models.
- **[Area] Attribute:** Use to associate controllers with specific areas.
- **HTTPS:** Enable HTTPS in production for secure communication.
- **XSRF (Cross-Site Request Forgery) Protection:** Use [ValidateAntiForgeryToken] and tag helpers in forms to prevent CSRF attacks.

Key Points to Remember

ASP.NET Core Identity

- **Purpose:** Robust membership system for user authentication and authorization.
- **Key Features:**
    - User registration, login, logout
    - Password management (hashing, reset)
    - Role-based and claims-based authorization
    - External login providers (Google, Facebook, etc.)
    - Two-factor authentication (2FA)

Identity Models

- **ApplicationUser:** Extends IdentityUser to represent your app's users.
- **ApplicationRole:** Extends IdentityRole to define roles in your app.

Key Services

- **UserManager<TUser>:** Manages user accounts (create, delete, find, update).
- **SignInManager<TUser>:** Handles user sign-in and sign-out.
- **RoleManager<TRole>:** Manages roles and their assignments to users.

Views and Controllers

- **Login and Logout Buttons:** Use asp-controller and asp-action tag helpers to create login/logout links.
- **Active Nav Link:** Use a custom tag helper or CSS classes to highlight the active navigation link.
- **Remote Validation:** Use [Remote] attribute on model properties for server-side validation during form input.
- **Conventional Routing:** Use route attributes ([Route]) to define routes for actions like Register and Login.

Security

- **Password Complexity:** Configure password requirements in AddIdentity (e.g., length, special characters).
- **Authorization Policies:** Use [Authorize] and Authorize(policyName) to protect actions and define custom policies.
- **ReturnUrl:** In login forms, use ReturnUrl to redirect users back to their original destination.
- **HTTPS: Always enable HTTPS in production** to encrypt sensitive data.
- **XSRF (Cross-Site Request Forgery) Protection:**
    - Use [ValidateAntiForgeryToken] in actions and @Html.AntiForgeryToken() in forms to prevent CSRF attacks.

Additional Concepts

- **User Roles:** Assign users to roles to control access to features.
- **Areas:** Organize large applications into logical areas, each with its own set of controllers and views.
- **Claims-Based Authorization:** Make authorization decisions based on claims associated with the user.

Code Snippets

- **Registering a user:**

```
ApplicationUser user = new ApplicationUser() { /* ... */ };

IdentityResult result = await _userManager.CreateAsync(user, registerDTO.Password);
```

- **Signing in a user:**

```
var result = await _signInManager.PasswordSignInAsync(loginDTO.Email, loginDTO.Password,
isPersistent: false, lockoutOnFailure: false);
```

- **Checking user roles:**

```
if (await _userManager.IsInRoleAsync(user, "Admin")) { /* ... */ }
```

Interview Tips

- **Concepts:** Explain authentication vs. authorization, role-based vs. claims-based authorization.
- **Implementation:** Demonstrate how you would set up user registration, login, and logout.
- **Security:** Emphasize the importance of security best practices (HTTPS, XSRF protection, password hashing).
- **Customization:** Discuss how you would customize Identity (e.g., adding user profile fields, creating custom authorization policies).