Contents

[.NET CORE WEB API – FINAL ARCHITECTURE 4](#_Toc203380271)

[3. Define Domain Entities 5](#_Toc203380272)

[4. Create EF Core DbContext (Infrastructure) 5](#_Toc203380273)

[5. Implement Repository Pattern 6](#_Toc203380274)

[6. Add EF Core & MediatR Services (Infrastructure DI) 6](#_Toc203380275)

[7. Add Connection String & Configure Program.cs (WebApi) 7](#_Toc203380276)

[8. Run EF Migrations 7](#_Toc203380277)

[9. Implement CQRS with MediatR 8](#_Toc203380278)

[10. Create Controllers 8](#_Toc203380279)

[Repeat for VideoViewing, WebSearch. 8](#_Toc203380280)

[11. Summary of Capabilities 8](#_Toc203380281)

[. 9](#_Toc203380282)

[APPENDIX A 10](#_Toc203380283)

[Full code templates for commands, handlers, DTOs and validation using fluent API and also ensure that the entities for Book, YouTubeVideos and Searches all have a foreign key to their parent table "Resources" 10](#_Toc203380284)

[. 13](#_Toc203380285)

[APPENDIX B: 14](#_Toc203380286)

[Methods in the Entities for Progress Calculations 14](#_Toc203380287)

[. 15](#_Toc203380288)

[APPENDIX C 16](#_Toc203380289)

[Code for queries and commands, the IRepositories, Repositories Classes, Controllers for Get, Post, and the updates and deletes with explanation of how they all work together, along with any Entity Framework Core and LINQ code. 16](#_Toc203380290)

[1. IRepository & Repository Implementation 16](#_Toc203380291)

[IRepository<T> (Application Layer) 16](#_Toc203380292)

[Repository<T> (Infrastructure → Repositories) 17](#_Toc203380293)

[2. CQRS — Queries and Handlers (using MediatR) 18](#_Toc203380294)

[GetAllBooksQuery 18](#_Toc203380295)

[GetBookByIdQuery 18](#_Toc203380296)

[3. CQRS — Commands and Handlers 19](#_Toc203380297)

[CreateBookCommand 19](#_Toc203380298)

[UpdateBookCommand 19](#_Toc203380299)

[DeleteBookCommand 20](#_Toc203380300)

[4. DTOs (Data Transfer Objects) 20](#_Toc203380301)

[5. API Controllers 21](#_Toc203380302)

[BooksController 21](#_Toc203380303)

[6. Connecting Everything 22](#_Toc203380304)

[Repeating for Other Entities 22](#_Toc203380305)

[Summary 22](#_Toc203380306)

[APPENDIX D 23](#_Toc203380307)

[for the following in .NET Core Web API, 1) Book entity class: please add methods to calculate number of pages left to read, percentage of book read, percentage of book left to read 23](#_Toc203380308)

[APPENDIX E 25](#_Toc203380309)

[Please flesh out the code for queries and commands for CQRS, the IRepositories code, the implementation of the Repositories, and the Controllers for Get, Post, and the updates and deletes, and provide an explanation of how they all work together, along with any Entity Framework Core and LINQ code. 25](#_Toc203380310)

[APPENDIX F 30](#_Toc203380311)

[Specific IRepository interfaces and Repository implementations for book, YouTubeVideo and InternetSearches. 30](#_Toc203380312)

[1. IBookRepository Interface 31](#_Toc203380313)

[2. BookRepository Implementation 32](#_Toc203380314)

[3. Create Repositories for Other Entities 33](#_Toc203380315)

[APPENDIX G 34](#_Toc203380316)

[Can you create **specific** IRepository interfaces and Repository implementations for book, YouTubeVideo and InternetSearches. Is it better to create a generic IRepository though? 34](#_Toc203380317)

[APPENDIX H 37](#_Toc203380318)

[Show the code and how a **specific** Book entity, IBookRepository, BookRepository Implementation, CQRS query, CQRS Command, BookDTO, and BookController all work together, including the sequence of code execution. 37](#_Toc203380319)

[APPENDIX I 41](#_Toc203380320)

[MediatR Pipeline – What Is It? 41](#_Toc203380321)

[APPENDIX J 42](#_Toc203380322)

[Required to install **MediatR** from NPM to use the IRequest interface in CommandHandler code 42](#_Toc203380323)

[What is IRequest, 42](#_Toc203380324)

[Where does IRequest come from? 42](#_Toc203380325)

[How to define and use IRequest<BookDto?> 42](#_Toc203380326)

[1.Define the query ( e.g. - public record GetBookByIdQuery(Guid Id) : **IRequest**<BookDto?>; ) 42](#_Toc203380327)

[2.Define the handler: 43](#_Toc203380328)

[3.Register MediatR in Startup.cs or Program.cs (depending on the .NET version): 43](#_Toc203380329)

[4.Use the query in your controller: 43](#_Toc203380330)

[5.Summary 44](#_Toc203380331)

[APPENDIX K 45](#_Toc203380332)

[COMPLETE EXAMPLE leaving nothing out 45](#_Toc203380333)

[Also adding a SQL Server database and any Entity Framework Core and LINQ code necessary to ensure a complete example. 45](#_Toc203380334)

[1. Project Structure 45](#_Toc203380335)

[2. DTOs/BookDto.cs 45](#_Toc203380336)

[3. Entities/Book.cs 45](#_Toc203380337)

[4. Data/ApplicationDbContext.cs 46](#_Toc203380338)

[5. Data/InMemoryBookRepository.cs *(optional – for local seed/testing)* 46](#_Toc203380339)

[6. (Optional) Data/BookRepository.cs *(EF Core implementation)* 46](#_Toc203380340)

[7. IBookRepository.cs *(Interface)* 47](#_Toc203380341)

[8. Queries/GetBookByIdQuery.cs 47](#_Toc203380342)

[9. Queries/GetBookByIdQueryHandler.cs 47](#_Toc203380343)

[10. Controllers/BooksController.cs 48](#_Toc203380344)

[11. Program.cs (Entry point & DI setup) 49](#_Toc203380345)

[12. appsettings.json *(Connection string)* 49](#_Toc203380346)

[13. EF Core Migrations & Database Setup 50](#_Toc203380347)

[14. Testing the API 50](#_Toc203380348)

[Summary Quick View 50](#_Toc203380349)

[APPENDIX L 52](#_Toc203380350)

[EXPANDED EXAMPLE:A full enhancement of the project: 52](#_Toc203380351)

[a. CreateBookCommand (CQRS write pattern) 52](#_Toc203380352)

[b. FluentValidation (for input validation) 52](#_Toc203380353)

[c. EF Core seeding (automatic SQL Server DB seeding) 52](#_Toc203380354)

[d. Swagger setup and testing flow 52](#_Toc203380355)

[e. Optional: basic unit test setup (text-only guidance) 52](#_Toc203380356)

# .NET CORE WEB API – FINAL ARCHITECTURE

**My Prompt:** You are an expert .NET Web Core Web API developer in July 2025. Please provide the complete and discrete steps in Visual Studio 2022, for adding the entire code to a Clean Architecture style .NET Core Web API that uses the Repository pattern, and CQRS, that tracks a student's readings of books, viewing of YouTube videos, and web searches for the student's studies in Software Programming/Development/Architecture - e.g. 1) can return a list of books, videos, and web searches data, 2) can inserts new books, YouTube video info, and web searches data, 3) can update books, videos, and web searches data, and 4) is able to delete books, videos, and web searches data. Assume the student has not yet not implemented any Entity Framework Core code yet but do need to use Entity Framework Core with an actual SQL Server database (not in-memory) so please include any steps required for use of Entity Framework Core and SQL Server.

**ChatGPT said:**

Here’s a complete, step-by-step guide to implement a Clean Architecture-style .NET Core Web API (using .NET 7/8 in July 2025) with Entity Framework Core (SQL Server), Repository pattern, and CQRS (MediatR) to track student readings, YouTube viewings, and web searches:

**📁 1. Create the solution & projects**

In **Visual Studio 2022**:

1. **File → New → Project**, choose **ASP.NET Core Web API**, name it StudentTracker.WebApi.
2. Add new Class Library projects:
   * StudentTracker.Domain
   * StudentTracker.Application
   * StudentTracker.Infrastructure
3. Create a test project (optional): StudentTracker.Application.Tests.

Arrange folders like:

src\

├─ StudentTracker.Domain

├─ StudentTracker.Application

├─ StudentTracker.Infrastructure

└─ StudentTracker.WebApi

Set project references:

* **Domain** → none.
* **Application** → references **Domain**.
* **Infrastructure** → references **Application**.
* **WebApi** → references **Application**, **Infrastructure**.

[Microsoft Learn+15David Veksler's Cheatsheets+15LinkedIn+15](https://cheatsheets.davidveksler.com/clean-architecture-dotnet.html?utm_source=chatgpt.com)

**2. Install NuGet Packages**

| **Project** | **Packages** |
| --- | --- |
| **Infrastructure** | Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.SqlServer, Microsoft.EntityFrameworkCore.Design, Microsoft.EntityFrameworkCore.Tools |
| **Application** | MediatR, MediatR.Extensions.Microsoft.DependencyInjection |
| **WebApi** | Microsoft.EntityFrameworkCore.Design |

Use **Package Manager Console** or VS **NuGet UI**.

## 3. Define Domain Entities

In StudentTracker.Domain/Entities:

public class BookReading { public int Id { get; set; } … relevant properties }

public class VideoViewing { public int Id { get; set; } … }

public class WebSearch { public int Id { get; set; } … }

Also create *common interfaces* in Domain (if needed).

## 4. Create EF Core DbContext (Infrastructure)

In StudentTracker.Infrastructure.Persistence:

public class AppDbContext : DbContext, IAppDbContext

{

public DbSet<BookReading> BookReadings { get; set; }

public DbSet<VideoViewing> VideoViewings { get; set; }

public DbSet<WebSearch> WebSearches { get; set; }

public AppDbContext(DbContextOptions<AppDbContext> options)

: base(options) { }

}

Define IAppDbContext interface in Application to decouple from EF:

public interface IAppDbContext {

DbSet<BookReading> BookReadings { get; }

… SaveChangesAsync();

}

## 5. Implement Repository Pattern

Create generic interfaces in Application:

public interface IRepository<T> where T : class

{

Task<List<T>> GetAllAsync();

Task<T?> GetByIdAsync(int id);

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(int id);

}

Implement in Infrastructure under Repositories:

public class Repository<T> : IRepository<T> where T : class

{

private readonly AppDbContext \_ctx;

public Repository(AppDbContext ctx) => \_ctx = ctx;

public async Task<List<T>> GetAllAsync() => await \_ctx.Set<T>().ToListAsync();

public async Task<T?> GetByIdAsync(int id) => await \_ctx.Set<T>().FindAsync(id);

public async Task AddAsync(T e) { await \_ctx.Set<T>().AddAsync(e); await \_ctx.SaveChangesAsync(); }

public async Task UpdateAsync(T e) { \_ctx.Set<T>().Update(e); await \_ctx.SaveChangesAsync(); }

public async Task DeleteAsync(int id) { var e = await GetByIdAsync(id); if (e != null) { \_ctx.Set<T>().Remove(e); await \_ctx.SaveChangesAsync(); } }

}

Then register in DI:

services.AddScoped(typeof(IRepository<>), typeof(Repository<>));

[C# Corner+1Reddit+1](https://www.c-sharpcorner.com/article/clean-architecture-with-net-6-using-entity-framework/?utm_source=chatgpt.com)

## 6. Add EF Core & MediatR Services (Infrastructure DI)

In Infrastructure/DependencyInjection.cs:

public static IServiceCollection AddInfrastructure(this IServiceCollection services, IConfiguration config)

{

services.AddDbContext<AppDbContext>(opts =>

opts.UseSqlServer(config.GetConnectionString("DefaultConnection")));

services.AddScoped<IAppDbContext>(prov => prov.GetService<AppDbContext>());

services.AddScoped(typeof(IRepository<>), typeof(Repository<>));

services.AddMediatR(typeof(DependencyInjection).Assembly);

return services;

}

## 7. Add Connection String & Configure Program.cs (WebApi)

**appsettings.json**:

{

"ConnectionStrings": {

"DefaultConnection": "Server=...;Database=StudentTrackerDb;Trusted\_Connection=True;"

}

}

**Program.cs**:

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddInfrastructure(builder.Configuration);

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

app.MapControllers();

app.UseSwagger(); app.UseSwaggerUI();

app.Run();

[Reddit+11C# Corner+11Gowtham K - Coder & Traveler+11](https://www.c-sharpcorner.com/article/clean-architecture-with-net-6-using-entity-framework/?utm_source=chatgpt.com)

## 8. Run EF Migrations

In Package Manager Console:

Add-Migration InitialCreate -Project StudentTracker.Infrastructure

Update-Database

## 9. Implement CQRS with MediatR

For each entity (**BookReading**, **VideoViewing**, **WebSearch**), create Commands and Queries.

**Example for BookReading** in Application/Features/BookReadings:

* GetAllBookReadingsQuery : IRequest<List<BookReadingDto>>
* GetAllBookReadingsQueryHandler uses IRepository<BookReading>
* Similarly: CreateBookReadingCommand, UpdateBookReadingCommand, DeleteBookReadingCommand.

Use AutoMapper or manual mapping to DTOs.

## 10. Create Controllers

In WebApi/Controllers/BookReadingsController:

[ApiController]

[Route("api/[controller]")]

public class BookReadingsController : ControllerBase

{

private readonly IMediator \_mediator;

public BookReadingsController(IMediator mediator) => \_mediator = mediator;

[HttpGet] public async Task<IActionResult> Get() => Ok(await \_mediator.Send(new GetAllBookReadingsQuery()));

[HttpPost] public async Task<IActionResult> Create(CreateBookReadingCommand cmd) => Ok(await \_mediator.Send(cmd));

// Put, Delete accordingly

}

## Repeat for VideoViewing, WebSearch.

## 11. Summary of Capabilities

The API now:

1. **Returns** lists or single for each entity via Queries (CQRS).
2. **Inserts** new items via Commands.
3. **Updates** entities via Update Commands.
4. **Deletes** entities via Delete Commands.

All using EF Core (SQL Server), Clean Architecture, Repository abstraction, and CQRS via MediatR.

**Additional Notes**

* **EF Core** already implements Repo/UoW; wrapping adds decoupling benefits like easier mocking or swapping data stores [David Veksler's Cheatsheets+2Reddit+2Medium+2](https://www.reddit.com/r/dotnet/comments/gm91hs?utm_source=chatgpt.com)[CodeProject+2Dot Net Tutorials+2LinkedIn+2](https://dotnettutorials.net/lesson/clean-architecture-in-asp-net-core-web-api/?utm_source=chatgpt.com)[Reddit+1Microsoft Learn+1](https://www.reddit.com/r/dotnet/comments/14fs7lo?utm_source=chatgpt.com)[Reddit+9Reddit+9David Veksler's Cheatsheets+9](https://www.reddit.com/r/dotnet/comments/1i07v2u?utm_source=chatgpt.com)[Medium+5Reddit+5David Veksler's Cheatsheets+5](https://www.reddit.com/r/dotnet/comments/14xxtwb?utm_source=chatgpt.com)[Reddit+3atalupadhyay+3Full Bootcamp+3](https://atalupadhyay.wordpress.com/2025/03/31/2486/?utm_source=chatgpt.com).
* **Repository + CQRS** is a common combo ensuring Clean Architecture separation [Medium+15Reddit+15Reddit+15](https://www.reddit.com/r/dotnet/comments/gm91hs?utm_source=chatgpt.com).

**✅ Action Checklist**

1. Create VS projects and references
2. Install required NuGet packages
3. Define Domain entities
4. Implement AppDbContext + IAppDbContext
5. Create IRepository<T> + Repository<T>
6. Configure DI (Infrastructure)
7. Add ConnectionString + Program.cs wiring
8. Run EF migrations
9. Implement CQRS commands/queries and handlers
10. Create Web API controllers

Once done, run the solution and test your CRUD endpoints for BookReadings, VideoViewings, and WebSearches using Swagger or Postman.

Let me know if you'd like full code templates for commands, handlers, DTOs, or wish to integrate paging, filtering, or validation!

Top of Form

Bottom of Form

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

**We use cookies**

Some cookies are essential for this site to function and cannot be turned off. Others are set to help us understand how our service performs and is used, and to support our marketing efforts. Learn more in our [Cookie Policy](https://openai.com/policies/cookie-policy/). For more info or to customize your preferences, tap ‘Manage Cookies’.

# .

# APPENDIX A

## Full code templates for commands, handlers, DTOs and validation using fluent API and also ensure that the entities for Book, YouTubeVideos and Searches all have a foreign key to their parent table "Resources"

Below is a concise yet thorough set of templates—based on Clean Architecture, MediatR/CQRS, FluentValidation, EF Core with a shared "Resources" parent FK structure—for managing Books, YouTube Videos, and Web Searches:

**🔹 1. Domain Entities (StudentTracker.Domain)**

public class Resource

{

public Guid Id { get; set; }

public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

public ICollection<Book>? Books { get; set; }

public ICollection<Video>? Videos { get; set; }

public ICollection<WebSearch>? WebSearches { get; set; }

}

public class Book

{

public Guid Id { get; set; }

public Guid ResourceId { get; set; }

public Resource Resource { get; set; } = default!;

public string Title { get; set; } = default!;

// other properties: Author, Pages, etc.

}

*(similar for Video and WebSearch)*

**🔹 2. DTOs (StudentTracker.Application.DTOs)**

public record BookDto(Guid Id, Guid ResourceId, string Title);

public record VideoDto(Guid Id, Guid ResourceId, string YoutubeId, string Title);

public record SearchDto(Guid Id, Guid ResourceId, string Query, DateTime SearchedAt);

**🔹 3. Application Layer: Commands, Queries, Handlers, Validators**

**🌟 GetAllBooksQuery + Handler:**

public record GetAllBooksQuery() : IRequest<List<BookDto>>;

public class GetAllBooksQueryHandler : IRequestHandler<GetAllBooksQuery, List<BookDto>>

{

private readonly IRepository<Book> \_repo;

public GetAllBooksQueryHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<List<BookDto>> Handle(GetAllBooksQuery q, CancellationToken ct)

{

var list = await \_repo.GetAllAsync();

return list.Select(b => new BookDto(b.Id, b.ResourceId, b.Title)).ToList();

}

}

**🌟 CreateBookCommand + Handler + Validator:**

public record CreateBookCommand(Guid ResourceId, string Title) : IRequest<Guid>;

public class CreateBookCommandHandler : IRequestHandler<CreateBookCommand, Guid>

{

private readonly IRepository<Book> \_repo;

public CreateBookCommandHandler(IRepository<Book> r) => \_repo = r;

public async Task<Guid> Handle(CreateBookCommand c, CancellationToken ct)

{

var book = new Book { Id = Guid.NewGuid(), ResourceId = c.ResourceId, Title = c.Title };

await \_repo.AddAsync(book);

return book.Id;

}

}

public class CreateBookCommandValidator : AbstractValidator<CreateBookCommand>

{

public CreateBookCommandValidator()

{

RuleFor(x => x.ResourceId).NotEmpty();

RuleFor(x => x.Title).NotEmpty().MaximumLength(200);

}

}

**🌟 Update & Delete (similar):**

public record UpdateBookCommand(Guid Id, string Title) : IRequest;

public class UpdateBookCommandHandler : IRequestHandler<UpdateBookCommand>

{ /\* loads, sets Title, \_repo.UpdateAsync \*/ }

public class UpdateBookCommandValidator : AbstractValidator<UpdateBookCommand>

{ RuleFor(x => x.Id).NotEmpty(); RuleFor(x => x.Title).NotEmpty(); }

public record DeleteBookCommand(Guid Id) : IRequest;

public class DeleteBookCommandHandler : IRequestHandler<DeleteBookCommand>

{ /\* calls \_repo.DeleteAsync \*/ }

public class DeleteBookCommandValidator : AbstractValidator<DeleteBookCommand>

{ RuleFor(x => x.Id).NotEmpty(); }

*(Repeat for Video and WebSearch with their own properties)*

**🔹 4. MediatR + FluentValidation Pipeline**

public class ValidationBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>

{

private readonly IEnumerable<IValidator<TRequest>> \_validators;

public ValidationBehavior(IEnumerable<IValidator<TRequest>> v) => \_validators = v;

public async Task<TResponse> Handle(TRequest req, RequestHandlerDelegate<TResponse> next, CancellationToken ct)

{

var failures = \_validators

.Select(v => v.Validate(new ValidationContext<TRequest>(req)))

.SelectMany(r => r.Errors)

.Where(f => f != null)

.ToList();

if (failures.Any()) throw new ValidationException(failures);

return await next();

}

}

Register in DI:

services.AddValidatorsFromAssembly(typeof(CreateBookCommandValidator).Assembly);

services.AddMediatR(cfg => {

cfg.RegisterServicesFromAssembly(...);

cfg.AddOpenBehavior(typeof(ValidationBehavior<,>));

});

[arg-software.medium.com+15medium.com+15github.com+15](https://medium.com/%40sayyedulawwab/a-comprehensive-guide-to-designing-an-api-from-scratch-with-asp-net-162643e55083?utm_source=chatgpt.com)[cheatsheets.davidveksler.com+8code-maze.com+8reddit.com+8](https://code-maze.com/cqrs-mediatr-fluentvalidation/?utm_source=chatgpt.com)

**🔹 5. Infrastructure EF Core Setup**

public class AppDbContext : DbContext, IAppDbContext

{

public DbSet<Resource> Resources { get; set; } = null!;

public DbSet<Book> Books { get; set; } = null!;

// Videos, WebSearches...

public AppDbContext(DbContextOptions opts) : base(opts) { }

protected override void OnModelCreating(ModelBuilder mb)

{

mb.Entity<Book>().HasOne(b => b.Resource).WithMany(r => r.Books).HasForeignKey(b => b.ResourceId);

// same for Video, WebSearch...

}

}

Register:

services.AddDbContext<AppDbContext>(opts => opts.UseSqlServer(conn));

services.AddScoped<IAppDbContext, AppDbContext>();

services.AddScoped(typeof(IRepository<>), typeof(Repository<>));

services.AddInfrastructure();

**🔹 6. Example API Controller (WebApi)**

[ApiController]

[Route("api/books")]

public class BooksController : ControllerBase

{

private readonly IMediator \_mediator;

public BooksController(IMediator m) => \_mediator = m;

[HttpGet] public Task<List<BookDto>> Get() => \_mediator.Send(new GetAllBooksQuery());

[HttpPost] public Task<Guid> Post(CreateBookCommand c) => \_mediator.Send(c);

[HttpPut("{id}")] public Task Put(Guid id, [FromBody] UpdateBookCommand c) => \_mediator.Send(c with { Id = id });

[HttpDelete("{id}")] public Task Delete(Guid id) => \_mediator.Send(new DeleteBookCommand(id));

}

**✅ Summary**

* Parent Resource table holds all resource types.
* Child entities link via ResourceId FK.
* CQRS commands/queries each have DTOs, handlers, and FluentValidation.
* A ValidationBehavior enforces validators automatically [dandoescode.com+2github.com+2arg-software.medium.com+2](https://github.com/BuilesYeison/CleanArchitectureWebApi?utm_source=chatgpt.com)[apriorit.com+7medium.com+7code-maze.com+7](https://medium.com/%40sayyedulawwab/a-comprehensive-guide-to-designing-an-api-from-scratch-with-asp-net-162643e55083?utm_source=chatgpt.com).
* EF Core config ensures proper foreign keys.
* Controllers simply forward HTTP to MediatR.

You can clone and replicate the above for **Video** and **WebSearch**, adjusting props and DTOs accordingly. Let me know if you'd like those fleshed out or an example Repository<T> implementation or Update handler!

Top of Form

Bottom of Form

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

**We use cookies**

Some cookies are essential for this site to function and cannot be turned off. Others are set to help us understand how our service performs and is used, and to support our marketing efforts. Learn more in our [Cookie Policy](https://openai.com/policies/cookie-policy/). For more info or to customize your preferences, tap ‘Manage Cookies’.

# .

# APPENDIX B:

## Methods in the Entities for Progress Calculations

For the following in .NET Core Web API, 1) Book entity class: please add methods to calculate number of pages left to read, percentage of book read, percentage of book left to read

**ChatGPT said:**

Here's how you can enhance your Book entity in a .NET Core Web API domain layer to include methods for:

1. **Calculating pages left to read**
2. **Percentage of the book read**
3. **Percentage of the book left to read**

**✅ Updated Book Entity (in StudentTracker.Domain.Entities)**

public class Book

{

public Guid Id { get; set; }

// Foreign Key to Resource table

public Guid ResourceId { get; set; }

public Resource Resource { get; set; } = default!;

public string Title { get; set; } = string.Empty;

public string Author { get; set; } = string.Empty;

public int TotalPages { get; set; }

public int PagesRead { get; set; }

public DateTime StartedAt { get; set; } = DateTime.UtcNow;

public DateTime? CompletedAt { get; set; }

// ---------------------------

// Utility methods for reading progress

// ---------------------------

public int PagesLeftToRead()

{

return Math.Max(0, TotalPages - PagesRead);

}

public double PercentageRead()

{

if (TotalPages <= 0) return 0;

return Math.Round((double)PagesRead / TotalPages \* 100, 2);

}

public double PercentageLeftToRead()

{

if (TotalPages <= 0) return 0;

return Math.Round((double)(TotalPages - PagesRead) / TotalPages \* 100, 2);

}

}

**🔎 Example Outputs**

Given:

var book = new Book { TotalPages = 300, PagesRead = 90 };

* book.PagesLeftToRead() → 210
* book.PercentageRead() → 30.0
* book.PercentageLeftToRead() → 70.0

**🛡️ Notes:**

* The methods **guard against divide-by-zero** (when TotalPages == 0).
* Math.Round(..., 2) keeps percentages clean for display.
* These methods are domain logic and fit well within your **Domain Model** in Clean Architecture.

Would you also like a DTO projection for this (i.e. returning progress info via API), or should we include the logic in a ViewModel/DTO mapper for API output?

Get smarter responses, upload files and images, and more.

Top of Form

Bottom of Form

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

**We use cookies**

Some cookies are essential for this site to function and cannot be turned off. Others are set to help us understand how our service performs and is used, and to support our marketing efforts. Learn more in our [Cookie Policy](https://openai.com/policies/cookie-policy/). For more info or to customize your preferences, tap ‘Manage Cookies’.

# .

# APPENDIX C

**My Prompt:**

## Code for queries and commands, the IRepositories, Repositories Classes, Controllers for Get, Post, and the updates and deletes with explanation of how they all work together, along with any Entity Framework Core and LINQ code.

**ChatGPT said:**

Below is a fully fleshed-out implementation of CQRS (Commands & Queries), the IRepository<T> abstraction, repository implementation, and Web API controllers for Book, Video, and WebSearch entities—using Entity Framework Core, LINQ, and Clean Architecture principles.

1. IRepository & Repository Implementation

IRepository<T> (Application Layer)

public interface IRepository<T> where T : class

{

Task<IEnumerable<T>> GetAllAsync();

Task<T?> GetByIdAsync(Guid id);

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(Guid id);

}

Repository<T> (Infrastructure → Repositories)

public class Repository<T> : IRepository<T> where T : class

{

private readonly AppDbContext \_ctx;

public Repository(AppDbContext ctx) => \_ctx = ctx;

public async Task<IEnumerable<T>> GetAllAsync() => await \_ctx.Set<T>().ToListAsync();

public async Task<T?> GetByIdAsync(Guid id) => await \_ctx.Set<T>().FindAsync(id);

public async Task AddAsync(T entity)

{

await \_ctx.Set<T>().AddAsync(entity);

await \_ctx.SaveChangesAsync();

}

public async Task UpdateAsync(T entity)

{

\_ctx.Set<T>().Update(entity);

await \_ctx.SaveChangesAsync();

}

public async Task DeleteAsync(Guid id)

{

var entity = await GetByIdAsync(id);

if (entity != null)

{

\_ctx.Set<T>().Remove(entity);

await \_ctx.SaveChangesAsync();

}

}

}

2. CQRS — Queries and Handlers (using MediatR)

We'll focus on Book; duplicating for Video and WebSearch is similar.

GetAllBooksQuery

public record GetAllBooksQuery() : IRequest<IEnumerable<BookDto>>;

public class GetAllBooksQueryHandler : IRequestHandler<GetAllBooksQuery, IEnumerable<BookDto>>

{

private readonly IRepository<Book> \_repo;

public GetAllBooksQueryHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<IEnumerable<BookDto>> Handle(GetAllBooksQuery q, CancellationToken ct)

{

return (await \_repo.GetAllAsync())

.Select(b => new BookDto(b.Id, b.ResourceId, b.Title, b.Author, b.TotalPages, b.PagesRead))

.ToList();

}

}

GetBookByIdQuery

public record GetBookByIdQuery(Guid Id) : IRequest<BookDto?>;

public class GetBookByIdQueryHandler : IRequestHandler<GetBookByIdQuery, BookDto?>

{

private readonly IRepository<Book> \_repo;

public GetBookByIdQueryHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<BookDto?> Handle(GetBookByIdQuery q, CancellationToken ct)

{

var b = await \_repo.GetByIdAsync(q.Id);

if (b == null) return null;

return new BookDto(b.Id, b.ResourceId, b.Title, b.Author, b.TotalPages, b.PagesRead);

}

}

3. CQRS — Commands and Handlers

CreateBookCommand

public record CreateBookCommand(Guid ResourceId, string Title, string Author, int TotalPages) : IRequest<Guid>;

public class CreateBookCommandHandler : IRequestHandler<CreateBookCommand, Guid>

{

private readonly IRepository<Book> \_repo;

public CreateBookCommandHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<Guid> Handle(CreateBookCommand c, CancellationToken ct)

{

var id = Guid.NewGuid();

await \_repo.AddAsync(new Book

{

Id = id,

ResourceId = c.ResourceId,

Title = c.Title,

Author = c.Author,

TotalPages = c.TotalPages,

PagesRead = 0 // initially zero

});

return id;

}

}

UpdateBookCommand

public record UpdateBookCommand(Guid Id, string Title, string Author, int TotalPages, int PagesRead) : IRequest;

public class UpdateBookCommandHandler : IRequestHandler<UpdateBookCommand>

{

private readonly IRepository<Book> \_repo;

public UpdateBookCommandHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<Unit> Handle(UpdateBookCommand c, CancellationToken ct)

{

var book = await \_repo.GetByIdAsync(c.Id);

if (book == null) throw new KeyNotFoundException($"Book {c.Id} not found.");

book.Title = c.Title;

book.Author = c.Author;

book.TotalPages = c.TotalPages;

book.PagesRead = c.PagesRead;

await \_repo.UpdateAsync(book);

return Unit.Value;

}

}

DeleteBookCommand

public record DeleteBookCommand(Guid Id) : IRequest;

public class DeleteBookCommandHandler : IRequestHandler<DeleteBookCommand>

{

private readonly IRepository<Book> \_repo;

public DeleteBookCommandHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<Unit> Handle(DeleteBookCommand c, CancellationToken ct)

{

await \_repo.DeleteAsync(c.Id);

return Unit.Value;

}

}

4. DTOs (Data Transfer Objects)

public record BookDto(Guid Id, Guid ResourceId, string Title, string Author, int TotalPages, int PagesRead);

Add additional fields like pages left or percentages if you want.

5. API Controllers

BooksController

[ApiController]

[Route("api/[controller]")]

public class BooksController : ControllerBase

{

private readonly IMediator \_mediator;

public BooksController(IMediator mediator) => \_mediator = mediator;

[HttpGet]

public async Task<ActionResult<IEnumerable<BookDto>>> Get()

{

var result = await \_mediator.Send(new GetAllBooksQuery());

return Ok(result);

}

[HttpGet("{id:guid}")]

public async Task<ActionResult<BookDto>> Get(Guid id)

{

var book = await \_mediator.Send(new GetBookByIdQuery(id));

if (book == null) return NotFound();

return Ok(book);

}

[HttpPost]

public async Task<ActionResult<Guid>> Post(CreateBookCommand cmd)

{

var id = await \_mediator.Send(cmd);

return CreatedAtAction(nameof(Get), new { id }, id);

}

[HttpPut("{id:guid}")]

public async Task<IActionResult> Put(Guid id, UpdateBookCommand cmd)

{

if (id != cmd.Id) return BadRequest("Mismatched ID");

await \_mediator.Send(cmd);

return NoContent();

}

[HttpDelete("{id:guid}")]

public async Task<IActionResult> Delete(Guid id)

{

await \_mediator.Send(new DeleteBookCommand(id));

return NoContent();

}

}

6. Connecting Everything

1. WebApi Layer registers:
   * MediatR handlers
   * IRepository<> → Repository<>
   * EF Core AppDbContext → SQL Server via AddDbContext
2. MediatR routes each IRequest (Command/Query) to its corresponding handler.
3. Repositories handle data access logic (SQL DB + EF Core + LINQ).
4. Controllers simply delegate HTTP requests to MediatR commands/queries.
5. Commands mutate state; queries read data; everything uses asynchronous methods with EF Core.

Repeating for Other Entities

* Replace Book with Video or WebSearch.
* Adjust DTOs, queries, commands with relevant properties.
* Controller endpoints use the same patterns.

Summary

| Layer | Role |
| --- | --- |
| Repository | Encapsulates EF Core + LINQ logic for CRUD |
| Commands | Mutate state (Create, Update, Delete) via MediatR |
| Queries | Read data and map to DTOs |
| Handlers | Bridge between commands/queries and repository |
| Controller | HTTP endpoint → command/query via MediatR |
| DI Setup | Wires EF Core, repositories, and MediatR in Startup/Program.cs |

# APPENDIX D

## for the following in .NET Core Web API, 1) Book entity class: please add methods to calculate number of pages left to read, percentage of book read, percentage of book left to read

**ChatGPT said:**

Here's how you can enhance your Book entity in a .NET Core Web API domain layer to include methods for:

1. **Calculating pages left to read**
2. **Percentage of the book read**
3. **Percentage of the book left to read**

**✅ Updated Book Entity (in StudentTracker.Domain.Entities)**

public class Book

{

public Guid Id { get; set; }

// Foreign Key to Resource table

public Guid ResourceId { get; set; }

public Resource Resource { get; set; } = default!;

public string Title { get; set; } = string.Empty;

public string Author { get; set; } = string.Empty;

public int TotalPages { get; set; }

public int PagesRead { get; set; }

public DateTime StartedAt { get; set; } = DateTime.UtcNow;

public DateTime? CompletedAt { get; set; }

// ---------------------------

// Utility methods for reading progress

// ---------------------------

public int PagesLeftToRead()

{

return Math.Max(0, TotalPages - PagesRead);

}

public double PercentageRead()

{

if (TotalPages <= 0) return 0;

return Math.Round((double)PagesRead / TotalPages \* 100, 2);

}

public double PercentageLeftToRead()

{

if (TotalPages <= 0) return 0;

return Math.Round((double)(TotalPages - PagesRead) / TotalPages \* 100, 2);

}

}

**🔎 Example Outputs**

Given:

var book = new Book { TotalPages = 300, PagesRead = 90 };

* book.PagesLeftToRead() → 210
* book.PercentageRead() → 30.0
* book.PercentageLeftToRead() → 70.0

**🛡️ Notes:**

* The methods **guard against divide-by-zero** (when TotalPages == 0).
* Math.Round(..., 2) keeps percentages clean for display.
* These methods are domain logic and fit well within your **Domain Model** in Clean Architecture.

Would you also like a DTO projection for this (i.e. returning progress info via API), or should we include the logic in a ViewModel/DTO mapper for API output?

**You said:**

# APPENDIX E

## Please flesh out the code for queries and commands for CQRS, the IRepositories code, the implementation of the Repositories, and the Controllers for Get, Post, and the updates and deletes, and provide an explanation of how they all work together, along with any Entity Framework Core and LINQ code.

**ChatGPT said:**

Below is a fully fleshed-out implementation of CQRS (Commands & Queries), the IRepository<T> abstraction, repository implementation, and Web API controllers for Book, Video, and WebSearch entities—using Entity Framework Core, LINQ, and Clean Architecture principles.

**🏷️ 1. IRepository & Repository Implementation**

**IRepository<T> (Application Layer)**

public interface IRepository<T> where T : class

{

Task<IEnumerable<T>> GetAllAsync();

Task<T?> GetByIdAsync(Guid id);

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(Guid id);

}

**Repository<T> (Infrastructure → Repositories)**

public class Repository<T> : IRepository<T> where T : class

{

private readonly AppDbContext \_ctx;

public Repository(AppDbContext ctx) => \_ctx = ctx;

public async Task<IEnumerable<T>> GetAllAsync() => await \_ctx.Set<T>().ToListAsync();

public async Task<T?> GetByIdAsync(Guid id) => await \_ctx.Set<T>().FindAsync(id);

public async Task AddAsync(T entity)

{

await \_ctx.Set<T>().AddAsync(entity);

await \_ctx.SaveChangesAsync();

}

public async Task UpdateAsync(T entity)

{

\_ctx.Set<T>().Update(entity);

await \_ctx.SaveChangesAsync();

}

public async Task DeleteAsync(Guid id)

{

var entity = await GetByIdAsync(id);

if (entity != null)

{

\_ctx.Set<T>().Remove(entity);

await \_ctx.SaveChangesAsync();

}

}

}

**📡 2. CQRS — Queries and Handlers (using MediatR)**

We'll focus on Book; duplicating for Video and WebSearch is similar.

**GetAllBooksQuery**

public record GetAllBooksQuery() : IRequest<IEnumerable<BookDto>>;

public class GetAllBooksQueryHandler : IRequestHandler<GetAllBooksQuery, IEnumerable<BookDto>>

{

private readonly IRepository<Book> \_repo;

public GetAllBooksQueryHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<IEnumerable<BookDto>> Handle(GetAllBooksQuery q, CancellationToken ct)

{

return (await \_repo.GetAllAsync())

.Select(b => new BookDto(b.Id, b.ResourceId, b.Title, b.Author, b.TotalPages, b.PagesRead))

.ToList();

}

}

**GetBookByIdQuery**

public record GetBookByIdQuery(Guid Id) : IRequest<BookDto?>;

public class GetBookByIdQueryHandler : IRequestHandler<GetBookByIdQuery, BookDto?>

{

private readonly IRepository<Book> \_repo;

public GetBookByIdQueryHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<BookDto?> Handle(GetBookByIdQuery q, CancellationToken ct)

{

var b = await \_repo.GetByIdAsync(q.Id);

if (b == null) return null;

return new BookDto(b.Id, b.ResourceId, b.Title, b.Author, b.TotalPages, b.PagesRead);

}

}

**✍️ 3. CQRS — Commands and Handlers**

**CreateBookCommand**

public record CreateBookCommand(Guid ResourceId, string Title, string Author, int TotalPages) : IRequest<Guid>;

public class CreateBookCommandHandler : IRequestHandler<CreateBookCommand, Guid>

{

private readonly IRepository<Book> \_repo;

public CreateBookCommandHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<Guid> Handle(CreateBookCommand c, CancellationToken ct)

{

var id = Guid.NewGuid();

await \_repo.AddAsync(new Book

{

Id = id,

ResourceId = c.ResourceId,

Title = c.Title,

Author = c.Author,

TotalPages = c.TotalPages,

PagesRead = 0 // initially zero

});

return id;

}

}

**UpdateBookCommand**

public record UpdateBookCommand(Guid Id, string Title, string Author, int TotalPages, int PagesRead) : IRequest;

public class UpdateBookCommandHandler : IRequestHandler<UpdateBookCommand>

{

private readonly IRepository<Book> \_repo;

public UpdateBookCommandHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<Unit> Handle(UpdateBookCommand c, CancellationToken ct)

{

var book = await \_repo.GetByIdAsync(c.Id);

if (book == null) throw new KeyNotFoundException($"Book {c.Id} not found.");

book.Title = c.Title;

book.Author = c.Author;

book.TotalPages = c.TotalPages;

book.PagesRead = c.PagesRead;

await \_repo.UpdateAsync(book);

return Unit.Value;

}

}

**DeleteBookCommand**

public record DeleteBookCommand(Guid Id) : IRequest;

public class DeleteBookCommandHandler : IRequestHandler<DeleteBookCommand>

{

private readonly IRepository<Book> \_repo;

public DeleteBookCommandHandler(IRepository<Book> repo) => \_repo = repo;

public async Task<Unit> Handle(DeleteBookCommand c, CancellationToken ct)

{

await \_repo.DeleteAsync(c.Id);

return Unit.Value;

}

}

**🧾 4. DTOs (Data Transfer Objects)**

public record BookDto(Guid Id, Guid ResourceId, string Title, string Author, int TotalPages, int PagesRead);

Add additional fields like pages left or percentages if you want.

**📐 5. API Controllers**

**BooksController**

[ApiController]

[Route("api/[controller]")]

public class BooksController : ControllerBase

{

private readonly IMediator \_mediator;

public BooksController(IMediator mediator) => \_mediator = mediator;

[HttpGet]

public async Task<ActionResult<IEnumerable<BookDto>>> Get()

{

var result = await \_mediator.Send(new GetAllBooksQuery());

return Ok(result);

}

[HttpGet("{id:guid}")]

public async Task<ActionResult<BookDto>> Get(Guid id)

{

var book = await \_mediator.Send(new GetBookByIdQuery(id));

if (book == null) return NotFound();

return Ok(book);

}

[HttpPost]

public async Task<ActionResult<Guid>> Post(CreateBookCommand cmd)

{

var id = await \_mediator.Send(cmd);

return CreatedAtAction(nameof(Get), new { id }, id);

}

[HttpPut("{id:guid}")]

public async Task<IActionResult> Put(Guid id, UpdateBookCommand cmd)

{

if (id != cmd.Id) return BadRequest("Mismatched ID");

await \_mediator.Send(cmd);

return NoContent();

}

[HttpDelete("{id:guid}")]

public async Task<IActionResult> Delete(Guid id)

{

await \_mediator.Send(new DeleteBookCommand(id));

return NoContent();

}

}

**⚙️ 6. Connecting Everything**

1. **WebApi Layer** registers:
   * MediatR handlers
   * IRepository<> → Repository<>
   * EF Core AppDbContext → SQL Server via AddDbContext
2. MediatR routes each IRequest (Command/Query) to its corresponding handler.
3. **Repositories** handle data access logic (SQL DB + EF Core + LINQ).
4. **Controllers** simply delegate HTTP requests to MediatR commands/queries.
5. Commands mutate state; queries read data; everything uses asynchronous methods with EF Core.

**🔁 Repeating for Other Entities**

* Replace Book with Video or WebSearch.
* Adjust DTOs, queries, commands with relevant properties.
* Controller endpoints use the same patterns.

**✅ Summary**

| **Layer** | **Role** |
| --- | --- |
| **Repository** | Encapsulates EF Core + LINQ logic for CRUD |
| **Commands** | Mutate state (Create, Update, Delete) via MediatR |
| **Queries** | Read data and map to DTOs |
| **Handlers** | Bridge between commands/queries and repository |
| **Controller** | HTTP endpoint → command/query via MediatR |
| **DI Setup** | Wires EF Core, repositories, and MediatR in Startup/Program.cs |

This structure cleanly separates concerns, ensuring each component focuses on a single responsibility. EF Core is used with LINQ inside repositories, and CQRS + MediatR keep commands and queries separate. Let me know if you’d like the same code for Video and WebSearch, or if you'd like to integrate FluentValidation on commands!

# APPENDIX F

## Specific IRepository interfaces and Repository implementations for book, YouTubeVideo and InternetSearches.

**ChatGPT said:**

Yes — you can **either** use:

* ✅ A **generic IRepository<T>** for code reuse
* ✅ Or create **specific repositories like IBookRepository, IVideoRepository, etc.** for more control

Let’s walk through:

**✅ When to Use a Generic IRepository<T>**

Use a generic repository when:

* All entities have standard CRUD needs (Create, Read, Update, Delete)
* You don’t need entity-specific queries
* You want reusable and consistent data access logic

**Example:**

public interface IRepository<T> where T : class

{

Task<IEnumerable<T>> GetAllAsync();

Task<T?> GetByIdAsync(Guid id);

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(Guid id);

}

Then inject and use:

private readonly IRepository<Book> \_bookRepo;

**When to Use Specific Repositories (IBookRepository, etc.)**

Use this when:

* You need **custom queries** per entity
* Each entity may need **special methods** not shared with others
* You want stricter domain boundaries and better encapsulation

## 1. IBookRepository Interface

public interface IBookRepository

{

Task<IEnumerable<Book>> GetAllAsync();

Task<Book?> GetByIdAsync(Guid id);

Task AddAsync(Book book);

Task UpdateAsync(Book book);

Task DeleteAsync(Guid id);

// Custom method

Task<IEnumerable<Book>> GetBooksByAuthorAsync(string author);

}

## 2. BookRepository Implementation

public class BookRepository : IBookRepository

{

private readonly AppDbContext \_ctx;

public BookRepository(AppDbContext ctx) => \_ctx = ctx;

public async Task<IEnumerable<Book>> GetAllAsync()

=> await \_ctx.Books.ToListAsync();

public async Task<Book?> GetByIdAsync(Guid id)

=> await \_ctx.Books.FindAsync(id);

public async Task AddAsync(Book book)

{

await \_ctx.Books.AddAsync(book);

await \_ctx.SaveChangesAsync();

}

public async Task UpdateAsync(Book book)

{

\_ctx.Books.Update(book);

await \_ctx.SaveChangesAsync();

}

public async Task DeleteAsync(Guid id)

{

var book = await GetByIdAsync(id);

if (book != null)

{

\_ctx.Books.Remove(book);

await \_ctx.SaveChangesAsync();

}

}

public async Task<IEnumerable<Book>> GetBooksByAuthorAsync(string author)

{

return await \_ctx.Books

.Where(b => b.Author.ToLower() == author.ToLower())

.ToListAsync();

}

}

## 3. Create Repositories for Other Entities

**IYouTubeVideoRepository**

public interface IYouTubeVideoRepository

{

Task<IEnumerable<YouTubeVideo>> GetAllAsync();

Task<YouTubeVideo?> GetByIdAsync(Guid id);

Task AddAsync(YouTubeVideo video);

Task UpdateAsync(YouTubeVideo video);

Task DeleteAsync(Guid id);

Task<IEnumerable<YouTubeVideo>> GetByChannelAsync(string channelName);

}

**IInternetSearchRepository**

public interface IInternetSearchRepository

{

Task<IEnumerable<WebSearch>> GetAllAsync();

Task<WebSearch?> GetByIdAsync(Guid id);

Task AddAsync(WebSearch search);

Task UpdateAsync(WebSearch search);

Task DeleteAsync(Guid id);

Task<IEnumerable<WebSearch>> GetByKeywordAsync(string keyword);

}

**💡 Which is Better?**

| **Use Case** | **Recommendation** |
| --- | --- |
| Simple CRUD across many entities | ✅ Generic Repository |
| Complex or entity-specific queries | ✅ Specific Repository |
| Strict domain boundaries (DDD) | ✅ Specific Repository |
| Testability and separation of concerns | ✅ Specific Repository |

**Best Practice**:  
🔁 Use **Generic Repository** as a base, then extend with **specific interfaces** when needed.

public interface IBookRepository : IRepository<Book>

{

Task<IEnumerable<Book>> GetBooksByAuthorAsync(string author);

}

Then your BookRepository can inherit from a base Repository<T> and add specifics.

**✅ Final Tip – Hybrid Approach (Recommended)**

1. Create a base generic Repository<T>
2. Extend with entity-specific IBookRepository, IVideoRepository, etc.
3. Inject specific interfaces where logic demands it

Would you like me to generate the hybrid setup now (shared generic + entity-specific extensions)?

# APPENDIX G

## Can you create **specific** IRepository interfaces and Repository implementations for book, YouTubeVideo and InternetSearches. Is it better to create a generic IRepository though?

**ChatGPT said:**

Yes — you can **either** use:

* ✅ A **generic IRepository<T>** for code reuse
* ✅ Or create **specific repositories like IBookRepository, IVideoRepository, etc.** for more control

Let’s walk through:

**✅ When to Use a Generic IRepository<T>**

Use a generic repository when:

* All entities have standard CRUD needs (Create, Read, Update, Delete)
* You don’t need entity-specific queries
* You want reusable and consistent data access logic

**Example:**

public interface IRepository<T> where T : class

{

Task<IEnumerable<T>> GetAllAsync();

Task<T?> GetByIdAsync(Guid id);

Task AddAsync(T entity);

Task UpdateAsync(T entity);

Task DeleteAsync(Guid id);

}

Then inject and use:

private readonly IRepository<Book> \_bookRepo;

**🔧 When to Use Specific Repositories (IBookRepository, etc.)**

Use this when:

* You need **custom queries** per entity
* Each entity may need **special methods** not shared with others
* You want stricter domain boundaries and better encapsulation

**✅ 1. IBookRepository Interface**

public interface IBookRepository

{

Task<IEnumerable<Book>> GetAllAsync();

Task<Book?> GetByIdAsync(Guid id);

Task AddAsync(Book book);

Task UpdateAsync(Book book);

Task DeleteAsync(Guid id);

// Custom method

Task<IEnumerable<Book>> GetBooksByAuthorAsync(string author);

}

**✅ 2. BookRepository Implementation**

public class BookRepository : IBookRepository

{

private readonly AppDbContext \_ctx;

public BookRepository(AppDbContext ctx) => \_ctx = ctx;

public async Task<IEnumerable<Book>> GetAllAsync()

=> await \_ctx.Books.ToListAsync();

public async Task<Book?> GetByIdAsync(Guid id)

=> await \_ctx.Books.FindAsync(id);

public async Task AddAsync(Book book)

{

await \_ctx.Books.AddAsync(book);

await \_ctx.SaveChangesAsync();

}

public async Task UpdateAsync(Book book)

{

\_ctx.Books.Update(book);

await \_ctx.SaveChangesAsync();

}

public async Task DeleteAsync(Guid id)

{

var book = await GetByIdAsync(id);

if (book != null)

{

\_ctx.Books.Remove(book);

await \_ctx.SaveChangesAsync();

}

}

public async Task<IEnumerable<Book>> GetBooksByAuthorAsync(string author)

{

return await \_ctx.Books

.Where(b => b.Author.ToLower() == author.ToLower())

.ToListAsync();

}

}

**🔁 3. Create Repositories for Other Entities**

**IYouTubeVideoRepository**

public interface IYouTubeVideoRepository

{

Task<IEnumerable<YouTubeVideo>> GetAllAsync();

Task<YouTubeVideo?> GetByIdAsync(Guid id);

Task AddAsync(YouTubeVideo video);

Task UpdateAsync(YouTubeVideo video);

Task DeleteAsync(Guid id);

Task<IEnumerable<YouTubeVideo>> GetByChannelAsync(string channelName);

}

**IInternetSearchRepository**

public interface IInternetSearchRepository

{

Task<IEnumerable<WebSearch>> GetAllAsync();

Task<WebSearch?> GetByIdAsync(Guid id);

Task AddAsync(WebSearch search);

Task UpdateAsync(WebSearch search);

Task DeleteAsync(Guid id);

Task<IEnumerable<WebSearch>> GetByKeywordAsync(string keyword);

}

**💡 Which is Better?**

| **Use Case** | **Recommendation** |
| --- | --- |
| Simple CRUD across many entities | ✅ Generic Repository |
| Complex or entity-specific queries | ✅ Specific Repository |
| Strict domain boundaries (DDD) | ✅ Specific Repository |
| Testability and separation of concerns | ✅ Specific Repository |

**Best Practice**:  
🔁 Use **Generic Repository** as a base, then extend with **specific interfaces** when needed.

public interface IBookRepository : IRepository<Book>

{

Task<IEnumerable<Book>> GetBooksByAuthorAsync(string author);

}

Then your BookRepository can inherit from a base Repository<T> and add specifics.

**✅ Final Tip – Hybrid Approach (Recommended)**

1. Create a base generic Repository<T>
2. Extend with entity-specific IBookRepository, IVideoRepository, etc.
3. Inject specific interfaces where logic demands it

# APPENDIX H

## Show the code and how a **specific** Book entity, IBookRepository, BookRepository Implementation, CQRS query, CQRS Command, BookDTO, and BookController all work together, including the sequence of code execution.

**A** full breakdown—code and explanation—of how the following components work together in a Clean Architecture .NET Core Web API using **CQRS**, **Entity Framework Core**, and **MediatR**:

**📚 Scenario:**

A Book entity is being **created**, **queried**, **updated**, or **deleted** through an API controller. We’ll trace what happens **end-to-end**, step by step.

**✅ 1. Book Entity (Domain Layer)**

public class Book

{

public Guid Id { get; set; }

public Guid ResourceId { get; set; } // FK

public string Title { get; set; } = string.Empty;

public string Author { get; set; } = string.Empty;

public int TotalPages { get; set; }

public int PagesRead { get; set; }

// Domain logic

public int PagesLeftToRead() => Math.Max(0, TotalPages - PagesRead);

public double PercentageRead() => TotalPages == 0 ? 0 : Math.Round((double)PagesRead / TotalPages \* 100, 2);

}

**✅ 2. BookDto (Application Layer)**

public record BookDto(Guid Id, Guid ResourceId, string Title, string Author, int TotalPages, int PagesRead);

Used for **output** to the client, not for internal domain logic.

**✅ 3. IBookRepository Interface**

public interface IBookRepository

{

Task<IEnumerable<Book>> GetAllAsync();

Task<Book?> GetByIdAsync(Guid id);

Task AddAsync(Book book);

Task UpdateAsync(Book book);

Task DeleteAsync(Guid id);

}

**✅ 4. BookRepository Implementation (Infrastructure Layer)**

public class BookRepository : IBookRepository

{

private readonly AppDbContext \_context;

public BookRepository(AppDbContext context) => \_context = context;

public async Task<IEnumerable<Book>> GetAllAsync() => await \_context.Books.ToListAsync();

public async Task<Book?> GetByIdAsync(Guid id) => await \_context.Books.FindAsync(id);

public async Task AddAsync(Book book)

{

await \_context.Books.AddAsync(book);

await \_context.SaveChangesAsync();

}

public async Task UpdateAsync(Book book)

{

\_context.Books.Update(book);

await \_context.SaveChangesAsync();

}

public async Task DeleteAsync(Guid id)

{

var book = await GetByIdAsync(id);

if (book != null)

{

\_context.Books.Remove(book);

await \_context.SaveChangesAsync();

}

}

}

**✅ 5. CQRS: Command & Handler**

**📥 CreateBookCommand**

public record CreateBookCommand(Guid ResourceId, string Title, string Author, int TotalPages) : IRequest<Guid>;

public class CreateBookCommandHandler : IRequestHandler<CreateBookCommand, Guid>

{

private readonly IBookRepository \_repo;

public CreateBookCommandHandler(IBookRepository repo) => \_repo = repo;

public async Task<Guid> Handle(CreateBookCommand request, CancellationToken ct)

{

var book = new Book

{

Id = Guid.NewGuid(),

ResourceId = request.ResourceId,

Title = request.Title,

Author = request.Author,

TotalPages = request.TotalPages,

PagesRead = 0

};

await \_repo.AddAsync(book);

return book.Id;

}

}

**✅ 6. CQRS: Query & Handler**

**📤 GetAllBooksQuery**

public record GetAllBooksQuery() : IRequest<IEnumerable<BookDto>>;

public class GetAllBooksQueryHandler : IRequestHandler<GetAllBooksQuery, IEnumerable<BookDto>>

{

private readonly IBookRepository \_repo;

public GetAllBooksQueryHandler(IBookRepository repo) => \_repo = repo;

public async Task<IEnumerable<BookDto>> Handle(GetAllBooksQuery request, CancellationToken ct)

{

var books = await \_repo.GetAllAsync();

return books.Select(b => new BookDto(b.Id, b.ResourceId, b.Title, b.Author, b.TotalPages, b.PagesRead));

}

}

**✅ 7. BooksController (Web API Layer)**

[ApiController]

[Route("api/[controller]")]

public class BooksController : ControllerBase

{

private readonly IMediator \_mediator;

public BooksController(IMediator mediator) => \_mediator = mediator;

[HttpGet]

public async Task<ActionResult<IEnumerable<BookDto>>> Get()

=> Ok(await \_mediator.Send(new GetAllBooksQuery()));

[HttpPost]

public async Task<ActionResult<Guid>> Post(CreateBookCommand cmd)

{

var id = await \_mediator.Send(cmd);

return CreatedAtAction(nameof(Get), new { id }, id);

}

}

**🔁 How They All Work Together (Step-by-Step)**

**🧩 Scenario: User sends POST /api/books**

**1. Web API Controller:**

* Receives CreateBookCommand via [HttpPost]
* Passes command to MediatR.Send()

**2. MediatR Pipeline:**

* Finds CreateBookCommandHandler
* Executes Handle()

**3. Command Handler:**

* Creates a Book entity
* Calls \_repo.AddAsync(book)

**4. Repository:**

* Uses AppDbContext (EF Core) to add the entity
* Saves to SQL Server database via SaveChangesAsync

**5**

Top of Form

Bottom of Form

Top of Form

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

**We use cookies**

Some cookies are essential for this site to function and cannot be turned off. Others are set to help us understand how our service performs and is used, and to support our marketing efforts. Learn more in our [Cookie Policy](https://openai.com/policies/cookie-policy/). For more info or to customize your preferences, tap ‘Manage Cookies’.

ChatGPT is still generating a response...

# APPENDIX I

## MediatR Pipeline – What Is It?

MediatR is a library in .NET used to implement the **Mediator pattern**, where requests (like commands or queries) are sent to a central "mediator" rather than directly invoking services or handlers.

A **MediatR pipeline** means:

1. A **request** (e.g., CreateBookCommand) is sent via IMediator.Send(...).
2. MediatR locates the **handler** for that request (e.g., CreateBookCommandHandler).
3. It executes the handler’s Handle(...) method.
4. **Optionally**, MediatR can apply pipeline behaviors (e.g., logging, validation, performance monitoring) **before or after** the handler runs.

**🧩 How This Code Uses the MediatR Pipeline**

Here’s the flow:

**a. Finds CreateBookCommandHandler**

* The command CreateBookCommand implements IRequest<Guid>, which makes it a **MediatR request**.
* MediatR uses **reflection and DI (dependency injection)** to find a class that implements IRequestHandler<CreateBookCommand, Guid>.
* That handler is CreateBookCommandHandler.

✅ **This step is part of the MediatR pipeline** — MediatR dynamically **finds the right handler** for the command.

**b. Executes Handle()**

* Once the handler is found, MediatR **calls** its Handle() method with the command as the parameter.
* In this method, the actual business logic — creating and saving a book — is performed.
* The result (the new book’s ID) is returned.

✅ **This is the handler execution** step in the pipeline — core to MediatR’s role in decoupling the sender from the handler.

**🧠 Why It’s Called MediatR Pipeline (Even Without Custom Behaviors)**

Even if you **don’t add custom pipeline behaviors**, the **default MediatR flow of request → handler → response** is still considered a pipeline. You can extend this pipeline with:

* Logging
* Validation
* Authorization
* Retry policies
* Caching

Each of these can plug in between the request and the handler using IPipelineBehavior<TRequest, TResponse>.

**📦 CQRS Tie-In**

This code follows the **Command** part of CQRS:

* CreateBookCommand = **Command** (intention to change state)
* CreateBookCommandHandler = **Handler** (contains business logic to handle the command)

MediatR facilitates this decoupling cleanly — hence its popularity in CQRS-based architectures.

**✅ Summary: Why It’s Called MediatR Pipeline**

It’s labeled **MediatR Pipeline** because:

* The command is sent via MediatR.
* MediatR finds and invokes the matching handler.
* This flow (send → resolve → handle → return) constitutes the **MediatR pipeline**, even if no custom behaviors are used.

# APPENDIX J

## Required to install **MediatR** from NPM to use the IRequest interface in CommandHandler code

## What is IRequest,

and what would be the c# code for it, in a .NET Core Web API project?

public record GetBookByIdQuery(Guid Id) : IRequest<BookDto?>;

**ChatGPT said:**

In your code:

public record GetBookByIdQuery(Guid Id) : IRequest<BookDto?>;

IRequest<T> is an interface from **MediatR**, a popular library used in .NET for implementing the **CQRS (Command Query Responsibility Segregation)** and **Mediator** design patterns.

* IRequest<TResponse> represents a request that returns a response of type TResponse.
* In this case, GetBookByIdQuery is a **query** (read operation) that expects to return a BookDto?.

So, IRequest<BookDto?> means this request will return a BookDto or null.

## Where does IRequest come from?

From the **MediatR** NuGet package.

To use it, install MediatR in your project:

dotnet add package MediatR

dotnet add package MediatR.Extensions.Microsoft.DependencyInjection

## How to define and use IRequest<BookDto?>

### 1.Define the query ( e.g. - public record GetBookByIdQuery(Guid Id) : **IRequest**<BookDto?>; )

public record GetBookByIdQuery(Guid Id) : IRequest<BookDto?>;

### 2.Define the handler:

You need to implement IRequestHandler<GetBookByIdQuery, BookDto?>:

public class GetBookByIdQueryHandler : IRequestHandler<GetBookByIdQuery, BookDto?>

{

private readonly IBookRepository \_bookRepository;

public GetBookByIdQueryHandler(IBookRepository bookRepository)

{

\_bookRepository = bookRepository;

}

public async Task<BookDto?> Handle(GetBookByIdQuery request, CancellationToken cancellationToken)

{

var book = await \_bookRepository.GetByIdAsync(request.Id);

return book == null ? null : new BookDto

{

Id = book.Id,

Title = book.Title,

Author = book.Author

};

}

}

### 3.Register MediatR in Startup.cs or Program.cs (depending on the .NET version):

For .NET 6+ (minimal hosting model):

builder.Services.AddMediatR(cfg => cfg.RegisterServicesFromAssemblyContaining<GetBookByIdQuery>());

### 4.Use the query in your controller:

[ApiController]

[Route("api/[controller]")]

public class BooksController : ControllerBase

{

private readonly IMediator \_mediator;

public BooksController(IMediator mediator)

{

\_mediator = mediator;

}

[HttpGet("{id}")]

public async Task<IActionResult> GetById(Guid id)

{

var result = await \_mediator.Send(new GetBookByIdQuery(id));

return result == null ? NotFound() : Ok(result);

}

}

## 5.Summary

* IRequest<T> comes from the **MediatR** library.
* It represents a request in the **CQRS** pattern that expects a response of type T.
* In your example, it's used to retrieve a book by ID.
* You'll also need a **handler** and to **register MediatR** in your application.

Would you like a complete example including the DTO and repository interfaces?

By messaging ChatGPT, you agree to our [Terms](https://openai.com/terms) and have read our [Privacy Policy](https://openai.com/privacy). See Cookie Preferences.  
Don't share sensitive info. Chats may be reviewed and used to train our models. [Learn more](https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance)

# APPENDIX K

## COMPLETE EXAMPLE leaving nothing out

## Also adding a SQL Server database and any Entity Framework Core and LINQ code necessary to ensure a complete example.

**ChatGPT said:**

Absolutely! Here's a **comprehensive, complete example** of a .NET 6+ Web API project using **MediatR**, **Entity Framework Core** (with SQL Server), and the **CQRS** pattern for fetching a book by ID.

### 1. Project Structure

/MyBookApi

├── Controllers

│ └── BooksController.cs

├── Data

│ ├── ApplicationDbContext.cs

│ └── InMemoryBookRepository.cs (optional—for seeding/testing)

├── DTOs

│ └── BookDto.cs

├── Entities

│ └── Book.cs

├── Queries

│ ├── GetBookByIdQuery.cs

│ └── GetBookByIdQueryHandler.cs

├── Migrations (EF Core migration files)

└── Program.cs

### 2. DTOs/BookDto.cs

namespace MyBookApi.DTOs

{

public class BookDto

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Author { get; set; } = string.Empty;

}

}

### 3. Entities/Book.cs

namespace MyBookApi.Entities

{

public class Book

{

public Guid Id { get; set; }

public string Title { get; set; } = string.Empty;

public string Author { get; set; } = string.Empty;

}

}

### 4. Data/ApplicationDbContext.cs

using Microsoft.EntityFrameworkCore;

using MyBookApi.Entities;

namespace MyBookApi.Data

{

public class ApplicationDbContext : DbContext

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options) { }

public DbSet<Book> Books => Set<Book>();

}

}

### 5. Data/InMemoryBookRepository.cs *(optional – for local seed/testing)*

using MyBookApi.Entities;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace MyBookApi.Data

{

public class InMemoryBookRepository : IBookRepository

{

private readonly List<Book> \_books = new()

{

new Book { Id = Guid.Parse("11111111-1111-1111-1111-111111111111"), Title = "Clean Code", Author = "Robert C. Martin" },

new Book { Id = Guid.Parse("22222222-2222-2222-2222-222222222222"), Title = "The Pragmatic Programmer", Author = "Andrew Hunt" }

};

public Task<Book?> GetByIdAsync(Guid id)

{

return Task.FromResult(\_books.FirstOrDefault(b => b.Id == id));

}

}

}

### 6. (Optional) Data/BookRepository.cs *(EF Core implementation)*

using Microsoft.EntityFrameworkCore;

using MyBookApi.Entities;

using System;

using System.Threading.Tasks;

namespace MyBookApi.Data

{

public class BookRepository : IBookRepository

{

private readonly ApplicationDbContext \_ctx;

public BookRepository(ApplicationDbContext ctx) => \_ctx = ctx;

public async Task<Book?> GetByIdAsync(Guid id)

{

return await \_ctx.Books.FindAsync(id);

}

}

}

### 7. IBookRepository.cs *(Interface)*

using MyBookApi.Entities;

using System;

using System.Threading.Tasks;

namespace MyBookApi.Data

{

public interface IBookRepository

{

Task<Book?> GetByIdAsync(Guid id);

}

}

### 8. Queries/GetBookByIdQuery.cs

using MediatR;

using MyBookApi.DTOs;

using System;

namespace MyBookApi.Queries

{

public record GetBookByIdQuery(Guid Id) : IRequest<BookDto?>;

}

### 9. Queries/GetBookByIdQueryHandler.cs

using MediatR;

using MyBookApi.Data;

using MyBookApi.DTOs;

using System.Threading;

using System.Threading.Tasks;

namespace MyBookApi.Queries

{

public class GetBookByIdQueryHandler : IRequestHandler<GetBookByIdQuery, BookDto?>

{

private readonly IBookRepository \_repository;

public GetBookByIdQueryHandler(IBookRepository repository)

=> \_repository = repository;

public async Task<BookDto?> Handle(GetBookByIdQuery request, CancellationToken cancellationToken)

{

var book = await \_repository.GetByIdAsync(request.Id);

return book == null

? null

: new BookDto

{

Id = book.Id,

Title = book.Title,

Author = book.Author

};

}

}

}

### 10. Controllers/BooksController.cs

using MediatR;

using Microsoft.AspNetCore.Mvc;

using MyBookApi.DTOs;

using MyBookApi.Queries;

using System;

using System.Threading.Tasks;

namespace MyBookApi.Controllers

{

[ApiController]

[Route("api/[controller]")]

public class BooksController : ControllerBase

{

private readonly IMediator \_mediator;

public BooksController(IMediator mediator) => \_mediator = mediator;

[HttpGet("{id}")]

public async Task<IActionResult> GetById(Guid id)

{

var book = await \_mediator.Send(new GetBookByIdQuery(id));

return book is null ? NotFound() : Ok(book);

}

}

}

### 11. Program.cs (Entry point & DI setup)

using MediatR;

using Microsoft.EntityFrameworkCore;

using MyBookApi.Data;

using MyBookApi.Queries;

var builder = WebApplication.CreateBuilder(args);

// Configure EF Core + SQL Server

builder.Services.AddDbContext<ApplicationDbContext>(options =>

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

// Register repository (choose only one)

builder.Services.AddScoped<IBookRepository, BookRepository>();

// builder.Services.AddScoped<IBookRepository, InMemoryBookRepository>(); // use for testing

// Register MediatR

builder.Services.AddMediatR(cfg =>

cfg.RegisterServicesFromAssemblyContaining<GetBookByIdQuery>());

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

### 12. appsettings.json *(Connection string)*

{

"ConnectionStrings": {

"DefaultConnection": "Server=localhost;Database=MyBookDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

}

}

### 13. EF Core Migrations & Database Setup

Run the following in your terminal (project directory):

dotnet tool install --global dotnet-ef # if needed

dotnet ef migrations add InitialCreate # scaffold migration

dotnet ef database update # create database + schema

This will create the Books table mirroring your Book entity.

### 14. Testing the API

1. Launch the API.
2. Add a book to the database manually or via seed script (if using EF repository).
3. Make a request:

GET https://localhost:5001/api/books/11111111-1111-1111-1111-111111111111

* If the book exists → 200 OK with JSON:
* {
* "id": "11111111-1111-1111-1111-111111111111",
* "title": "Clean Code",
* "author": "Robert C. Martin"
* }
* If not found → 404 Not Found.

### Summary Quick View

| **Layer** | **Code File** | **Purpose** |
| --- | --- | --- |
| DTO | BookDto.cs | Defines the shape of API response |
| Entity | Book.cs | Database model |
| Data Context | ApplicationDbContext.cs | EF Core DB connection |
| Repository | BookRepository.cs *(or InMemory)* | Data access abstraction |
| Query | GetBookByIdQuery.cs | MediatR request definition |
| Handler | GetBookByIdQueryHandler.cs | Handles query via EF/Repo logic |
| Controller | BooksController.cs | Exposes a REST endpoint |
| Program | Program.cs | Bootstraps DI, MediatR, EF, Swagger |

This is a **completely working example**—nothing left out! You can expand by adding CreateBookCommand, more queries, validation, DI configurations, etc.

Let me know if you'd like help scaffolding commands, validation (e.g., FluentValidation), automated seeding, or unit tests!

Top of Form

# APPENDIX L

## EXPANDED EXAMPLE:A full enhancement of the project:

### CreateBookCommand (CQRS write pattern)

### FluentValidation (for input validation)

### EF Core seeding (automatic SQL Server DB seeding)

### Swagger setup and testing flow

### Optional: basic unit test setup (text-only guidance)

All of this uses **Entity Framework Core + SQL Server**, no shortcuts or in-memory trickery.

**✅ 1. CreateBookCommand Implementation**

**📄 Commands/CreateBookCommand.cs**

using MediatR;

using MyBookApi.DTOs;

namespace MyBookApi.Commands

{

public record CreateBookCommand(string Title, string Author) : IRequest<BookDto>;

}

**📄 Commands/CreateBookCommandHandler.cs**

using MediatR;

using MyBookApi.Data;

using MyBookApi.DTOs;

using MyBookApi.Entities;

namespace MyBookApi.Commands

{

public class CreateBookCommandHandler : IRequestHandler<CreateBookCommand, BookDto>

{

private readonly ApplicationDbContext \_dbContext;

public CreateBookCommandHandler(ApplicationDbContext dbContext)

{

\_dbContext = dbContext;

}

public async Task<BookDto> Handle(CreateBookCommand request, CancellationToken cancellationToken)

{

var book = new Book

{

Id = Guid.NewGuid(),

Title = request.Title,

Author = request.Author

};

\_dbContext.Books.Add(book);

await \_dbContext.SaveChangesAsync(cancellationToken);

return new BookDto

{

Id = book.Id,

Title = book.Title,

Author = book.Author

};

}

}

}

**📄 Update BooksController.cs**

using MyBookApi.Commands;

[HttpPost]

public async Task<IActionResult> Create(CreateBookCommand command)

{

var result = await \_mediator.Send(command);

return CreatedAtAction(nameof(GetById), new { id = result.Id }, result);

}

**✅ 2. Add FluentValidation**

**🔧 Install NuGet Packages**

dotnet add package FluentValidation

dotnet add package FluentValidation.AspNetCore

**📄 Validators/CreateBookCommandValidator.cs**

using FluentValidation;

using MyBookApi.Commands;

namespace MyBookApi.Validators

{

public class CreateBookCommandValidator : AbstractValidator<CreateBookCommand>

{

public CreateBookCommandValidator()

{

RuleFor(x => x.Title)

.NotEmpty().WithMessage("Title is required.")

.MaximumLength(100);

RuleFor(x => x.Author)

.NotEmpty().WithMessage("Author is required.")

.MaximumLength(100);

}

}

}

**📄 Update Program.cs**

using FluentValidation;

using FluentValidation.AspNetCore;

// After AddControllers()

builder.Services.AddFluentValidationAutoValidation();

builder.Services.AddValidatorsFromAssemblyContaining<CreateBookCommandValidator>();

**✅ 3. Seed SQL Server Data Automatically**

**📄 Data/ApplicationDbInitializer.cs**

using MyBookApi.Entities;

namespace MyBookApi.Data

{

public static class ApplicationDbInitializer

{

public static void SeedDatabase(WebApplication app)

{

using var scope = app.Services.CreateScope();

var context = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

if (!context.Books.Any())

{

context.Books.AddRange(

new Book

{

Id = Guid.Parse("11111111-1111-1111-1111-111111111111"),

Title = "Clean Code",

Author = "Robert C. Martin"

},

new Book

{

Id = Guid.Parse("22222222-2222-2222-2222-222222222222"),

Title = "The Pragmatic Programmer",

Author = "Andrew Hunt"

}

);

context.SaveChanges();

}

}

}

}

**📄 Update Program.cs**

Add this before app.Run();:

ApplicationDbInitializer.SeedDatabase(app);

**✅ 4. Swagger Setup**

Already mostly handled if you use this:

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

Just ensure this is in Program.cs:

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

* Run dotnet run and navigate to:  
  https://localhost:5001/swagger

**✅ 5. Unit Test Setup (Guidance)**

If you'd like to write unit tests:

**📁 Project structure:**

/MyBookApi

/MyBookApi.Tests

**Add test project:**

dotnet new xunit -n MyBookApi.Tests

dotnet add MyBookApi.Tests reference MyBookApi

**Example Test: CreateBookCommandHandlerTests.cs**

using Xunit;

using Moq;

using MyBookApi.Commands;

using MyBookApi.Entities;

using MyBookApi.Data;

using Microsoft.EntityFrameworkCore;

public class CreateBookCommandHandlerTests

{

[Fact]

public async Task Handle\_ShouldCreateBookAndReturnDto()

{

var options = new DbContextOptionsBuilder<ApplicationDbContext>()

.UseInMemoryDatabase("TestDb")

.Options;

using var context = new ApplicationDbContext(options);

var handler = new CreateBookCommandHandler(context);

var command = new CreateBookCommand("Test Book", "Test Author");

var result = await handler.Handle(command, default);

Assert.NotNull(result);

Assert.Equal("Test Book", result.Title);

Assert.Equal("Test Author", result.Author);

}

}

**✅ Bonus: appsettings.Development.json with Connection String**

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=MyBookDb;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

Make sure your SQL Server instance is running. You can also use localhost, 127.0.0.1, or a Docker container.

**✅ 6. Recap: Packages You Need**

Make sure your .csproj includes:

<PackageReference Include="MediatR" Version="12.\*" />

<PackageReference Include="MediatR.Extensions.Microsoft.DependencyInjection" Version="12.\*" />

<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.\*" />

<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.\*" />

<PackageReference Include="FluentValidation" Version="11.\*" />

<PackageReference Include="FluentValidation.AspNetCore" Version="11.\*" />

**✅ Final Thoughts**

You now have a **complete, production-like .NET Web API** using:

* ✅ **SQL Server**
* ✅ **Entity Framework Core**
* ✅ **CQRS via MediatR** (Queries and Commands)
* ✅ **Input validation** with FluentValidation
* ✅ **Swagger UI** for testing
* ✅ **Unit test setup** using xUnit
* ✅ **Seeding** SQL Server on startup

Would you like to add pagination, filtering, update/delete commands, or Docker support for SQL Server?

**You said:**

Why are you using a GUID Id in the Seed data? And yes please provide the code to implement Docker containerization to the project.

**ChatGPT said:**

Great questions! Let's tackle both:

**🧩 Why Use a GUID for Id in Seed Data?**

**✅ Reasons for Using Guid:**

1. **Uniqueness Across Systems**: GUIDs are globally unique, making them safer for distributed systems or merging data from multiple sources.
2. **Predictable Seeding**: By manually assigning GUIDs in your seed data, you:
   * Avoid auto-generation.
   * Can reference seeded records reliably in tests or other entities (e.g., foreign keys).
3. **Client-Side Generation**: GUIDs allow clients to generate IDs offline (if needed), which is common in mobile/distributed apps.

**❗Downsides (When to Use int Instead):**

* Slightly larger index size in SQL Server.
* Harder to read/debug.

**Summary**: For demo purposes, seeded Guids are stable and useful. In a production app, you might prefer int with auto-increment unless you have a distributed or syncing scenario.

**🐳 Docker Containerization for SQL Server + Web API**

Now, let’s fully containerize your app with:

1. ✅ SQL Server in Docker
2. ✅ .NET 6+ Web API Dockerized
3. ✅ Docker Compose for orchestration

**📁 Folder Setup:**

/MyBookApi

├── Controllers/

├── Data/

├── ...

├── Dockerfile

└── docker-compose.yml

**📄 1. Dockerfile (for .NET Web API)**

# Build stage

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /src

COPY \*.csproj ./

RUN dotnet restore

COPY . ./

RUN dotnet publish -c Release -o /app

# Runtime stage

FROM mcr.microsoft.com/dotnet/aspnet:8.0

WORKDIR /app

COPY --from=build /app ./

EXPOSE 80

ENTRYPOINT ["dotnet", "MyBookApi.dll"]

Replace MyBookApi.dll with your actual output DLL if named differently.

**📄 2. docker-compose.yml**

version: '3.8'

services:

api:

build: .

ports:

- "5000:80"

environment:

- ASPNETCORE\_ENVIRONMENT=Development

- ConnectionStrings\_\_DefaultConnection=Server=sqlserver;Database=MyBookDb;User=sa;Password=Your\_password123;

depends\_on:

- sqlserver

sqlserver:

image: mcr.microsoft.com/mssql/server:2022-latest

container\_name: sqlserver

environment:

SA\_PASSWORD: "Your\_password123"

ACCEPT\_EULA: "Y"

ports:

- "1433:1433"

volumes:

- sqlserverdata:/var/opt/mssql

volumes:

sqlserverdata:

**📄 3. appsettings.json (Use container-friendly SQL Server host)**

{

"ConnectionStrings": {

"DefaultConnection": "Server=sqlserver;Database=MyBookDb;User=sa;Password=Your\_password123;"

}

}

* sqlserver is the name of the Docker service in docker-compose.yml.

**✅ 4. Add EF Migrations to Docker Image (Optional but recommended)**

If you want to auto-apply migrations in Docker, modify Program.cs like so:

using Microsoft.EntityFrameworkCore;

var app = builder.Build();

// Auto-migrate

using (var scope = app.Services.CreateScope())

{

var db = scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();

db.Database.Migrate(); // Apply any pending migrations

}

ApplicationDbInitializer.SeedDatabase(app);

**📦 5. Build and Run with Docker Compose**

From project root:

docker-compose build

docker-compose up

**🧪 Test It:**

* Web API available at: http://localhost:5000/swagger
* SQL Server accessible on port 1433 (you can connect via SSMS or Azure Data Studio)

**📥 Optional: .dockerignore**

bin/

obj/

\*.db

**🧼 To Stop and Clean**

docker-compose down -v # removes containers + volume (SQL Server data)

**✅ Summary**

* ✔ GUIDs are used in seeding for stable, unique identity across environments.
* 🐳 Full Docker containerization includes:
  + SQL Server container
  + .NET Web API container
  + docker-compose orchestration
  + Automatic EF Core migrations + data seeding

Would you like to add:

* ✅ EF Core **logging** to console?
* ✅ Integration test setup for docker-compose?

✅ Swagger JSON auth tokens or JWT setup for secure APIs?

Bottom of Form

Top of Form

Bottom of Form

Bottom of Form

Bottom of Form

Top of Form

Bottom of Form