Glossary Management System – Repository + Service Pattern (Mock Data)

Task 1: Create the Project and Structure

Problem Statement:
Set up an ASP.NET Core Web API project with clean separation: Controller → Service → Repository.

Solution Outline:

1. Run `dotnet new webapi -n GlossaryManagementSystem`.
2. Add folders:
   - `Models`
   - `DTOs/GlossaryItem`
   - `Controllers`
   - `Services/GlossaryItem`
   - `Repositories/GlossaryItem`
   - `Interfaces/Services`
   - `Interfaces/Repositories`

Task 2: Define the `GlossaryItem` Model

Problem Statement:
Define the model that represents a glossary entry.

Solution Outline:

1. Create `GlossaryItem.cs` in `Models`.
2. Add properties:
    - `Id` (int)
    - `Term` (string)
    - `Definition` (string)

Task 3: Create DTOs

Problem Statement:
Create DTOs to decouple the internal model from the API layer.

Solution Outline:

In `DTOs/GlossaryItem`, define:

- `GlossaryItemDTO`
- `CreateGlossaryItemDTO`
- `UpdateGlossaryItemDTO`

Task 4: Create Repository Interface and Implementation

Problem Statement:
Handle mock data using repository pattern.

Solution Outline:

1. In `Interfaces/Repositories`, create `IGlossaryRepository.cs`.
2. In `Repositories/GlossaryItem`, create `GlossaryRepository.cs`.
3. Use `List<GlossaryItem>` to simulate DB.
4. Methods:
    - `GetAll()`
    - `GetById(int id)`
    - `Add(GlossaryItem item)`
    - `Update(GlossaryItem item)`
    - `Delete(int id)`

Task 5: Create Service Interface and Implementation

Problem Statement:
Write business logic that interacts with the repository.

Solution Outline:

1. In `Interfaces/Services`, create `IGlossaryService.cs`.
2. In `Services/GlossaryItem`, create `GlossaryService.cs`.
3. Handle:
    - Mapping between DTOs and Models
    - Business validation or formatting logic
    - Calling repository methods

Task 6: Implement the `GlossaryController`

Problem Statement:
Receive HTTP requests and delegate to the service layer.

Solution Outline:

1. Create `GlossaryController.cs` in `Controllers`.
2. Inject `IGlossaryService`.
3. Expose endpoints:
    - `GET /api/glossary`
    - `GET /api/glossary/{id}`
    - `POST /api/glossary`
    - `PUT /api/glossary/{id}`
    - `DELETE /api/glossary/{id}`

Task 7: Add Data Validation

Problem Statement:
Validate incoming data at the DTO level.

Solution Outline:

1. Add `[Required]` and `[StringLength]` attributes to DTOs.
2. Use `ModelState.IsValid` inside the controller before passing to service.

Task 8: Filtering and Sorting (in Service)

Problem Statement:
Allow filtering glossary items by term and sorting alphabetically.

Solution Outline:

1. Add optional query params: `term`, `sortBy`, `isDescending`.
2. Handle logic in the **service** using LINQ.
3. Call the repository's `GetAll()` and apply filters/sorting in service.

Task 9: Implement Pagination (in Service)

Problem Statement:
Support pagination in the list endpoint.

Solution Outline:

1. Accept `PageNumber`, `PageSize` in query.
2. In the service, apply `.Skip().Take()` on filtered list.

Task 10: Seed Initial Mock Data (in Repository)

Problem Statement:
Pre-populate some glossary items.

Solution Outline:

1. In `GlossaryRepository` constructor, seed a few entries:
    - Example: Term: "API", Definition: "Application Programming Interface"

Task 11: Test with Swagger

Problem Statement:
Ensure everything works via Swagger UI.

Solution Outline:

1. Run the project.
2. Open Swagger at `/swagger`.
3. Test:
    - Add glossary item
    - Get all items (with pagination/filtering/sorting)
    - Update & delete items

✅ Clean Architecture – Repository + Service + Controller