ASP.NET Core - True Ultimate Guide

Section 16: CRUD Operations

Controllers for Read and Create Operations

- **Read (Index Action):**
    - Typically, the Index action method in your controller will be responsible for handling the "read" operation. It retrieves data from a service layer (your business logic), processes it if needed, and passes it to a view for display.
    - Common tasks in the Index action:
        - Fetching a list of items from the database.
        - Applying filtering or sorting based on query parameters or user input.
        - Preparing a view model to package the data for the view.
    - Return type: Typically an IActionResult, often returning a ViewResult that renders the "Index" view.

- **Create (Create Actions):**
    - The Create action typically has two versions:
        - **HttpGet (Displaying the Form):** This action simply returns the "Create" view, which contains a form for the user to fill in.
        - **HttpPost (Processing the Form Submission):** This action receives the form data (usually via model binding), validates it, and if valid, passes it to the service layer to create the new entity in the database. Then, it typically redirects to the "Index" action to display the updated list of items.

Views for Read and Create Operations

- **Read (Index View):**
    - Displays a list or table of items fetched from the database.
    - May include filtering and sorting options to allow users to customize the view.
    - Could use partial views to break down complex UI elements (e.g., a partial view for each item in the list).
    - Should be strongly typed to a view model (e.g., IEnumerable<PersonResponse> in your code) for type safety and maintainability.

- **Create (Create View):**
    - Renders a form for collecting data to create a new item.
    - Uses tag helpers (<form>, <input>, <select>, etc.) or HTML helpers to generate the form elements.
    - Includes validation attributes on form fields to provide immediate feedback to the user.
    - Should be strongly typed to the appropriate model class (e.g., PersonAddRequest in your code) to take advantage of model binding and validation.

Implementing Sorting and Search in Views

Your code example already demonstrates sorting and searching functionality in the Persons/Index.cshtml view:

1. **Sorting:**
    - It uses ViewBag to pass sorting information (current sort column and order) to the view.
    - A partial view (_GridColumnHeader) is used to render each table column header as a link, which, when clicked, triggers a sort action.
    - The query string parameters (sortBy and sortOrder) are used to control the sorting logic on both the client and server sides.

2. **Searching:**
    - ViewBag is also used to pass search fields and the current search criteria to the view.
    - A form with a dropdown (searchBy) and a text input (searchString) allows users to filter the results.
    - The search parameters are submitted to the Index action, which then calls the service's GetFilteredPersons method to retrieve the filtered results.

Best Practices

- **Strongly Typed Views:** Always use strongly typed views and view models for type safety and maintainability.
- **Separation of Concerns:** Keep your views focused on presentation logic and delegate data access and manipulation to your controllers and services.

- **Partial Views:** Use partial views to create reusable UI components, especially for complex lists or grids.
- **Tag Helpers:** Utilize tag helpers to simplify form generation and interaction with models in your views.
- **Validation:** Implement validation in your models using data annotations and custom validation attributes.
- **Error Handling:** Handle potential errors gracefully and provide informative error messages to the user.
- **Client-Side Enhancement:** Use JavaScript to enhance the user experience with features like client-side validation, AJAX-based filtering/sorting, and dynamic updates.

Things to Avoid

- **Complex Logic in Views:** Avoid complex business logic in your views. Keep them simple and focused on presentation.
- **Tight Coupling:** Don't tightly couple your views to specific controllers or models. Strive for reusability.
- **Overuse of ViewBag or ViewData:** Prefer strongly typed models for passing data to your views.
- **Hardcoding Strings:** Avoid hardcoding strings like column names or sorting options. Use constants or enums for better maintainability.
- **Neglecting Security:** Always validate and sanitize user input, especially in search functionality, to prevent vulnerabilities.

Key points to remember

Controllers

- **Purpose:**
    - Handle HTTP requests (GET, POST, PUT, DELETE).
    - Interact with services to perform CRUD operations on data.
    - Select and return views with data.

- **Typical Actions:**
    - Index: Display a list of items (Read).

    - Details/{id}: Show details of a single item (Read).
    - Create: Display a form for creating a new item (Get) and process the form submission (Post).
    - Edit/{id}: Display a form for editing an existing item (Get) and process the form submission (Post).
    - Delete/{id}: Delete an item.

- **Model Binding:** Use parameters and attributes ([FromQuery], [FromRoute], [FromBody]) to bind data from the request.

- **Validation:**
    - Use data annotations in your models (e.g., [Required], [StringLength]).
    - Check ModelState.IsValid in actions and handle invalid input.

- **Redirects:**
    - Use RedirectToAction, RedirectToRoute, or Redirect after successful POST requests (Create/Edit/Delete) to prevent duplicate submissions.
    - Use LocalRedirect for redirects within your application.

- **Error Handling:** Return appropriate status codes (e.g., 400 Bad Request, 404 Not Found) for errors.

Views

- **Purpose:** Render the user interface (UI) for each action.
- **Razor View Engine:** Use Razor syntax (@Model, @Html, etc.) to embed C# code in your views.
- **Strongly Typed Views:** Use the @model directive to bind a view to a specific model class (view model).
- **Displaying Data:** Use Razor syntax to access and display model properties (e.g., @Model.Name).
- **Forms:** Use tag helpers or HTML helpers to create forms for Create and Edit actions.
- **Validation:** Display validation error messages using @Html.ValidationSummary() and @Html.ValidationMessageFor().

- **Partial Views:** Use partial views (_PartialName.cshtml) for reusable UI components (e.g., a list item template).
- **Layouts:** Use _Layout.cshtml to define the common structure for your pages.
- **_ViewImports.cshtml:** Centralize common @using and @addTagHelper directives.
- **Sorting and Filtering:**
    - Use query string parameters (e.g., ?sort=name_desc) to control sorting and filtering on the server-side.
    - Optionally use JavaScript for client-side filtering or sorting.

Additional Tips

- **Separate Concerns:** Keep your controllers focused on handling requests and your views focused on presentation.
- **Use Services:** Delegate data access and business logic to separate service classes.
- **DTO (Data Transfer Objects):** Use DTOs to transfer data between your controllers and services.
- **Asynchronous Actions:** Use async and await for actions that involve I/O operations.
- **Security:** Always validate and sanitize user input.

Interview Focus Areas

- **Understanding of MVC:** Explain the roles of controllers and views and how they interact.
- **Model Binding & Validation:** Show proficiency in using model binding and validation attributes.
- **View Logic:** Demonstrate how to write clear and concise Razor code in your views.
- **Error Handling & Redirects:** Explain how to handle errors gracefully and prevent duplicate form submissions.
- **Best Practices:** Discuss how to follow the best practices mentioned above to write clean and maintainable code.