Die optimale Projektstruktur für ein C#-Projekt hängt von dessen Komplexität und Anforderungen ab. In professionellen Anwendungen wird meist eine **mehrschichtige Architektur** (Layered Architecture) verwendet. Eine gängige und bewährte Struktur ist die **Drei-Schichten-Architektur (Three-Tier Architecture)** oder eine erweiterte Form davon.

**Typische Layer einer C#-Anwendung**

1. **Presentation Layer (PL) / UI Layer**
   * Verantwortlich für die Benutzeroberfläche (z. B. Web-Frontend, GUI, API-Endpoints).
   * Beispiele: ASP.NET Core MVC, Razor Pages, Blazor, WPF, WinForms.
2. **Business Logic Layer (BLL) / Application Layer**
   * Enthält Geschäftslogik und Regeln der Anwendung.
   * Kommuniziert mit dem Data Access Layer, um Daten zu verarbeiten.
   * Implementiert Services, Validierungen, Berechnungen.
   * Beispiel: OrderService, CustomerService.
3. **Data Access Layer (DAL) / Persistence Layer**
   * Zuständig für den Datenzugriff (z. B. mit Entity Framework Core, Dapper, ADO.NET).
   * Enthält Repository-Klassen für CRUD-Operationen.
   * Beispiel: OrderRepository, CustomerRepository.

**Erweiterte Architektur mit zusätzlichen Layern**

In größeren Projekten kann es sinnvoll sein, weitere Schichten einzuführen:

1. **Domain Layer (Optional, für Domain-Driven Design)**
   * Enthält Geschäftsobjekte, Aggregate, Entitäten und Domain Services.
   * Keine direkte Abhängigkeit von der Infrastruktur.
   * Beispiel: Order, Customer, Product.
2. **Infrastructure Layer (Optional, für technische Abstraktionen)**
   * Enthält Implementierungen für externe Abhängigkeiten (z. B. Datenbank, Logging, Authentifizierung).
   * Beispiel: EfCoreRepository, FileLogger, JwtAuthService.

**Projektstruktur in einer .NET-Lösung**

Eine saubere Trennung der Layer in **separate Projekte** innerhalb einer .NET Solution (.sln) sorgt für eine bessere Wartbarkeit:

MySolution

│── MyApp.Presentation (ASP.NET Core API, MVC, Blazor, WPF)

│── MyApp.Application (Business-Logik, Services, DTOs)

│── MyApp.Domain (Modelle, Entities, Interfaces, Business-Regeln)

│── MyApp.Infrastructure (Datenbankzugriff, Repository, Logging)

│── MyApp.Tests (Unit-Tests, Integrationstests)

**Abhängigkeiten zwischen den Layern**

* **Presentation Layer → Application Layer**
* **Application Layer → Domain Layer + Infrastructure Layer**
* **Infrastructure Layer → Domain Layer**
* **Tests → alle Schichten (je nach Testart)**

**Zusätzliche Best Practices**

* Verwende **Dependency Injection (DI)**, um Schichten voneinander zu entkoppeln.
* Nutze **DTOs (Data Transfer Objects)**, um keine Entitäten direkt in der UI zu verwenden.
* Vermeide direkte Abhängigkeiten zwischen UI und DAL.
* Implementiere **Repository Pattern** oder **Unit of Work**, falls notwendig.

**Fazit:**  
Für einfache Projekte reicht eine **Drei-Schichten-Architektur** (UI → BLL → DAL). Bei komplexeren Anwendungen lohnt sich eine **Clean Architecture** oder **Hexagonal Architecture**, um langfristig flexibel und wartbar zu bleiben.