

Hinweise

Wichtige Grundregeln

- **1:n-Beziehungen:** Immer zuerst die 1-Seite, dann die n-Seite implementieren
- **IDE-Features:** Strg+. für „Generate Method“, „Implement Interface“, „Add using“
- **Copy-Paste-Pattern:** Template kopieren → anpassen → generieren
- ▷ EF Core setzt FK automatisch — ▷ Immer await bei Factory — ▷ null! für Required Nav. Props

1 Phase 1: Domain Layer (Entitäten & Geschäftslogik)

1.1 Entities - „1“-Seite zuerst (z.B. Author)

1. Domain/Entities/Author.cs erstellen, von BaseEntity erben
2. Private Properties definieren (Id von BaseEntity, eigene Properties mit private set)
3. Private parameterloser Konstruktor für EF Core: `private Author() {}`
4. Navigation Property zur „n“-Seite: `public ICollection<Book> Books { get; set; } = new List<Book>();`
5. Factory-Methode `CreateAsync()` als `public static async Task<Author>` definieren
6. In Factory: ArgumentNullException für Objekt-Parameter werfen
7. In Factory: String-Parameter trimmen: `var trimmed = (param ?? string.Empty).Trim();`
8. In Factory: Interne Validierung aufrufen: `ValidateAuthorProperties(...)`
9. In Factory: Externe Validierung (Uniqueness): `await AuthorSpecifications.ValidateAuthorExternal(...)`
10. In Factory: Neue Instanz mit private Konstruktor zurückgeben
11. Update-Methode `UpdateAsync()` analog zu Factory (ohne new, Properties direkt setzen)

1.2 Entities - „n“-Seite (z.B. Book)

12. Domain/Entities/Book.cs erstellen, von BaseEntity erben
13. Properties analog, aber: `public int AuthorId { get; private set; }` für Fremdschlüssel
14. Navigation Property zur „1“-Seite: `public Author Author { get; set; } = null!;`
15. Bei weiterer „n“-Seite: `public ICollection<Loan> Loans { get; set; } = new List<Loan>();`
16. Factory mit Author-Objekt als Parameter: `CreateAsync(string isbn, Author author, ...)`
17. In Factory: Book und Author verknüpfen, AuthorId NICHT manuell setzen
18. Domain-Methoden für Business-Logik: `DecreaseCopies()`, `IncreaseCopies()`

1.3 Entities - Weitere „n“-Seite (z.B. Loan)

19. Domain/Entities/Loan.cs analog zu Book, mit BookId und Book Navigation Property
20. Spezielle Properties: `DateTime? ReturnDate` für optionale Werte
21. Factory-Methode mit Book-Objekt: `Create(Book book, ...)`
22. Business-Methoden: `MarkAsReturned(DateTime)`, `IsOverdue()`

1.4 Domain Specifications - „1“-Seite

23. Domain/Specifications/AuthorSpecifications.cs als static class erstellen
24. Konstanten definieren (`MinLength`, `MaxLength`)
25. Einzelne Check-Methoden: `public static DomainValidationResult CheckFirstName(...)`
26. Pattern: Prüfung → ? Success(...) : Failure(...)
27. `ValidateAuthorInternal()`: Liste von Checks, bei Fehler `DomainValidationException`
28. `ValidateAuthorExternal()`: Uniqueness-Check mit `IAuthorUniquenessChecker`

1.5 Domain Specifications - „n“-Seiten

29. Domain/Specifications/BookSpecifications.cs analog zu Author
30. Spezielle Checks: ISBN-Format (13 Ziffern), PublicationYear (Bereich)
31. Domain/Specifications/LoanSpecifications.cs: Checks für Dates, BorrowerName
32. Konstante: `public const int StandardLoanDurationDays = 14;`

1.6 Domain Contracts (Interfaces)

33. Domain/Contracts/IAuthorUniquenessChecker.cs: `Task<bool> IsUniqueAsync(...)`
34. Domain/Contracts/IBookUniquenessChecker.cs: `Task<bool> IsUniqueAsync(...)`

- ▷ **Factory:** Private Ctor + async Factory
- ▷ **Trimmen:** `(param??).Trim()`
- ▷ **Nav Props:** `null!` (Required), `new List<>()` (Collection)
- ▷ **1:n-Beziehung:** Immer zuerst 1-Seite erstellen, dann n-Seite
- ▷ **n:m-Beziehung:** Join-Entity + 2x 1:n (z.B. BookAuthor)

2 Phase 2: Application Layer (Use Cases)

2.1 DTOs (Data Transfer Objects)

35. Application/Dtos/GetAuthorDto.cs als `public readonly record struct`
36. Application/Dtos/GetBookDto.cs mit `string? AuthorName` für lesbare Anzeige
37. Application/Dtos/GetLoanDto.cs mit `string? BookTitle` und `bool IsOverdue`

2.2 Application Interfaces

38. Application/Interfaces/Repositories/IAuthorRepository.cs von `IGenericRepository<Author>` erben
39. Zusätzliche Query-Methoden: `GetAuthorsWithBooksAsync()`, `GetByFullName()`, `GetByISBNAsync()`
40. Application/Interfaces/Repositories/IBookRepository.cs analog
41. Application/Interfaces/Repositories/ILoanRepository.cs: `GetLoansByBookIdAsync()`, `GetOverdueLoansAsync()`
42. Application/Interfaces/IUnitOfWork.cs erweitern: Properties für alle Repositories

2.3 Commands & Command Handlers - Create

43. Folder: Application/Features/Authors/Commands/CreateAuthor/
44. `CreateAuthorCommand.cs: record struct (...) : IRequest<Result<GetAuthorDto>>`
45. `CreateAuthorCommandValidator.cs: Von AbstractValidator<> erben, Rules im Konstruktor`
46. `CreateAuthorCommandHandler.cs: IRequestHandler<> implementieren (Strg+.)`
47. Handler-Konstruktor: `IUnitOfWork`, `IAuthorUniquenessChecker` injizieren
48. Handle-Methode: Factory aufrufen, `AddAsync()`, `SaveChangesAsync()`, DTO mappen, `Result.Created()`
49. Analog für Book: `CreateBookCommand`, Validator, Handler (Author laden!)
50. Analog für Loan: `CreateLoanCommand`, Handler (Book laden, `DecreaseCopies()`)

2.4 Commands & Command Handlers - Update

51. `UpdateAuthorCommand.cs mit int Id als ersten Parameter`
52. `UpdateAuthorCommandValidator.cs mit Id > 0 Regel`
53. `UpdateAuthorCommandHandler.cs: Entity laden, NotFound prüfen, UpdateAsync(), Save`
54. Analog für Book: `UpdateBookCommand`, Validator, Handler

2.5 Commands & Command Handlers - Delete

55. `DeleteAuthorCommand.cs: record struct (int Id) : IRequest<Result<bool>>`

56. DeleteAuthorCommandHandler.cs: Entity laden, Remove(), Save, Result.NoContent()
57. Analog für Book: DeleteBookCommand, Handler

2.6 Commands - Spezielle Operationen

58. ReturnLoanCommand.cs: record struct (int LoanId, DateTime ReturnDate)
59. ReturnLoanCommandHandler.cs: Loan laden (mit Book!), MarkAsReturned(), IncreaseCopies()

2.7 Queries & Query Handlers - GetById

60. Features/Authors/Queries/GetAuthorById/GetAuthorByIdQuery.cs: record struct mit int Id
61. GetAuthorByIdQueryHandler.cs: IRequestHandler implementieren, GetByIdAsync(), null-check
62. Analog für Book und Loan

2.8 Queries & Query Handlers - GetAll

63. GetAllAuthorsQuery.cs: record struct ohne Parameter
64. GetAllAuthorsQueryHandler.cs: GetAllAsync(), Collection von DTOs mappen
65. Analog für Books und Loans

2.9 Queries - Spezielle Abfragen

66. GetAuthorByIdQuery mit Validator für Id > 0
67. GetLoansByBookQuery.cs: Query mit int BookId Parameter
68. GetLoansByBookQueryHandler.cs: GetLoansByBookIdAsync(), DTOs mappen
69. GetOverdueLoansQuery.cs: Query ohne Parameter
70. GetOverdueLoansQueryHandler.cs: GetOverdueLoansAsync()

2.10 Mapster Configuration

71. Application/Common/Mappings/AuthorMappingConfig.cs: Static class, ConfigureAuthorMappings()
72. In Methode: TypeAdapterConfig<Author, GetAuthorDto>.NewConfig()
73. BookMappingConfig.cs: .Map(dest => dest.AuthorName, src => ...)
74. LoanMappingConfig.cs: .Map(dest => dest.IsOverdue, src => src.IsOverdue())

2.11 Application Dependency Injection

75. In Application/DependencyInjection.cs: MediatR registrieren
76. FluentValidation: .AddValidatorsFromAssembly(...)
77. ValidationBehavior: .AddTransient(typeof(IPipeline<,>))...
78. Mapster-Configs: AuthorMappingConfig.Configure...();

3 Phase 3: Infrastructure Layer (Datenzugriff)

3.1 Persistence - AppDbContext

79. Infrastructure/Persistence/AppDbContext.cs: DbSet-Properties hinzufügen
80. DbSets für alle Entities: Authors, Books, Loans
81. In OnModelCreating(): Fluent-API für Author (MaxLength, Required, Unique, RowVersion)
82. Fluent-API für Book: ISBN unique, Beziehung zu Author mit HasMany().WithOne().HasForeignKey()
83. Fluent-API für Loan: Beziehung zu Book konfigurieren
84. OnDelete-Verhalten: Cascade, Restrict, etc.

3.2 Repositories - Generic Repository

85. Infrastructure/Persistence/Repositories/GenericRepository.cs prüfen/anpassen

3.3 Repositories - Spezifische Repositories

86. Repositories/AuthorRepository.cs: Von GenericRepository<Author> und IAuthorRepository
87. Konstruktor: public AuthorRepository(AppDbContext ctx) : base(ctx) {}

88. Spezielle Methoden: GetAuthorsWithBooksAsync() mit .Include(a => a.Books)
89. GetByFullName(): .FirstOrDefaultAsync(a => (a.FirstName + + a.LastName) == fullName)
90. GetByISBNAsync(): .Include(a => a.Books).FirstOrDefaultAsync(a => a.Books.Any(...))
91. Repositories/BookRepository.cs analog
92. BookRepository: GetByIdAsync() und GetAllAsync() overriden für .Include(b => b.Author)
93. Repositories/LoanRepository.cs: GetLoansByBookIdAsync(), alle mit .Include(l => l.Book)

3.4 Unit of Work

94. Infrastructure/Persistence/UnitOfWork.cs: Properties für alle Repositories
95. Lazy Initialization: private IAuthorRepository? _authors; public IAuthorRepository Authors => ...
96. Analog für Books und Loans

3.5 Services - Uniqueness Checker

97. Services/AuthorUniquenessChecker.cs: IAuthorUniquenessChecker implementieren (Strg+.)
98. Konstruktor: AppDbContext injizieren
99. IsUniqueAsync(): Datenbank-Query ob Author existiert (außer eigene Id)
100. Services/BookUniquenessChecker.cs analog für ISBN-Prüfung

3.6 StartupDataSeeder

101. Services/StartupDataSeeder.cs: IHostedService implementieren
102. Nested class SeedDataUniquenessChecker mit explicit interface implementation
103. StartAsync(): CSV einlesen, Entities mit Factory-Methoden erstellen
104. Beziehungen auflösen (Author zu Book über ID-Mapping)
105. Alle Entities hinzufügen, EINMAL SaveChangesAsync()

3.7 StartupDataSeederOptions

106. Services/StartupDataSeederOptions.cs: Class mit public string CsvPath { get; set; }

3.8 Infrastructure Dependency Injection

107. Infrastructure/DependencyInjection.cs: DbContext registrieren
108. Repository: .AddScoped<IAuthorRepo, AuthorRepo>();
109. UnitOfWork: .AddScoped<IUnitOfWork, UnitOfWork>();
110. Uniqueness: .AddScoped<IAuthorUniqueness...>();
111. DataSeeder: .AddHostedService<StartupSeeder>();
112. Options: .Configure<StartupSeederOptions>(...)

- ▷ **EF**: .Include() für Nav Props
- ▷ **FK**: EF setzt automatisch via Nav Prop
- ▷ **Seed**: Erst Add(), dann 1x Save()
- ▷ **n:m**: HasMany().WithMany().UsingEntity<Join>()

4 Phase 4: API Layer (REST-Schnittstelle)

4.1 Result Extensions

113. Api/Extensions/ResultExtensions.cs prüfen .ToActionResult()

4.2 Controllers - „1“-Seite

114. Api/Controllers/AuthorsController.cs: [ApiController], [Route("{api}/{controller}")]
115. Konstruktor: IMediator mediator per primary constructor injizieren
116. [HttpGet] GetAll: await mediator.Send(new GetAllAuthorsQuery(), ct)
117. [HttpGet("{id:int}")] GetById mit ProducesResponseType-Attributnen
118. [HttpPost] Create: Command senden, result.ToActionResult(this, createdAtAction: ...)

119. `[HttpPut("{id:int}")]` Update: Id-Prüfung gegen Command.Id
120. `[HttpDelete("{id:int}")]` Delete: Command senden
121. XML-Kommentare für Swagger

4.3 Controllers - „n“-Seiten

122. `Api/Controllers/BooksController.cs` analog zu Authors
123. `Api/Controllers/LoansController.cs` mit speziellen Endpoints:
124. `[HttpPost]` CreateLoan
125. `[HttpPut("{id:int}/return")]` ReturnLoan mit `DateTime returnDate` im Body
126. `[HttpGet("book/{bookId:int}")]` GetLoansByBook
127. `[HttpGet("overdue")]` GetOverdueLoans

4.4 Program.cs

128. `Api/Program.cs`: Builder, Services registrieren
129. `builder.Services.AddApplication();` aufrufen
130. `builder.Services.AddInfrastructure(...);` aufrufen
131. Swagger mit XML-Kommentaren konfigurieren
132. Middleware: UseSwagger, UseSwaggerUI, MapControllers

4.5 appsettings.json

133. `Api/appsettings.json`: ConnectionString hinzufügen
134. Seeder-Config: `SStartupDataSeeder: { CsvPath: "..."}`

4.6 Seed-Daten

135. `Api/Data/library_seed_data.csv` erstellen mit Spalten für Authors und Books

5 Phase 5: Datenbank-Migration

136. Terminal öffnen, zu Api-Projekt navigieren
137. `dotnet ef migrations add InitialCreate` ausführen
138. `dotnet ef database update` ausführen
139. Prüfen ob Datenbank erstellt wurde (SQL Server Object Explorer)

6 Phase 6: Tests (Domain & Integration)

6.1 Domain Tests

140. `Domain.Tests/AuthorTests.cs`: FakeUniquenessChecker-Klassen erstellen
141. Tests für Factory-Methode: `CreateAsync_Succeeds_WithValidData`
142. Tests für Validierungen mit `[Theory]` und `[InlineData]`
143. Tests für Duplikat-Erkennung: `CreateAsync_DuplicateFullName_Throws`
144. `Domain.Tests/BookTests.cs` analog
145. `Domain.Tests/LoanTests.cs`: `MarkAsReturned_SetReturnDate, IsOverdue_ReturnsTrue`
146. `Domain.Tests/AuthorSpecificationsTests.cs` für einzelne Check-Methoden
147. Analog für BookSpecifications und LoanSpecifications

6.2 API Integration Tests - Setup

148. `Api.Tests/Utilities/TestWebApplicationFactory.cs`: `WebApplicationFactory<Program>`
149. `ConfigureWebHost()`: InMemory-Database, Repositories manuell registrieren
150. `db.Database.EnsureCreated()` aufrufen

6.3 API Integration Tests - Endpoints

151. `Api.Tests/Books/BooksEndpointTests.cs`: `IClassFixture<TestWebApplicationFactory<Program>>`
152. `HttpClient _client` im Konstruktor von Factory holen
153. Helper-Methode `GetFirstAuthorId()` um seeded Authors zu finden
154. Test: `GetAll_ReturnsOk_WithBooks` - einfacher GET-Test
155. Test: `Create_ReturnsCreated_WithValidData` - POST mit Command
156. Test: `Create_ReturnsBadRequest_WithInvalidISBN` - Validierungs-Test
157. Test: `Create_ReturnsNotFound_WithNonExistentAuthor` - Entity-Not-Found

158. Test: `GetById_ReturnsOk_WhenBookExists` - Erst erstellen, dann abrufen
159. Test: `Update_ReturnsOk_WithValidData` - PUT-Test
160. Test: `Update_ReturnsBadRequest_WhenIdMismatch` - Id-Prüfung
161. Test: `Delete_ReturnsNoContent_WhenBookExists` - DELETE-Test
162. Test: `Create_ReturnsConflict_WithDuplicateISBN` - Uniqueness-Test
163. `Api.Tests/Loans/LoansEndpointTests.cs` mit allen Loan-Endpunkt-Tests
164. Test: `CreateLoan_DecreasesAvailableCopies` - Business-Logik testen
165. Test: `ReturnLoan_IncreasesAvailableCopies` - Business-Logik testen
166. Test: `GetOverdueLoans_ReturnsOk_WithOverdueLoans` - Loan mit altem Datum
167. Test: `GetLoansByBook_ReturnsOk_WithLoans` - Mehrere Loans abrufen

- ▷ **InMemory**: DB für schnelle Tests
- ▷ **Helper**: `GetFirstAuthorId()`, `CreateTestBook()`
- ▷ **AAA**: Arrange-Act-Assert Pattern
- ▷ **Assert**: `.Should().BeOfType<Ok>()`

7 Phase 7: Finale Prüfungen

168. Alle Warnings beheben (Strg+ für Quick Fixes)
169. Solution builden: `dotnet build` - muss erfolgreich sein
170. Alle Tests ausführen: `dotnet test` - alle grün
171. API starten: `dotnet run --project Api`
172. Swagger aufrufen: <https://localhost:xxxx/swagger>
173. Alle Endpoints manuell testen: Create → GetAll → GetById → Update → Delete
174. Validierungen testen: Leere Strings, ungültige ISBNs, nicht existierende FKs
175. Business-Logik testen: AvailableCopies ändern sich bei Loan/Return
176. Overdue-Loans abrufen
177. Code-Review: Namenskonventionen, XML-Kommentare, async/await
178. Git: Branch erstellen, alle Änderungen committen