# Clean Architecture in .NET Core – Interview Guide with Code Snippets

---

## 🔹 Overview

Clean Architecture is a software design approach that separates the application into layers, enforcing a clear dependency rule: **outer layers depend on inner layers** but never the reverse. The focus is to keep business logic independent from frameworks, databases, or UI.

---

## 🔹 Layered Architecture

### 1. Domain Layer (Entities)

- Pure business rules
- No dependencies on other layers

**Example: Order Entity**

```
public class Order {
    public int Id { get; set; }
    public decimal TotalAmount { get; private set; }

    public void AddItem(decimal price) {
        TotalAmount += price;
    }
}
```

---

### 2. Application Layer (Use Cases)

- Contains use case logic and service interfaces
- Knows about the domain layer

**Example: CreateOrderHandler**

```
public class CreateOrderCommand : IRequest<int> {
    public List<decimal> Items { get; set; }
}

public class CreateOrderHandler : IRequestHandler<CreateOrderCommand, int> {
    private readonly IOrderRepository _repository;
```

```
    public CreateOrderHandler(IOrderRepository repository) {
        _repository = repository;
    }

    public async Task<int> Handle(CreateOrderCommand request, CancellationToken
cancellationToken) {
        var order = new Order();
        foreach (var price in request.Items) {
            order.AddItem(price);
        }
        await _repository.AddAsync(order);
        return order.Id;
    }
}
```

---

**3. Infrastructure Layer**

- Implements interfaces defined in the Application layer
- Contains EF Core, file storage, external services, etc.

**Example: EF Core Repository**

```
public class OrderRepository : IOrderRepository {
    private readonly AppDbContext _context;

    public OrderRepository(AppDbContext context) {
        _context = context;
    }

    public async Task AddAsync(Order order) {
        _context.Orders.Add(order);
        await _context.SaveChangesAsync();
    }
}
```

---

**4. Presentation Layer (Web API)**

- API controllers, input validation, authentication
- Calls application use cases via MediatR or services

**Example: Controller**

```
[ApiController]
[Route("api/[controller]")]
public class OrdersController : ControllerBase {
    private readonly IMediator _mediator;

    public OrdersController(IMediator mediator) {
        _mediator = mediator;
    }

    [HttpPost]
    public async Task<IActionResult> Create([FromBody] CreateOrderCommand
command) {
        var id = await _mediator.Send(command);
        return Ok(new { orderId = id });
    }
}
```

---

### 🔹 Project Folder Structure

```
/src
|-- MyApp.Domain
|   |-- Entities
|   \-- ValueObjects
|
|-- MyApp.Application
|   |-- Interfaces
|   \-- UseCases
|
|-- MyApp.Infrastructure
|   \-- Repositories
|
|-- MyApp.API
    |-- Controllers
    \-- Program.cs / Startup.cs
```

---

### 🔹 Dependency Flow Diagram

```
UI (Web API)
   |
Application Layer (Use Cases)
   |
```

```
Domain Layer (Entities)
   ^
   |
Infrastructure Layer (EF Core, etc.)
```

Note: Only **interfaces flow inward**, actual dependencies in outer layers depend on abstractions.

---

### 🔹 Key Principles

- **Dependency Inversion:** High-level modules shouldn't depend on low-level modules.
- **Use Cases = Business Processes**
- **Infrastructure is a plugin**, replaceable/swappable
- **No Framework in Core Layers:** Domain/Application layers have zero idea of EF Core, ASP.NET, etc.

---

### 🆚 Interview Tips

- Be ready to whiteboard the flow of a request: Controller → Use Case → Repository → Domain
- Emphasize testability and decoupling
- Mention MediatR for CQRS pattern
- Stress on how infrastructure can change without affecting core logic

---

Need a printable PDF, diagram, or mock Q&A session next? Say the word!