Full CRUD for Any Entity (Service + Repo + Controller)

I'm working on a backend using C# and Onion Architecture (Domain → Application → Infrastructure → API).
Please generate the full CRUD setup for the entity: `{EntityName}`.

Requirements:
- Generate Create, Update, GetById, GetAll, and Delete methods.
- Include:
  - DTOs (Create{EntityName}Dto, Update{EntityName}Dto, {EntityName}Dto)
  - AutoMapper Profile
  - Repository interface and implementation (EF Core)
  - Service interface and implementation
  - ASP.NET Core Controller
- Use async patterns, clean architecture principles, AutoMapper, and FluentValidation.

Generate DTOs

Generate 3 DTOs for the entity `{EntityName}`:
- Create{EntityName}Dto
- Update{EntityName}Dto
- {EntityName}Dto

Include appropriate data types and validation annotations (like [Required], [StringLength], etc.)

Generate FluentValidator for DTO

Generate a FluentValidation class in C# for the DTO `{DtoName}`.

Include:
- [NotEmpty], [EmailAddress], [Length], and other relevant validations
- Class should inherit from `AbstractValidator<{DtoName}>`

AutoMapper Profile

Generate an AutoMapper Profile class for `{EntityName}` in C#.

It should map:
- Create{EntityName}Dto → {EntityName}
- Update{EntityName}Dto → {EntityName}
- {EntityName} → {EntityName}Dto

Repository Interface + Implementation

Generate a repository interface and its EF Core implementation for `{EntityName}`.

Include async methods:
- GetAllAsync
- GetByIdAsync
- AddAsync
- UpdateAsync
- DeleteAsync

Interface goes in `Application.Interfaces`, implementation in `Infrastructure.Repositories`.

Service Interface + Implementation

Generate a service interface and implementation class for managing `{EntityName}`.

- Use AutoMapper for conversions between DTOs and Entity.
- Use async methods: Create, GetById, GetAll, Update, Delete.
- Inject repository.
- Return DTOs from service methods, not entities.

Controller

Generate an ASP.NET Core API Controller in C# for `{EntityName}`.

- Inject the service in constructor.
- Define endpoints: GET (all, by id), POST, PUT, DELETE.
- Use proper route attributes and IActionResult return types.
- Use async methods and return appropriate status codes.

Unit Tests for Service

Generate xUnit tests for `{EntityName}Service` in C#.

- Use Moq to mock the repository and AutoMapper.
- Test cases: Create, GetById, Update, Delete.
- Check for correct return types and null/exception cases.

Update Program.cs for Dependency Injection

Update Program.cs in .NET 8 to register dependencies for `{EntityName}`:

- AddScoped for repository and service interfaces.
- Register AutoMapper and FluentValidation.
- Add Swagger and CORS config.
- Assume minimal API.

Master Prompt: All Layers for One Entity

I'm building a .NET 8 backend using Onion Architecture (Domain → Application → Infrastructure → API).
Please generate the complete code for managing the entity: `{EntityName}`.

Architecture Stack:
- EF Core for data access
- AutoMapper for DTO ↔ Entity mapping
- FluentValidation for input validation
- Async Task-based service and repository methods
- Clean layering (Interfaces, DI, Separation of Concerns)

Please generate:
1. **DTOs**:
   - Create{EntityName}Dto
   - Update{EntityName}Dto
   - {EntityName}Dto

2. **FluentValidation**:
   - Validators for Create and Update DTOs using FluentValidation syntax

3. **AutoMapper**:
   - Mapping Profile between DTOs and Entity

4. **Repository**:
   - Interface in `Application.Interfaces`
   - EF Core Implementation in `Infrastructure.Repositories`
   - Methods: GetAllAsync, GetByIdAsync, AddAsync, UpdateAsync, DeleteAsync

5. **Service**:
   - Interface and implementation for CRUD
   - Inject repository and AutoMapper

   - Return DTOs
   - Handle not-found cases gracefully

6. **Controller**:
   - ASP.NET Core Web API controller
   - Endpoints: GetAll, GetById, Post, Put, Delete
   - Use DTOs and async Task<IActionResult> methods
   - Proper route annotations and status codes

7. **Program.cs Setup**:
   - Dependency injection for repository, service
   - Register AutoMapper
   - Register FluentValidation
   - Add Swagger and CORS

Notes:
- Assume external authentication and logging are handled via DLLs in `/ExternalLibs`
- Keep code modular, clean, and production-ready

Ultimate Master Prompt — Full Application

I'm building a complete backend application in C# (.NET 8+) using **Onion Architecture** with clean code principles. Please generate all required boilerplate and implementations for this project.

🧱 **Project Overview**
- Application Name: `{AppName}`
- Architecture: Onion (Domain → Application → Infrastructure → API)
- ORM: Entity Framework Core (Code First or Reverse Engineered)
- Language: C#
- Database: SQL Server
- Tools: AutoMapper, FluentValidation
- Logging & Auth: Handled via third-party DLLs in `/ExternalLibs`
- Pattern: Controller → Service → Repository → Database

🎯 **Required Deliverables**

1. ✅ **Solution Structure**
   - `YourApp.Domain`: Entities, Interfaces

- `YourApp.Application`: DTOs, Interfaces, Services, Validators
- `YourApp.Infrastructure`: Repositories, EF Context, 3rd-party integrations
- `YourApp.API`: Controllers, Program.cs, Middleware

2. ✅ **Entity Layer (Domain)**
   - Create entity classes for: `{Entity1}`, `{Entity2}`, ...
   - Define properties, relationships (1-to-many, many-to-many), navigation

3. ✅ **DTOs & Mapping (Application)**
   - Create:
     - Create{Entity}Dto
     - Update{Entity}Dto
     - {Entity}Dto
   - AutoMapper Profile for each entity

4. ✅ **FluentValidation**
   - Validators for Create and Update DTOs
   - Located in `Validators` folder, clean and separate

5. ✅ **Repositories (Application + Infrastructure)**
   - I{Entity}Repository with CRUD
   - {Entity}Repository using EF Core
   - Dependency Injection ready

6. ✅ **Services**
   - I{Entity}Service interface
   - {Entity}Service: async methods, maps DTOs, handles exceptions

7. ✅ **Controllers (API)**
   - ASP.NET Core Web API Controllers for each entity
   - Routes: `api/{entity}` with GET, POST, PUT, DELETE
   - Uses DTOs and async pattern

8. ✅ **DbContext Configuration**
   - `AppDbContext` for all entities
   - Fluent API configuration if needed
   - Connection string placeholder

9. ✅ **Program.cs Setup**
   - Register all services, repositories
   - Add AutoMapper, FluentValidation
   - Enable Swagger, CORS
   - Load external DLLs for Auth and Logging from `/ExternalLibs` folder

10. ✅ **Error Handling & Response**
    - Centralized error middleware
    - Custom API response model (e.g., ApiResponse<T>)

11. ✅ **Folder and File Organization**
   - Organize everything by feature (optional)
   - Clean structure, easily extensible

⚙️ **Additional Instructions**
- Follow SOLID principles
- Use async/await consistently
- Keep layers loosely coupled
- Avoid putting logic in controllers
- Return meaningful error messages and status codes

🚨 **Final Output**
Generate complete working code (or stubs where needed) for all of the above. The output should be ready for copy-paste and compilation in a Visual Studio solution.

You can assume `{Entity1}` = `User`, `{Entity2}` = `LoanApplication`, etc.

Architecture Selector Prompt

I want to build a C# backend for a {ProjectType} app. Suggest the best architectural pattern (Onion, Clean, CQRS, Layered, DDD, etc.) and explain why it's suitable. Then generate the base folder structure for it.

UNIVERSAL BACKEND MASTER PROMPT (for .NET 8, Any Architecture)

I am building a backend application using C# and .NET 8.

Please help me generate a complete backend structure using one of the following architectures:
- Clean Architecture
- Onion Architecture
- Layered Architecture
- (You choose the most suitable one based on the app type below)

🧩 Application Context:

- Project Name: {AppName}
- Application Type: {e.g., Banking, E-commerce, Healthcare, Ticket Booking}
- Entities: {Entity1, Entity2, Entity3, ...}
- Database: SQL Server using EF Core
- ORM: Entity Framework Core (Code-First or Database-First as needed)
- Patterns: SOLID, async/await, Clean Code
- External Libraries:
  - AutoMapper
  - FluentValidation
  - xUnit + Moq for Testing
  - Auth & Logging handled via DLLs in `/ExternalLibs`

🎯 Deliverables:

1. ✅ **Folder Structure & Architecture**
   - Suggest best-fit architecture (Onion, Clean, etc.) based on app type
   - Generate base folder structure with projects like:
     - `.Domain`, `.Application`, `.Infrastructure`, `.API`
     - Optionally `.Contracts`, `.Shared`, `.Tests`

2. ✅ **Entity Generation**
   - Define C# entity classes with properties and relationships
   - Use EF Core conventions + Fluent API config

3. ✅ **DTOs**
   - CreateDTO, UpdateDTO, and ReadDTO for each entity
   - Follow naming conventions and clean separation of concerns

4. ✅ **FluentValidation**
   - Validators for Create and Update DTOs
   - Use AbstractValidator<T> and relevant rules (e.g., NotEmpty, EmailAddress, etc.)

5. ✅ **AutoMapper**
   - Generate a Profile class for all DTO↔Entity mappings

6. ✅ **Repository Pattern**
   - Interface in `Application.Interfaces`
   - Implementation in `Infrastructure.Repositories`
   - Async methods: GetAll, GetById, Add, Update, Delete

7. ✅ **Service Layer**
   - Interface and implementation for each entity's business logic
   - Inject repository and mapper
   - Use DTOs in input/output
   - Return Task-based async methods

8. ✅ **Controller Layer**
   - ASP.NET Core Web API Controllers
   - Endpoints: GET, POST, PUT, DELETE
   - Use dependency injection and return IActionResult
   - Use [ApiController], [Route("api/[controller]")]

9. ✅ **DbContext Configuration**
   - AppDbContext with DbSet<TEntity> properties
   - Configure relationships using Fluent API

10. ✅ **Program.cs Setup**
    - Register services, repositories, AutoMapper, FluentValidation
    - Load Auth and Logging DLLs from `/ExternalLibs`
    - Enable Swagger and CORS

11. ✅ **Error Handling**
    - Centralized error handling middleware
    - Return standardized API response format (e.g., ApiResponse<T>)

12. ✅ **Unit Testing**
    - Generate xUnit tests for each Service
    - Use Moq to mock repositories and mapper
    - Cover success and failure paths

🧠 Additional Requirements:
- Use async/await in all data/service/controller methods
- Ensure clean code, single responsibility, and separation of concerns
- Don't mix logic in DTOs or Controllers
- Output all generated code as per .NET 8 standards

👋 Let me know if you need anything else like:
- CQRS support
- MediatR integration
- Role-based Authorization
- Swagger Example Requests

Keywords:

List of **powerful, intent-driven prompt keywords and phrases** to unlock the **full potential of Cursor.ai** — especially tailored for **.NET backend development**, **clean architecture**, and **enterprise-quality codebases**.

Use these keywords to **steer the conversation**, **get production-quality output**, and **refine your codebase collaboratively** — not just generate it.

🧠 Prompt Keywords & Phrases for Maximum Impact in Cursor.ai

🔨 For High-Quality Code Generation

🎯 Intent	✅ Use Keywords/Phrases
Get production-ready code	`Generate production-ready codeFollow clean coding and SOLID principles`
Enterprise-grade structure	`Generate scalable, maintainable codeFollow best practices used in enterprise apps`
Full-stack layering	`Use Onion ArchitectureUse Clean ArchitectureUse Layered Architecture`
Async everywhere	`Use async/await across repository, service, and controller layers`
Error handling	`Include centralized error handling middleware`
Dependency injection	`Setup DI properly in Program.cs`
Service/Repository pattern	`Use repository and service pattern with interfaces and implementations`
DTO usage	`Keep DTOs clean and separate from entities`
Don't write DTO logic	`Avoid putting logic in DTOs or Controllers`
AutoMapper setup	`Generate AutoMapper profile with mapping between DTOs and entities`
FluentValidation integration	`Use FluentValidation for Create and Update DTOs`

Testing	`Generate xUnit tests with Moq for service layer`

🧭 For Design-First or Discussion-Based Guidance

🎯 Intent	✅ Use Keywords/Phrases
Plan architecture first	`Imagine you're a .NET architect. Let's design the architecture first before coding.`
Don't code yet	`Don't write code yet. Let's just discuss the flow.`
Refactor-oriented	`Review and refactor this code using best practices.`
Ask for patterns	`What design pattern is best suited for this feature?`
API-first mindset	`Let's define the API contract and flow before coding.`
State machine/domain workflow	`Model this logic using a domain-driven state machine.`
Model relationships clearly	`Design entity relationships using EF Core Fluent API.`

🛠️ For Cursor.ai-Specific Power Features

🎯 Intent	✅ Use Keywords/Phrases
Cursor understands intent	`Show me where I might be violating SOLID principles`
Local reasoning	`Based on my current file/project structure, suggest better layering`
Code navigation	`Summarize this classExplain what this method does`
Insert intelligently	`Insert repository code for {EntityName} right here`

Cross-file awareness	`Does this DTO match the entity in the domain layer?`
Explain complex block	`Explain this method line by line like I'm a senior dev`

🧹 For Cleanup, Optimization, and Refactoring

🎯 Intent	✅ Use Keywords/Phrases
Improve readability	`Refactor this code for better readability and maintainability`
Reduce duplication	`Identify duplicate logic and move to shared methods or services`
Modernize syntax	`Use modern C# features (like records, pattern matching)`
Optimize LINQ	`Optimize this LINQ query for performance and readability`
Improve naming	`Suggest better method/variable/class names`
Highlight anti-patterns	`Are there any anti-patterns in this implementation?`

🧩 Example Master Prompt Using These Keywords

Imagine you're a .NET Architect building a high-scale financial application in .NET 8 using Onion Architecture.

Let's not write code yet — I want to discuss the end-to-end flow for managing "LoanApplication" entity including validation, mapping, persistence, and exposure via RESTful APIs.

Make sure your design supports:
- Clean layering (Domain, Application, Infrastructure, API)
- Async repository/service/controller
- DTO separation
- AutoMapper and FluentValidation
- xUnit + Moq support for service testing
- Production-grade logging/auth (via DLLs)

Once the flow is finalized, generate production-ready code step-by-step with clean structure and proper DI setup.

Would you like this compiled into a downloadable “**Cursor Prompt Cheat Sheet for .NET Devs**”? Or I can create a reusable **prompt template file** you can store in your repo and reuse for every feature/module.