Onion Architecture: Comprehensive Guide

1. Introduction

Onion Architecture is a design pattern proposed by Jeffrey Palermo that emphasizes a layered approach to application architecture, where dependencies are directed towards the core, promoting a clean separation of concerns. It facilitates maintainability, testability, and scalability.

2. Core Principles

2.1. Separation of Concerns

Each layer of the Onion Architecture has distinct responsibilities, allowing for clear separation of concerns. This leads to improved code maintainability and testability.

2.2. Dependency Inversion

The architecture promotes dependency inversion, meaning higher-level modules should not depend on lower-level modules but instead rely on abstractions.

3. Onion Architecture Layers

3.1. Core Layer

- **Entities**: Represents the domain model, encapsulating the business logic.
- **Interfaces**: Contains interfaces for repositories and services, defining contracts.

3.2. Application Layer

- **Services**: Contains application logic and orchestration of use cases.
- **DTOs**: Data Transfer Objects are used to transfer data between layers without exposing the core entities.

3.3. Infrastructure Layer

- **Repositories**: Implements data access logic and interacts with the database.
- **Data Context**: Configures the database context and manages data operations.

3.4. Presentation Layer

- **API Controllers**: Exposes endpoints for client applications to interact with the application.

- **Middleware**: Custom middleware components for cross-cutting concerns such as exception handling.

4. Dependency Injection Configuration

Dependency Injection (DI) is a fundamental aspect of Onion Architecture. It allows for decoupling and easier unit testing by injecting dependencies at runtime.

Example Configuration

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseInMemoryDatabase("DemoDb"));

    services.AddScoped<IOwnerRepository, OwnerRepository>();
    services.AddScoped<IVehicleRepository, VehicleRepository>();
    services.AddControllers();

}
```

5. Middleware for Global Exception Handling

Custom middleware can be used to handle exceptions globally, providing a centralized mechanism for logging and returning appropriate responses.

Example Middleware

```
public class GlobalExceptionHandler

{

    private readonly RequestDelegate _next;

    public GlobalExceptionHandler(RequestDelegate next)
    {
        _next = next;

    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
```

```
        }
        catch (Exception ex)
        {
            // Log exception and return error response
        }
    }
}
```

6. Internal Sealed Classes

Using internal sealed classes within the core layer helps to encapsulate functionality while preventing inheritance. This can enhance security and enforce strict design.

7. Service and Repository Managers

Service managers orchestrate operations across repositories, ensuring that business logic remains cohesive and transactional.

Example Structure

```
public class OwnerService : IOwnerService
{

    private readonly IOwnerRepository _ownerRepository;

    public OwnerService(IOwnerRepository ownerRepository)
    {

        _ownerRepository = ownerRepository;

    }

    public Owner GetOwnerById(int id) => _ownerRepository.GetById(id);
}
```

8. Data Transfer Objects (DTOs)

DTOs are used to transfer data between layers, minimizing the amount of data sent over the network and preventing exposure of the core entities.

Example DTO

```
public class OwnerDTO
{
    public int Id { get; set; }
```

```
    public string Name { get; set; }
}
```

9. Common Patterns and Principles

9.1. Repository Pattern

Encapsulates data access logic, promoting a separation between the domain and data mapping layers.

9.2. Unit of Work Pattern

Manages multiple repositories as a single transaction, ensuring that all changes are committed or rolled back together.

9.3. Specification Pattern

Encapsulates business rules in a reusable manner, allowing for complex queries without cluttering the repository.

9.4. CQRS (Command Query Responsibility Segregation)

Separates commands (writes) from queries (reads), allowing for optimized data retrieval and modification.