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;
    }
}
```

🔹 MediatR + CQRS in Clean Architecture

✔️What is MediatR?

MediatR is a .NET library that helps implement **Mediator pattern**. It decouples senders and receivers, and is commonly used to implement **CQRS (Command Query Responsibility Segregation)**.

⚠️What is CQRS?

CQRS splits operations into:

- **Commands**: mutate data (e.g., `CreateOrderCommand`)
- **Queries**: fetch data (e.g., `GetOrderByIdQuery`)

🔹 Example: Command Handler with MediatR

Command

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

Handler

```
public class CreateOrderHandler : IRequestHandler<CreateOrderCommand, int> {
    private readonly IOrderRepository _repo;
    public CreateOrderHandler(IOrderRepository repo) {
        _repo = repo;
    }

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

🔹 **Example: Query Handler with MediatR**

Query

```
public class GetOrderByIdQuery : IRequest<OrderDto> {
    public int Id { get; set; }
}
```

Handler

```
public class GetOrderByIdHandler : IRequestHandler<GetOrderByIdQuery, OrderDto>
{
    private readonly IOrderRepository _repo;

    public GetOrderByIdHandler(IOrderRepository repo) {
        _repo = repo;
    }

    public async Task<OrderDto> Handle(GetOrderByIdQuery request,
CancellationToken cancellationToken) {
        var order = await _repo.GetByIdAsync(request.Id);
        return new OrderDto {
            Id = order.Id,
            TotalAmount = order.TotalAmount
        };
    }
}
```

🔹 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();
    }

    public async Task<Order> GetByIdAsync(int id) =>
        await _context.Orders.FindAsync(id);
}
```

🔹 Presentation Layer (Web API)

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

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 });
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetById(int id) {
        var result = await _mediator.Send(new GetOrderByIdQuery { Id = id });
        return Ok(result);
    }
}
```

🔹 Folder Structure (With CQRS)

```
/src
|-- MyApp.Domain
|   \-- Entities
|
|-- MyApp.Application
|   |-- Commands
|   |-- Queries
|   \-- Interfaces
|
|-- MyApp.Infrastructure
|   \-- Repositories
|
|-- MyApp.API
    \-- Controllers
```

🔹 Benefits of MediatR + CQRS

- Keeps controllers thin
- Decouples request handling logic
- Makes testing individual requests easier
- Encourages separation between reads and writes

🆚Interview Tips

- Explain how CQRS helps organize code into responsibilities
- Show how MediatR avoids direct service calls in controllers
- Emphasize testability and how handler logic is easily mocked

---

Need a diagram or want to add validation + logging to the pipeline? Say the word!