What is the onion architecture

**Onion Architecture** is a software design pattern that organizes an application into concentric layers, with the domain (business logic) at the core and outer layers (like infrastructure and UI) wrapping around it. Introduced by Jeffrey Palermo in 2008, it emphasizes separation of concerns, dependency inversion, and keeping the core business logic independent of external systems (e.g., databases, frameworks, or UI). It’s particularly popular in backend development, including .NET projects, for building maintainable, testable, and scalable systems.

The name "Onion" reflects its layered structure, where each layer depends only on the layers inside it, much like peeling an onion from the outside in.

**Core Principles of Onion Architecture**

Domain-Centric Design: The business logic (domain) is the heart of the application and doesn’t depend on outer layers.

Dependency Inversion: Outer layers depend on inner layers, not vice versa. Dependencies flow inward toward the core.

Separation of Concerns: Each layer has a specific responsibility, making the system modular and easier to maintain.

Testability: By isolating the domain logic, it’s easier to write unit tests without relying on external systems like databases.

**Layers of Onion Architecture**

**1. Domain Layer (Core)**

**Purpose**: Contains the business entities, rules, and logic—the essence of what the application does.

**Components:**

**Entities**: Core business objects (e.g., Order, Customer).

**Domain Services**: Business logic that operates on entities (e.g., OrderValidator).

**Interfaces**: Abstract contracts for infrastructure (e.g., IRepository).

**Key Trait**: No dependencies on external systems or frameworks—just pure business logic.

**2. Application Layer**

Purpose: Acts as an intermediary between the domain and outer layers, defining application-specific use cases and workflows.

Components:

Application Services: Coordinate business operations (e.g., OrderService to place an order).

DTOs (Data Transfer Objects): Used to pass data between layers.

Key Trait: Depends on the domain layer and defines interfaces for infrastructure but doesn’t implement them.

**3. Infrastructure Layer**

Purpose: Implements the details of external systems (e.g., databases, APIs, file systems) based on interfaces defined in inner layers.

**Components:**

Repositories: Concrete implementations (e.g., SqlOrderRepository).

External Services: Integrations with email, logging, or third-party APIs.

Key Trait: Depends on the application and domain layers, not the other way around.

**4. Presentation Layer (Outer Layer)**

Purpose: Handles user interaction and delivers the application to the end user.

Components:

Controllers: In ASP.NET Core, these handle HTTP requests.

UI: Web pages, desktop forms, or mobile screens.

Key Trait: Depends on the application layer to invoke use cases.

**Advantages of Onion Architecture**

Independence: The domain layer is isolated from frameworks, databases, or UI, making it portable and reusable.

Testability: Business logic can be tested without mocks for external systems, as dependencies are inverted.

Flexibility: Swap out infrastructure (e.g., switch from SQL Server to MongoDB) by changing only the infrastructure layer.

Maintainability: Clear boundaries reduce coupling and make changes easier.

Scalability: Outer layers can scale independently (e.g., add more web servers without touching the domain).

**Disadvantages of Onion Architecture**

Complexity: Requires more upfront design and understanding of dependency inversion, which can be overkill for small projects.

Learning Curve: Developers unfamiliar with the pattern may struggle with the inward dependency flow.

Boilerplate Code: Defining interfaces and wiring them up can add extra code compared to simpler architectures.