ASP.NET Core - True Ultimate Guide

Section 24: Clean Architecture - Notes

Clean Architecture in ASP.NET Core

Clean Architecture, also known as Onion Architecture, is a software design principle that emphasizes separation of concerns, testability, and maintainability. It achieves this by organizing your application into layers, with each layer having a specific responsibility and dependency direction.

Core Layers

1. **Domain Layer (Core)**
    - **Purpose:** The heart of your application, containing your business rules and domain models.
    - **Contents:**
        - Entities: Represent the core concepts of your domain (e.g., Person, Order, Product).
        - Value Objects: Immutable objects representing concepts like Money, Address, or EmailAddress.
        - Domain Services: Encapsulate complex business logic or operations that involve multiple entities.
        - Interfaces (Contracts): Define the contracts for repositories and other dependencies.
    - **Dependencies:** None. The domain layer is independent of any external infrastructure or frameworks.

2. **Application Layer**
    - **Purpose:** Orchestrates the use cases of your application.
    - **Contents:**
        - Use Cases (Application Services): Implement the high-level use cases or operations of your system (e.g., CreatePerson, GetPersonById).
        - DTOs (Data Transfer Objects): Represent data structures used for communication between layers.
        - Interfaces (Contracts): Define the contracts for infrastructure services (e.g., repositories, email services).
    - **Dependencies:** Depends on the Domain Layer.

3. **Infrastructure Layer**
    - **Purpose:** Implements the technical details of how your application interacts with external systems (databases, file systems, email services, etc.).
    - **Contents:**
        - Repositories: Implement the data access logic for your entities.
        - Services: Implement the interfaces defined in the application layer for interacting with external systems (e.g., EmailService, FileStorageService).
    - **Dependencies:** Depends on the Application Layer and any external libraries or frameworks needed for infrastructure tasks.

4. **Presentation Layer (UI)**
    - **Purpose:** Handles user interaction and presentation logic.
    - **Contents:**
        - Controllers: Handle HTTP requests, interact with use cases, and return views or API responses.
        - Views: Render the user interface.
        - View Models: Shape data for presentation in views.
    - **Dependencies:** Depends on the Application Layer.

5. **Tests**
    - **Purpose:** Ensures the correctness of your application's behavior.
    - **Contents:**
        - Unit Tests: Test individual units of code (e.g., domain models, services) in isolation.
        - Integration Tests: Test the interaction between multiple components.
        - End-to-End Tests: Test the entire application flow from the user's perspective.

Dependency Direction:

- **Inner Layers to Outer Layers:** Dependencies flow from the inner layers (Domain) to the outer layers (Presentation).
- **Abstraction:** Outer layers depend on abstractions (interfaces) defined in the inner layers. This allows you to easily swap implementations in the outer layers without affecting the core business logic.

Sample Code Implementation (Persons Records Management)

Let's illustrate Clean Architecture using a simplified example of managing person records.

```
// Domain Layer (Core)
public class Person
{
    public Guid PersonId { get; set; }
    public string Name { get; set; }
    // ... other properties
}

public interface IPersonsRepository
{
    Task<Person> AddPerson(Person person);
    Task<List<Person>> GetAllPersons();
    // ... other CRUD operations ...
}

// Application Layer
public class PersonDto { /* ... */ } // DTO for transferring person data

public interface IPersonsService
{
    Task<PersonDto> CreatePerson(PersonDto personDto);
    Task<List<PersonDto>> GetAllPersons();
    // ... other operations ...
}

public class PersonsService : IPersonsService
{
```

```
    private readonly IPersonsRepository _personsRepository;

    public PersonsService(IPersonsRepository personsRepository)
    {
        _personsRepository = personsRepository;
    }

    public async Task<PersonDto> CreatePerson(PersonDto personDto)
    {
        // Validation, mapping, etc.
        var person = new Person { /* ... map from DTO ... */ };
        var createdPerson = await _personsRepository.AddPerson(person);
        return createdPerson.ToDto(); // Map back to DTO
    }

    // ... other methods ...
}

// Infrastructure Layer
public class PersonsRepository : IPersonsRepository
{
    private readonly MyDbContext _dbContext;

    public PersonsRepository(MyDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task<Person> AddPerson(Person person)
    {
        _dbContext.Persons.Add(person);
```

```
        await _dbContext.SaveChangesAsync();
        return person;
    }

    // ... other methods ...
}

// Presentation Layer (UI) - Controller
public class PersonsController : Controller
{
    private readonly IPersonsService _personsService;

    public PersonsController(IPersonsService personsService)
    {
        _personsService = personsService;
    }

    [HttpPost]
    public async Task<IActionResult> Create(PersonDto personDto)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        var createdPerson = await _personsService.CreatePerson(personDto);
        return CreatedAtAction(nameof(GetPersonById), new { id = createdPerson.PersonId },
createdPerson);
    }

    // ... other actions ...
}
```

Explanation:

- The Domain layer defines the core Person entity and the IPersonsRepository interface.
- The Application layer defines the IPersonsService interface and the PersonsService implementation that uses the repository to perform CRUD operations.
- The Infrastructure layer contains the PersonsRepository that implements the repository interface and interacts with the database.
- The Presentation layer has the PersonsController that handles requests, uses the PersonsService, and returns appropriate responses.

Notes

- **Separation of Concerns:** Each layer has a distinct responsibility.
- **Dependency Direction:** Dependencies flow inwards, towards the Domain layer.
- **Abstractions:** Outer layers depend on abstractions (interfaces) defined in inner layers.
- **Testability:** Each layer can be tested in isolation using mocks or stubs for its dependencies.
- **Maintainability:** Changes to one layer have minimal impact on other layers.
- **Flexibility:** You can easily swap out implementations in the outer layers (e.g., change the database provider) without affecting the core business logic.

Key points to remember

Clean Architecture in ASP.NET Core

- **Separation of Concerns:** Decouples the different parts of your application into well-defined layers.
- **Dependency Inversion Principle (DIP):** Inner layers define abstractions (interfaces), outer layers depend on these abstractions, leading to loose coupling.
- **Testability:** Each layer can be tested in isolation using mocks or stubs.
- **Maintainability:** Easier to modify and extend the application as requirements evolve.
- **Flexibility:** You can swap out implementations in outer layers without affecting the core business logic.

Layers

1. **Domain (Core):**
    - Contains entities, value objects, and domain services.
    - Defines interfaces for repositories and other dependencies.
    - **No external dependencies.**
2. **Application:**
    - Contains use cases (application services) that orchestrate business logic.
    - Defines DTOs (Data Transfer Objects) for communication between layers.
    - Defines interfaces for infrastructure services (e.g., repositories).
    - **Depends on the Domain layer.**
3. **Infrastructure:**
    - Contains implementations of repositories, services for interacting with external systems (e.g., email, database).
    - **Depends on the Application layer and external libraries/frameworks.**
4. **Presentation (UI):**
    - Contains controllers, views, and view models.
    - Handles user interaction and presentation logic.
    - **Depends on the Application layer.**
5. **Tests:**
    - Contains unit tests, integration tests, and end-to-end tests.
    - Ensures the correctness of each layer and the entire application.

Benefits

- **Improved Maintainability:** Changes are isolated to specific layers.
- **Testability:** Each layer is easily testable in isolation.
- **Flexibility:** Swapping implementations in outer layers doesn't affect the core.
- **Focus on Business Logic:** The domain layer is at the center, emphasizing the core of your application.

Interview Tips

- **Explain the Layers:** Be able to clearly explain the purpose of each layer and how they interact.
- **Dependency Direction:** Emphasize that dependencies flow inwards towards the Domain layer.
- **Abstractions:** Highlight the importance of using interfaces to achieve loose coupling.
- **Real-World Scenarios:** Discuss how you've used or would use Clean Architecture in a project.
- **Benefits:** Articulate the advantages of Clean Architecture in terms of maintainability, testability, and flexibility.

Remember:

- **Trade-offs:** Clean Architecture adds some complexity, so consider if it's appropriate for your project's size and requirements.
- **Focus on the Domain:** The domain layer should be the most important and stable part of your application.
- **Continuous Refactoring:** As your application evolves, continuously refactor to maintain the separation of concerns and keep your code clean.