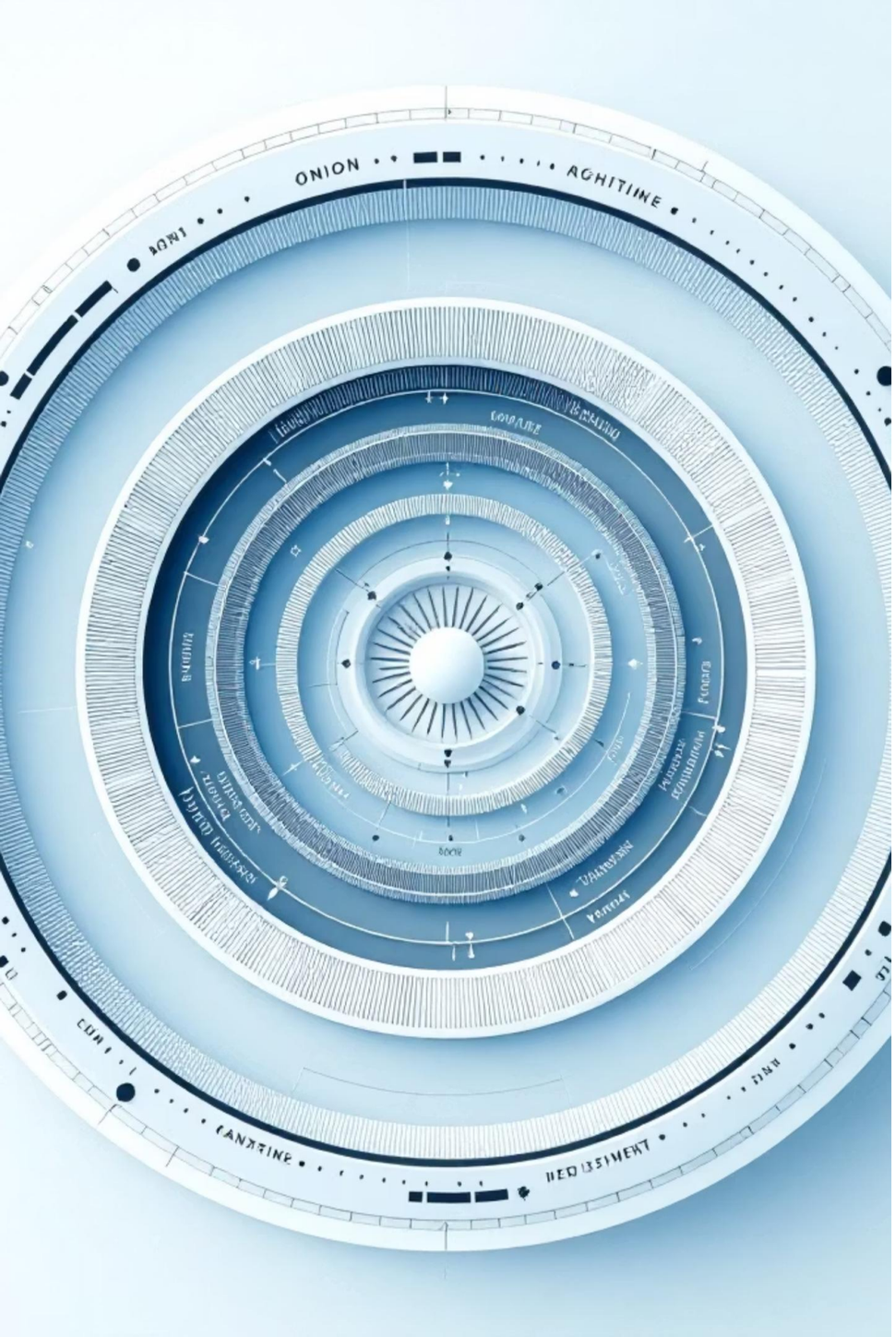


Onion Architecture in Modern Software Engineering

Exploring a robust architectural pattern for building maintainable and scalable software systems

Presented by: Müslüm Agah - 20210808042

What is Onion Architecture?

Onion Architecture is a software design pattern that structures applications into concentric layers, where each layer represents a specific responsibility. This pattern promotes loose coupling, testability, and maintainability by enforcing strict dependency rules.

Software architecture patterns are fundamental to modern software systems because they provide blueprints for organizing code, managing complexity, and ensuring systems can evolve over time. Without proper architectural patterns, software becomes tightly coupled, difficult to test, and nearly impossible to maintain as requirements change.

Key Problems It Solves

- Tight coupling between components
- Difficulty in testing business logic
- Poor maintainability and scalability
- Hard dependency on external frameworks
- Business logic scattered across layers

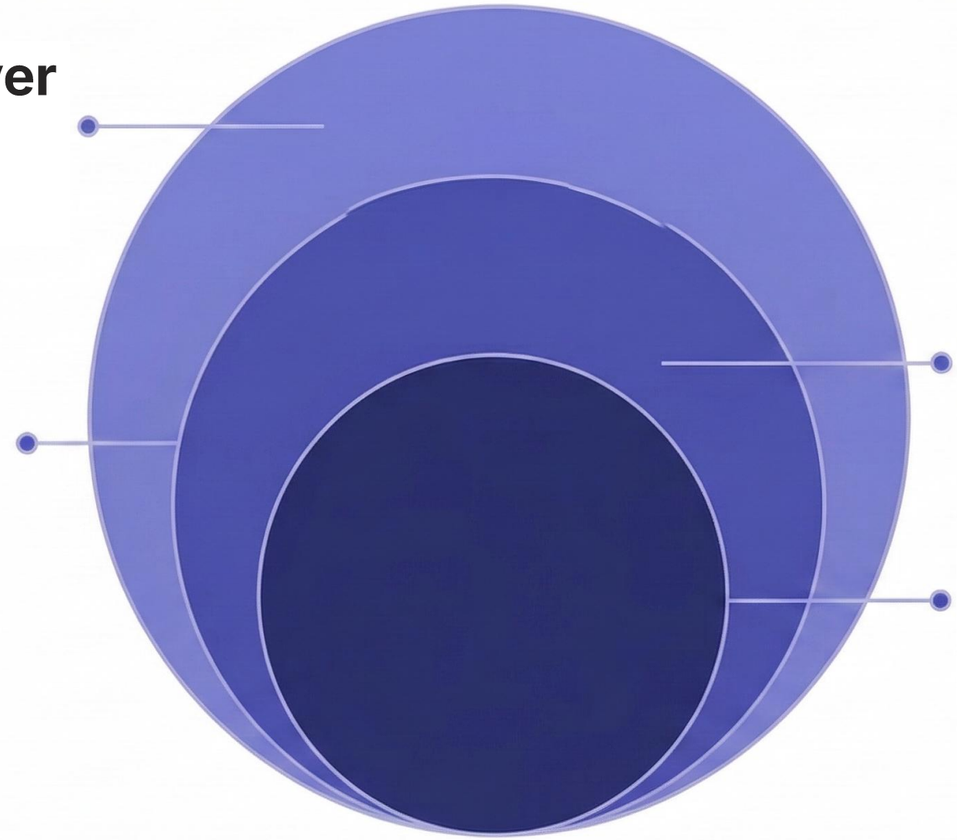
Core Concepts: The Layered Structure

Presentation Layer

Handles user interface, API controllers, and user input

Application Layer

Use cases, orchestration, and interfaces



Infrastructure Layer

Data access, external services, and adapters

Domain Layer

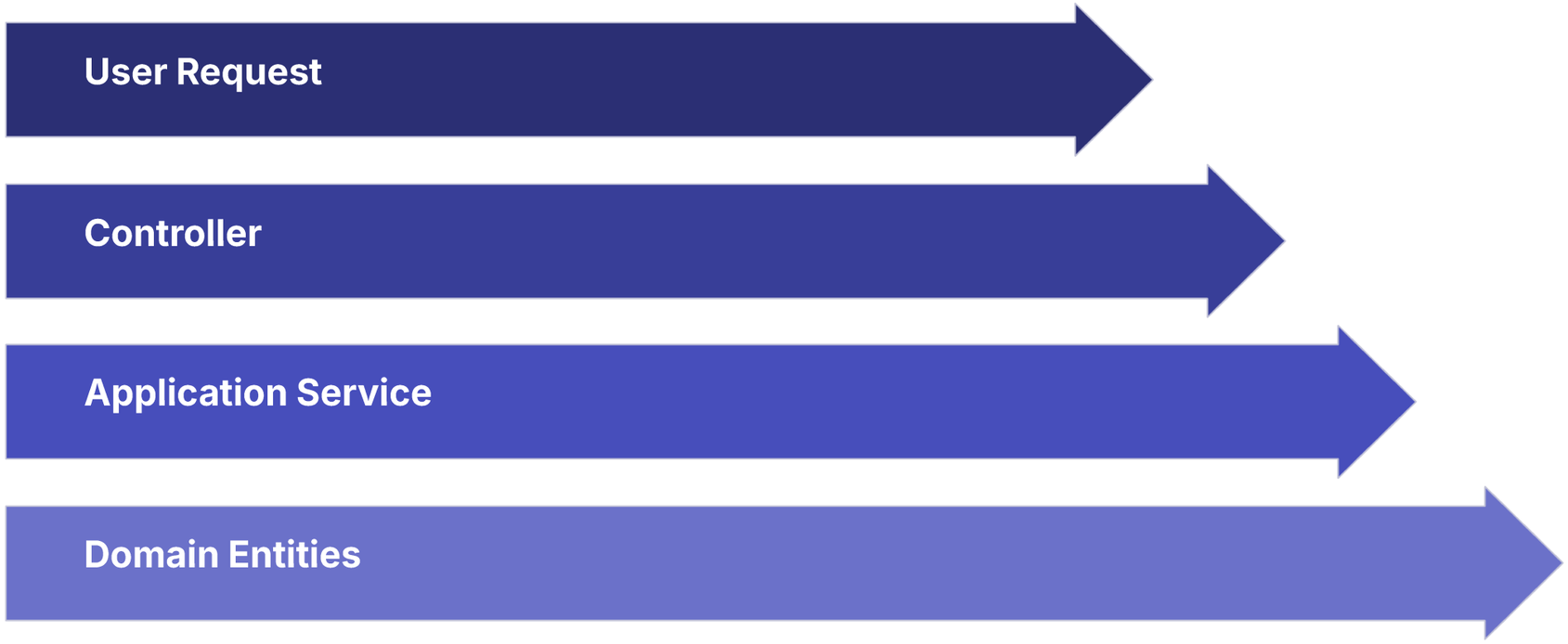
Core business logic and entities

Technical Principles & Design Patterns

Onion Architecture leverages fundamental software design principles to create flexible, testable systems.

- **Dependency Inversion Principle**
- **Separation of Concerns**
- **Testability**
- **Maintainability**

Request Flow Through Layers



Interfaces and repository patterns decouple infrastructure from business logic, allowing different implementations without changing core domain code.

Real-World Example: E-Commerce Order System

Let's examine how Onion Architecture structures a common e-commerce scenario:

Domain Layer

`Order` entity with business rules for validation, calculation of totals, and order state management

Application Layer

`OrderService` implementing use cases like place order, cancel order, and calculate shipping

Repository Interface

`IOrderRepository` defining contract for data access operations without implementation details

Infrastructure Layer

`OrderRepository` implementing database persistence using Entity Framework or similar ORM

Presentation Layer

`OrderController` handling HTTP requests and responses via REST API endpoints

Common Technology Stacks

- ASP.NET Core with Dependency Injection
- Spring Boot with Java
- Node.js with NestJS framework
- Entity Framework for data access
- PostgreSQL/MySQL databases
- RESTful API design patterns

Practical Importance & Conclusion

Why Onion Architecture Matters

Better Testability

Isolate business logic from infrastructure for fast, reliable unit tests without external dependencies

Scalable Architecture

Add new features and infrastructure without modifying core business logic

Long-term Maintainability

Clear separation reduces technical debt and makes refactoring safer and more predictable

Domain-Driven Design

Complements DDD principles by keeping domain models pure and focused on business concerns

---

Key Benefits Summary

Onion Architecture provides a proven approach to building software systems that remain maintainable and adaptable as requirements evolve. By enforcing dependency inversion and separation of concerns, it creates architectures where business logic remains protected from external changes.

References: Jeffrey Palermo (Onion Architecture blog posts) • Microsoft Architecture Documentation • Martin Fowler's Patterns of Enterprise Application Architecture