Contents

[Domain 9](#_Toc202702409)

[Entities 9](#_Toc202702410)

[Value Objects 9](#_Toc202702411)

[Aggregates and Aggregate Roots 9](#_Toc202702412)

[Repository Interfaces 9](#_Toc202702413)

[Domain Services 9](#_Toc202702414)

[Domain Events 9](#_Toc202702415)

[Specifications 9](#_Toc202702416)

[Infrastructure 9](#_Toc202702417)

[Repositories: Repository implementations 12](#_Toc202702418)

[Persistence: Database contexts, ORM, migration scripts 12](#_Toc202702419)

[Logging: Logging services (e.g., Serilog, NLog) 12](#_Toc202702420)

[Frameworks: Framework-specific code like DI and application setup 12](#_Toc202702421)

[File/Cloud Storage: File storage systems like AWS S3, Azure Blob 12](#_Toc202702422)

[ExternalServices (optional): Integrations with third-party services like email, payments, etc. 12](#_Toc202702423)

In a Clean Architecture solution, **interfaces for repositories** typically go in the **Domain** layer (often the **Core** or **Entities** project), while the **implementations** of those interfaces should be placed in the **Infrastructure** layer.

Here's a more detailed breakdown of how this can be structured:

1. **Domain Layer (Core or Entities Project)**:
   * Contains the **repository interfaces**.
   * These interfaces should be defined in a way that reflects the domain's requirements.
   * They don't depend on any infrastructure or external frameworks (i.e., they only specify the **contract**).
2. **Infrastructure Layer**:
   * Contains the **implementations** of the repository interfaces.
   * The implementation would use tools like Entity Framework, Dapper, or other data access methods.
   * The **infrastructure layer** is where you would put the actual code that interacts with databases, file systems, etc.
3. **Application Layer (Optional)**:
   * This is where the use cases (services, command handlers, etc.) reside.
   * The application layer depends on the **domain layer** and often uses interfaces from the domain to achieve business logic.

**Folder Structure Example:**

* **Domain**
  + IRepository.cs
  + IUnitOfWork.cs
  + Entities/SomeEntity.cs
* **Infrastructure**
  + Repositories/RepositoryImplementation.cs
  + Repositories/SomeEntityRepository.cs

**Why keep them separate?**

* **Separation of concerns**: Interfaces in the domain layer represent abstractions that your application needs, whereas the implementations in the infrastructure layer are dependent on external systems like databases, files, etc.
* **Testability**: By keeping the interfaces in the domain layer and the implementations in the infrastructure layer, you can easily mock the interfaces in unit tests, without worrying about the implementation details.
* **Decoupling**: This separation allows you to replace infrastructure components (e.g., change database implementation) without affecting the core business logic.

**To Answer Your Question:**

**No**, the interfaces for repositories should **not** go in the same folder as the classes that implement them. Keep interfaces in the **Domain** layer and implementations in the **Infrastructure** layer.

Ask ChatGPT

**You said:**

what software architect experts recomment this?

**ChatGPT said:**

The structure of Clean Architecture, including where to place repository interfaces and their implementations, is widely supported and recommended by many prominent software architecture experts. The design principles around Clean Architecture have been primarily popularized by **Robert C. Martin** (Uncle Bob), but also have strong endorsements from various other architects and developers.

**Key Experts Who Recommend This Approach:**

1. **Robert C. Martin (Uncle Bob)**:
   * **Clean Architecture** is a framework created and championed by Uncle Bob, and his book *"Clean Architecture: A Craftsman's Guide to Software Structure and Design"* is foundational to understanding the structure.
   * Uncle Bob emphasizes **separation of concerns** and **dependency inversion** (i.e., high-level modules should not depend on low-level modules, but both should depend on abstractions).
   * His work guides developers to place **interfaces in the domain layer** and **implementations in the infrastructure layer**, ensuring that core business logic is decoupled from the details of data access and external systems.
2. **Martin Fowler**:
   * Martin Fowler, a well-known author and software architect, has written extensively on design patterns, architecture, and clean code. He emphasizes **separation of concerns** and **layers of abstraction** in software systems.
   * In his articles and books (such as *"Patterns of Enterprise Application Architecture"*), Fowler advocates for clean separation between different parts of the application and suggests keeping domain logic isolated from infrastructure concerns. This is essentially what Clean Architecture proposes.
3. **Eric Evans**:
   * Eric Evans, in his book *"Domain-Driven Design: Tackling Complexity in the Heart of Software"*, advocates for **modeling the domain** and **focusing on business logic**. While Domain-Driven Design (DDD) isn’t strictly the same as Clean Architecture, many of the principles align—especially the idea of **separating domain logic** from infrastructure concerns.
   * In DDD, **repositories** are typically defined as abstractions within the **domain** layer, and the **implementations** are left to the infrastructure layer.
4. **Micah Silverman** (Clean Code Practitioner):
   * In his blog posts and talks, Micah Silverman often highlights the importance of creating testable, maintainable, and flexible architectures. He supports a clean separation of concerns and follows the principles outlined in Clean Architecture.
   * He specifically mentions the need for **interfaces in the domain layer**, as it promotes easier testing and helps decouple business logic from persistence concerns.
5. **Jeffrey Palermo**:
   * Jeffrey Palermo, a Clean Architecture advocate, provides numerous resources and insights on how to structure solutions according to Clean Architecture principles.
   * He emphasizes **domain-driven design** and **dependency inversion**, both of which support separating repository interfaces (in the domain layer) from their implementations (in the infrastructure layer).
6. **Jason Taylor**:
   * Jason Taylor has written about Clean Architecture in .NET applications, providing examples and tutorials. In his examples, he clearly demonstrates how repository interfaces belong in the **Core** (or **Domain**) layer, while the implementations should reside in the **Infrastructure** layer.
   * His approach is widely adopted in .NET communities for structuring applications in a scalable and maintainable way.

**Key Concepts That Drive This Recommendation:**

* **Separation of Concerns**: The idea is that the domain logic should not be concerned with the specifics of infrastructure or data access. The **repository interface** is an abstraction that allows the business logic to remain isolated from how data is actually stored or retrieved.
* **Dependency Inversion Principle (DIP)**: One of the five SOLID principles, this states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. Repository interfaces represent abstractions, while implementations represent low-level details, which should live in the infrastructure layer.
* **Testability and Flexibility**: Keeping interfaces in the domain layer and implementations in the infrastructure layer makes testing easier. You can mock the repository interface in unit tests without worrying about the underlying data access logic or dependencies. Also, this allows you to swap out implementations (e.g., changing databases or storage mechanisms) without changing your core business logic.

**Practical Examples:**

* **Clean Architecture in .NET**: When building applications in .NET using Clean Architecture (e.g., with ASP.NET Core), the **repository interfaces** will be placed in the **Core** project (representing the domain), and the **implementations** will be placed in the **Infrastructure** project. This structure is consistently recommended in tutorials, books, and online courses about Clean Architecture in .NET.
* **Microservices**: In a microservices architecture, **service interfaces** (representing the business operations) are typically placed in the domain layer, with implementations (e.g., database, messaging, etc.) placed in the infrastructure layer, ensuring the service logic remains focused on business concerns.

**Conclusion:**

The structure of placing repository **interfaces** in the **domain** layer and their **implementations** in the **infrastructure** layer is **strongly recommended** by **Robert C. Martin** and other leading software architects like **Martin Fowler** and **Eric Evans**. This is a widely accepted approach in Clean Architecture, and it ensures that your code is maintainable, testable, and decoupled. It helps in keeping your domain logic focused on the business needs while allowing flexibility in how data is persisted.

**You said:**

Do they specifically recommend: Domain folder for repository interfaces, and Infrastructure folder for repository implementation classes?

**ChatGPT said:**

While the specific names of folders (like **"Domain"** and **"Infrastructure"**) may vary depending on the context, the **concept** of **separating repository interfaces** (in the **Domain** layer) from **implementations** (in the **Infrastructure** layer) is strongly recommended by Clean Architecture advocates, including **Robert C. Martin (Uncle Bob)**, **Martin Fowler**, and other experts. The actual folder names can differ, but the **separation of concerns** and **layered approach** is consistent.

**Here's a breakdown of what these experts recommend in terms of structure:**

**1. Robert C. Martin (Uncle Bob):**

In his book *"Clean Architecture: A Craftsman's Guide to Software Structure and Design"*, Uncle Bob provides a high-level guideline rather than specifying folder names. However, he does emphasize the **separation of concerns** and the **dependency rule**, which is foundational to the Clean Architecture style.

* **Domain**: Contains the core business logic and abstractions, including repository **interfaces**.
* **Infrastructure**: Contains **implementations** of those abstractions, such as data access and external system communication.

He doesn't explicitly say "place your repository interfaces in a folder named 'Domain,'" but the design principles clearly support that idea. The **Domain** is where your business rules (including interfaces for persistence) should live, and the **Infrastructure** is where the implementation details (like database access) should be.

**2. Martin Fowler:**

Martin Fowler focuses on **separation of concerns**, emphasizing the need for different layers (such as the **domain layer**, **application layer**, and **infrastructure layer**) to handle different responsibilities.

* **Repository Interfaces**: These should be placed in the **Domain** or **Core** layer, as they define the abstraction without knowledge of specific data access technologies.
* **Repository Implementations**: These should reside in the **Infrastructure** layer, where they depend on external systems like databases, file systems, etc.

Again, he doesn't necessarily specify folder names, but he advocates the concept of keeping domain-related abstractions separate from implementation details, which naturally leads to the organization you're asking about.

**3. Eric Evans (Domain-Driven Design):**

In **Domain-Driven Design (DDD)**, **repository interfaces** are often part of the **domain layer** since repositories represent **abstractions** for accessing domain objects. Implementations of these repositories should be located in the **infrastructure** or **persistence** layer.

Evans doesn't explicitly call out folder names either, but his guidance on DDD closely aligns with Clean Architecture, recommending that abstractions (like repositories) live in the **core** (or **domain**) layer and are implemented in the **infrastructure** layer.

**4. Jason Taylor:**

In the .NET ecosystem, Jason Taylor has written extensively about applying Clean Architecture patterns to .NET applications. His example projects use the following structure:

* **Core** or **Domain** layer: Contains repository **interfaces** and business logic.
* **Infrastructure** layer: Contains the **implementations** of those repository interfaces (e.g., using Entity Framework, Dapper, or other data access strategies).

In his tutorials and examples, he specifically organizes the code in the **Core** layer (which is the same as Domain) for interfaces and in the **Infrastructure** layer for implementations. This pattern is directly in line with Clean Architecture principles.

**To Directly Answer Your Question:**

No, **they don't necessarily specify "Domain" folder for repository interfaces** and **"Infrastructure" folder for repository implementation classes** by name in their writings. However, **the pattern and reasoning behind the organization** is very clear:

* **Repository interfaces** should live in the **Domain** or **Core** layer, representing abstractions of business logic.
* **Repository implementations** should live in the **Infrastructure** layer, where the specifics of interacting with databases or external systems are managed.

The **"Domain"** and **"Infrastructure"** folder names are widely adopted in various Clean Architecture resources and tutorials because they are clear and intuitive. However, the exact names may vary slightly in different contexts, but the **layered separation** of concerns is the critical principle they all advocate.

**Practical Folder Structure Example:**

Here’s how the folder structure could look in a typical Clean Architecture solution:

pgsql

CopyEdit

/MyApp

/Core (or Domain)

/Entities

- SomeEntity.cs

/Interfaces

- IRepository.cs <-- Repository Interface

- IUnitOfWork.cs

/Services

- SomeService.cs

/Infrastructure

/Data

- RepositoryImplementation.cs <-- Repository Implementation

- SomeDbContext.cs

/Application

- SomeUseCase.cs

/WebAPI (or UI)

- Controllers/SomeController.cs

**Conclusion:**

While **Robert C. Martin**, **Martin Fowler**, and **Eric Evans** don’t explicitly state that the folder names should be **"Domain"** and **"Infrastructure"**, they do recommend the **concept** of separating repository **interfaces** from their **implementations**, which naturally leads to the organization you're asking about. In practice, **repository interfaces** typically go in the **Domain/Core** layer, and their **implementations** go in the **Infrastructure** layer.

**You said:**

what other code would be appropriate to go into the Domain layer, and what other code would be appropriate to go into the Infrastructure layer according to principles recommended by Fowler, Martin and Evans?

**ChatGPT said:**

Great question! Understanding what code goes into the **Domain** and **Infrastructure** layers is key to applying Clean Architecture and principles from **Martin Fowler**, **Robert C. Martin (Uncle Bob)**, and **Eric Evans**. Let’s break this down further.

**Domain Layer (Core)**

The **Domain** layer is the heart of your application and should contain all business-related logic and domain abstractions. Code in this layer should be **purely business-oriented** and **independent** of any specific frameworks, databases, or external services. This helps keep the core logic flexible, testable, and reusable.

**What Goes in the Domain Layer?**

1. **Entities**:
   * These are the core business objects that represent key concepts in your domain.
   * They often have rich behavior (methods) associated with them, but they don't know anything about persistence mechanisms.
   * Example: Customer, Order, Product, Invoice.

csharp

CopyEdit

public class Customer

{

public string Name { get; set; }

public DateTime CreatedAt { get; set; }

public void UpdateCustomerDetails(string name)

{

if (string.IsNullOrEmpty(name)) throw new ArgumentException("Name is required.");

Name = name;

}

}

1. **Value Objects**:
   * These represent immutable types that don’t have an identity but carry important attributes that should be treated as a whole.
   * Example: Money, Address, Email, PhoneNumber.

csharp

CopyEdit

public class Money

{

public decimal Amount { get; }

public string Currency { get; }

public Money(decimal amount, string currency)

{

if (amount <= 0) throw new ArgumentOutOfRangeException("Amount must be positive.");

Amount = amount;

Currency = currency;

}

}

1. **Aggregates and Aggregate Roots**:
   * An **aggregate** is a group of related entities that are treated as a single unit. The **aggregate root** is the main entry point for accessing an aggregate.
   * Example: A Customer entity might be an aggregate root that contains related Order entities.

csharp

CopyEdit

public class Customer : IEntity

{

public Guid Id { get; set; }

public List<Order> Orders { get; set; }

public void AddOrder(Order order)

{

Orders.Add(order);

}

}

1. **Repository Interfaces**:
   * Repositories provide methods to interact with the persistence layer, but these interfaces **must be defined in the domain layer** and only represent the **abstraction** of data access.
   * Example: ICustomerRepository, IOrderRepository.

csharp

CopyEdit

public interface ICustomerRepository

{

Customer GetById(Guid id);

void Add(Customer customer);

}

1. **Domain Services**:
   * These are services that contain business logic that doesn't naturally fit within an entity or value object but is still part of the domain.
   * Example: A service that calculates tax, discounts, or shipping costs.

csharp

CopyEdit

public class TaxService

{

public decimal CalculateTax(Order order)

{

// Business logic to calculate tax

}

}

1. **Domain Events**:
   * These represent events that happen within the domain that other parts of the system might be interested in. Domain events help decouple components.
   * Example: OrderPlacedEvent, CustomerRegisteredEvent.

csharp

CopyEdit

public class OrderPlacedEvent

{

public Order Order { get; }

public DateTime OccurredAt { get; }

public OrderPlacedEvent(Order order)

{

Order = order;

OccurredAt = DateTime.UtcNow;

}

}

1. **Specifications**:
   * **Specifications** are used to encapsulate business rules and logic that can be combined, reused, and tested.
   * Example: CustomerHasValidCreditSpecification, OrderHasMinimumQuantitySpecification.

csharp

CopyEdit

public class OrderHasMinimumQuantitySpecification : ISpecification<Order>

{

public bool IsSatisfiedBy(Order order)

{

return order.Quantity >= 1;

}

}

**What Should NOT Go in the Domain Layer?**

* **External Dependencies**: Don’t put code in the Domain layer that’s dependent on external frameworks or libraries, such as database access or HTTP services.
* **UI code**: Any code related to user interfaces, such as controllers or views, should not go into the Domain layer.

**Infrastructure Layer**

The **Infrastructure** layer is responsible for implementing the technical details needed by the Domain. It contains code that **interacts with external systems** (such as databases, file systems, APIs) and implements the **abstractions** defined in the Domain layer.

**What Goes in the Infrastructure Layer?**

1. **Repository Implementations**:
   * The **Infrastructure** layer contains the concrete implementations of the repository interfaces defined in the Domain layer.
   * This layer is responsible for interacting with data sources like a database (e.g., using Entity Framework, Dapper), file system, or web APIs.

csharp

CopyEdit

public class EfCustomerRepository : ICustomerRepository

{

private readonly AppDbContext \_context;

public EfCustomerRepository(AppDbContext context)

{

\_context = context;

}

public Customer GetById(Guid id)

{

return \_context.Customers.FirstOrDefault(c => c.Id == id);

}

public void Add(Customer customer)

{

\_context.Customers.Add(customer);

\_context.SaveChanges();

}

}

1. **Persistence-related Code**:
   * The Infrastructure layer handles the **data access logic** such as database contexts, data mappers, and SQL queries.
   * Example: DbContext, DatabaseConnection, or any file system utilities.

csharp

CopyEdit

public class AppDbContext : DbContext

{

public DbSet<Customer> Customers { get; set; }

public DbSet<Order> Orders { get; set; }

}

1. **External Integrations**:
   * Code that integrates with external services or systems, such as third-party APIs, message queues, or payment gateways.
   * Example: Sending emails, interacting with external APIs (e.g., a payment provider or external authentication service).

csharp

CopyEdit

public class EmailService : IEmailService

{

public void SendEmail(string to, string subject, string body)

{

// Code to send an email

}

}

1. **Logging**:
   * Infrastructure handles logging, though logging may also occur in the **Domain** layer to track domain-specific events.
   * Example: A logging implementation using a framework like Serilog, NLog, or logging directly to a file or external service.

csharp

CopyEdit

public class SerilogLogger : ILogger

{

private readonly ILogger \_logger = Log.ForContext<SerilogLogger>();

public void LogInfo(string message)

{

\_logger.Information(message);

}

}

1. **Framework-specific Implementations**:
   * The Infrastructure layer often contains code tied to specific frameworks or libraries, such as **Entity Framework**, **ASP.NET Core**, **RabbitMQ**, **Redis**, etc.
   * Example: Dependency Injection (DI) setup, configuration classes, and application startup logic.

csharp

CopyEdit

public class DependencyInjection

{

public static void ConfigureServices(IServiceCollection services)

{

services.AddScoped<ICustomerRepository, EfCustomerRepository>();

services.AddScoped<IEmailService, EmailService>();

}

}

1. **File/Cloud Storage**:
   * If your application interacts with file storage systems (local disk, AWS S3, Azure Blob Storage, etc.), the logic for reading and writing files belongs in the Infrastructure layer.

csharp

CopyEdit

public class S3FileStorage : IFileStorage

{

public void UploadFile(string filePath)

{

// Upload file to AWS S3

}

}

**What Should NOT Go in the Infrastructure Layer?**

* **Business Logic**: Any business rules, calculations, or domain logic should not go in the Infrastructure layer. That belongs in the Domain layer.
* **Framework-agnostic code**: The goal of the Infrastructure layer is to contain code that depends on external systems, but business logic should not depend on infrastructure-specific code.

**Summary: Where Does Code Belong?**

**Domain Layer:**

* **Entities**
* **Value Objects**
* **Aggregates and Aggregate Roots**
* **Repository Interfaces**
* **Domain Services**
* **Domain Events**
* **Specifications**
* **Business rules and logic**

**Infrastructure Layer:**

* **Repository Implementations**
* **Database contexts and queries**
* **External integrations (APIs, services)**
* **Persistence (e.g., Entity Framework, Dapper)**
* **Logging**
* **File and Cloud Storage**
* **External frameworks and dependencies**

**Conclusion**

By adhering to **Clean Architecture** principles and the advice of **Fowler**, **Martin**, and **Evans**, you ensure that the **Domain** layer remains **purely business-focused**, independent of infrastructure concerns. Meanwhile, the **Infrastructure** layer handles all external system integrations and technical implementation details. This clear separation of concerns ensures your system is **flexible**, **testable**, and **maintainable** over time.

**You said:**

In the Domain layer, should the following all be subfolders to hold their various code files? Entities: Value Objects: Aggregates and Aggregate Roots: Repository Interfaces: Domain Services: Domain Events: Specifications:

**ChatGPT said:**

In a **Clean Architecture** approach, organizing your **Domain** layer into subfolders for different kinds of business logic is an excellent idea. While the exact structure is somewhat flexible based on your project’s size, complexity, and specific needs, organizing the **Domain** layer into subfolders for each type of code (like **Entities**, **Value Objects**, **Aggregates**, **Repository Interfaces**, **Domain Services**, etc.) is **highly recommended**. It helps keep the domain layer organized, clear, and maintainable as your project grows.

Here’s how you might structure the **Domain** layer in terms of subfolders:

**Recommended Domain Layer Structure:**

pgsql

CopyEdit

/MyApp.Domain

/Entities

- Customer.cs

- Order.cs

/ValueObjects

- Money.cs

- Address.cs

/Aggregates

- CustomerAggregate.cs

/Repositories

- ICustomerRepository.cs

- IOrderRepository.cs

/DomainServices

- TaxService.cs

- DiscountService.cs

/DomainEvents

- OrderPlacedEvent.cs

- CustomerRegisteredEvent.cs

/Specifications

- OrderHasMinimumQuantitySpecification.cs

- CustomerHasValidCreditSpecification.cs

**Breakdown of Each Subfolder:**

**1. Entities:**

* **What it is**: Contains **core business entities** that represent key concepts in your domain. Entities have **identity** (they can be tracked over time) and typically contain business rules (methods) relevant to them.
* **Why subfolder**: As entities are usually the most important and fundamental part of your domain model, having a dedicated folder for them keeps things clear.
* **Example**:

csharp

CopyEdit

public class Customer

{

public Guid Id { get; set; }

public string Name { get; set; }

public void UpdateName(string name)

{

if (string.IsNullOrEmpty(name))

throw new ArgumentException("Name cannot be empty.");

Name = name;

}

}

**2. Value Objects:**

* **What it is**: Contains **value objects**, which are types that don’t have a distinct identity but represent certain values in your domain (e.g., money, address). These are typically immutable.
* **Why subfolder**: Value objects usually contain logic around validation, comparison, and other rules that apply to values in your domain, making them distinct from entities and worthy of their own folder.
* **Example**:

csharp

CopyEdit

public class Money

{

public decimal Amount { get; }

public string Currency { get; }

public Money(decimal amount, string currency)

{

if (amount < 0) throw new ArgumentOutOfRangeException("Amount cannot be negative.");

Amount = amount;

Currency = currency;

}

}

**3. Aggregates and Aggregate Roots:**

* **What it is**: Aggregates are groups of related entities that should be treated as a **single unit**. The aggregate root is the main entry point for interacting with the aggregate.
* **Why subfolder**: Aggregates and their roots can be complex and often contain multiple entities or value objects, so it’s helpful to have them organized into a subfolder to distinguish them from simple entities.
* **Example**:

csharp

CopyEdit

public class CustomerAggregate : Customer

{

public List<Order> Orders { get; private set; }

public void AddOrder(Order order)

{

Orders.Add(order);

}

}

**4. Repository Interfaces:**

* **What it is**: Contains **interfaces** for repositories. These repositories provide abstractions for accessing data and should be **defined in the Domain layer**. The implementation of these interfaces belongs to the **Infrastructure layer**.
* **Why subfolder**: Having a dedicated folder for repository interfaces helps organize the domain contracts that define how business entities should be persisted or retrieved. Keeping them separate also ensures that domain logic stays independent of infrastructure.
* **Example**:

csharp

CopyEdit

public interface ICustomerRepository

{

Task<Customer> GetByIdAsync(Guid customerId);

Task AddAsync(Customer customer);

}

**5. Domain Services:**

* **What it is**: Contains **domain services**, which represent **business logic** that doesn’t naturally fit within an entity or value object. These services contain behavior relevant to the domain but are generally stateless.
* **Why subfolder**: Domain services often contain logic that ties entities or aggregates together. Grouping them into their own folder helps distinguish them from entities or value objects.
* **Example**:

csharp

CopyEdit

public class TaxService

{

public decimal CalculateTax(Order order)

{

// Tax calculation logic here

return order.TotalAmount \* 0.1m; // Example: 10% tax rate

}

}

**6. Domain Events:**

* **What it is**: Contains **domain events** that represent things that have **happened** in the domain, which other parts of the system might need to react to. Domain events help decouple the system by letting other modules subscribe to these events.
* **Why subfolder**: Domain events represent important state transitions or actions within your domain. Keeping them in their own folder makes it easy to find and manage.
* **Example**:

csharp

CopyEdit

public class OrderPlacedEvent

{

public Order Order { get; }

public DateTime Timestamp { get; }

public OrderPlacedEvent(Order order)

{

Order = order;

Timestamp = DateTime.UtcNow;

}

}

**7. Specifications:**

* **What it is**: Contains **specifications**, which encapsulate business rules that can be **composed** and **reused**. Specifications help enforce constraints or filters on entities or aggregates.
* **Why subfolder**: Specifications are a key pattern for expressing complex rules in a reusable manner. They should be in their own folder for clarity and maintainability.
* **Example**:

csharp

CopyEdit

public class CustomerHasValidCreditSpecification : ISpecification<Customer>

{

public bool IsSatisfiedBy(Customer customer)

{

return customer.CreditLimit > 0;

}

}

**Why Organizing Code into Subfolders in the**

Domain **Layer Makes Sense:**

1. **Clarity and Separation**: As your application grows, having dedicated folders for different types of domain concepts helps maintain clarity. It also makes it easier for new developers to navigate the codebase and understand where to look for specific domain logic.
2. **Cohesion**: Grouping related code into subfolders ensures that each domain concept (like entities, services, events, etc.) has a clear focus and adheres to the **Single Responsibility Principle (SRP)**.
3. **Maintainability**: As your domain logic evolves, having distinct subfolders allows you to make changes in one area without having to worry about mixing up with other areas (for example, changes in domain events or repository interfaces).
4. **Scalability**: For larger applications, this organization scales well. It’s easier to add new business concepts (like new entities, services, or specifications) and keep the structure clean and organized.

**Is this Folder Structure Mandatory?**

No, the folder structure is **not mandatory**—it’s up to you to decide based on your project’s needs. However, it’s a **best practice** to have a structure that reflects the **logical grouping** of your business logic. As your domain model grows, it’ll help to have this organization in place to avoid a cluttered and hard-to-navigate domain layer.

In smaller projects, you might not need this level of organization right away. However, as your application grows in complexity, **having well-organized subfolders** will pay off in the long run.

**Conclusion:**

For a **Clean Architecture** setup, it’s a great idea to organize your **Domain layer** into subfolders based on the type of code:

### Entities

### Value Objects

### Aggregates and Aggregate Roots

### Repository Interfaces

### Domain Services

### Domain Events

### Specifications

This structure ensures that your **domain logic remains well-encapsulated**, clear, and maintainable, and it will help keep the complexity of your business logic manageable as your system grows.

**You said:**

In the Infrastructure folder should these also be separate folders? Repository Implementations: Persistence-related Code: Logging: Framework-specific Implementations: File/Cloud Storage:

**ChatGPT said:**

Yes, it’s a great idea to **separate the code in the**

Infrastructure **layer into distinct folders** to improve maintainability, clarity, and organization. The **Infrastructure layer** deals with the external systems, databases, file storage, messaging, and other technical concerns, so keeping these concerns separate in their respective folders helps ensure that each responsibility is clear and well-encapsulated.

Here’s a suggested structure for the **Infrastructure** layer, with subfolders for each category of concern:

**Recommended Infrastructure Layer Structure:**

markdown

CopyEdit

/MyApp.Infrastructure

/Repositories

- EfCustomerRepository.cs

- EfOrderRepository.cs

/Persistence

- AppDbContext.cs

- MigrationScripts

- 20210701000000\_InitialCreate.cs

- 20210701010000\_AddOrderTable.cs

/Logging

- SerilogLogger.cs

- NLogLogger.cs

/Frameworks

- DependencyInjection.cs

- ApplicationStartup.cs

/FileStorage

- S3FileStorage.cs

- LocalFileStorage.cs

/ExternalServices

- EmailService.cs

- PaymentGatewayService.cs

**Breakdown of Each Subfolder:**

**1. Repositories:**

* **What it is**: This folder contains the **concrete implementations** of the repository interfaces defined in the **Domain** layer.
* **Why subfolder**: The repository implementations typically interact with a database or other data store, and keeping them together helps organize the data access logic.
* **Example**:

csharp

CopyEdit

public class EfCustomerRepository : ICustomerRepository

{

private readonly AppDbContext \_context;

public EfCustomerRepository(AppDbContext context)

{

\_context = context;

}

public Task<Customer> GetByIdAsync(Guid id)

{

return \_context.Customers.FirstOrDefaultAsync(c => c.Id == id);

}

public Task AddAsync(Customer customer)

{

\_context.Customers.Add(customer);

return \_context.SaveChangesAsync();

}

}

**2. Persistence:**

* **What it is**: Contains the **persistence-related code** such as **database contexts**, **ORM mappings**, and **migration scripts**.
* **Why subfolder**: Persistence-related code is central to how your application interacts with the data layer (usually a database), so it's good to keep this in one place. It also allows you to isolate database migrations and configuration in a clean way.
* **Example**:

csharp

CopyEdit

public class AppDbContext : DbContext

{

public DbSet<Customer> Customers { get; set; }

public DbSet<Order> Orders { get; set; }

}

**Migration script example**:

csharp

CopyEdit

public partial class InitialCreate : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "Customers",

columns: table => new

{

Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),

Name = table.Column<string>(type: "nvarchar(max)", nullable: false),

},

constraints: table =>

{

table.PrimaryKey("PK\_Customers", x => x.Id);

});

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable(name: "Customers");

}

}

**3. Logging:**

* **What it is**: This folder contains implementations of **logging services** for the application. This could include logging configuration and log service classes (using frameworks like **Serilog**, **NLog**, **Log4Net**, etc.).
* **Why subfolder**: Logging is a cross-cutting concern, and organizing it in its own folder helps isolate logging logic from business logic, making it easier to change logging libraries or implementations if needed.
* **Example**:

csharp

CopyEdit

public class SerilogLogger : ILogger

{

private readonly ILogger \_logger = Log.ForContext<SerilogLogger>();

public void LogInfo(string message)

{

\_logger.Information(message);

}

public void LogError(string message, Exception exception)

{

\_logger.Error(exception, message);

}

}

**4. Frameworks:**

* **What it is**: This folder contains **framework-specific implementations** like **dependency injection** setup, application startup configuration, middleware setup, and other **framework-specific infrastructure** code.
* **Why subfolder**: Framework-specific code such as DI configuration or app initialization should be separate because it's part of how the application interacts with the external environment, but it’s not part of the domain logic.
* **Example**:

csharp

CopyEdit

public static class DependencyInjection

{

public static void ConfigureServices(IServiceCollection services)

{

// Register application services

services.AddScoped<ICustomerRepository, EfCustomerRepository>();

services.AddScoped<IEmailService, EmailService>();

// Register infrastructure services

services.AddSingleton<ILogger, SerilogLogger>();

}

}

**Application Startup** example:

csharp

CopyEdit

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

DependencyInjection.ConfigureServices(services);

}

}

**5. File/Cloud Storage:**

* **What it is**: Contains **file storage** implementations for storing and retrieving files, either on the local file system or cloud storage platforms like **AWS S3**, **Azure Blob Storage**, or **Google Cloud Storage**.
* **Why subfolder**: This is a typical infrastructure concern that has its own logic and dependencies, so organizing it separately is a good practice.
* **Example**:

csharp

CopyEdit

public class S3FileStorage : IFileStorage

{

private readonly IAmazonS3 \_s3Client;

public S3FileStorage(IAmazonS3 s3Client)

{

\_s3Client = s3Client;

}

public async Task UploadFileAsync(string bucketName, string key, Stream fileStream)

{

var uploadRequest = new PutObjectRequest

{

BucketName = bucketName,

Key = key,

InputStream = fileStream

};

await \_s3Client.PutObjectAsync(uploadRequest);

}

}

**6. ExternalServices (Optional but Recommended):**

* **What it is**: Contains implementations that interact with **external services** like **email providers**, **payment gateways**, **SMS services**, or any external API or service your application integrates with.
* **Why subfolder**: This isolates external integrations from the core business logic and helps with maintaining the communication with third-party services.
* **Example**:

csharp

CopyEdit

public class EmailService : IEmailService

{

private readonly SmtpClient \_smtpClient;

public EmailService(SmtpClient smtpClient)

{

\_smtpClient = smtpClient;

}

public async Task SendEmailAsync(string to, string subject, string body)

{

var message = new MailMessage("noreply@myapp.com", to, subject, body);

await \_smtpClient.SendMailAsync(message);

}

}

**Why Should You Separate These Concerns into Different Folders?**

1. **Separation of Concerns**: The **Infrastructure** layer handles technical concerns such as data persistence, external services, logging, and file storage. By separating these responsibilities into distinct folders, you reduce the risk of mixing domain logic with technical concerns.
2. **Maintainability**: As your application grows, each part of the infrastructure can evolve independently. If you need to change how logging is done or switch file storage providers, you’ll know exactly where to go without affecting business logic.
3. **Testing**: With clear separation, you can mock or swap out infrastructure dependencies easily. For example, you can mock the EfCustomerRepository when testing domain logic or replace the real EmailService with a mock service in unit tests.
4. **Scalability**: Over time, your infrastructure will grow, and having it split into separate folders allows for scalability. You can easily add new infrastructure components, like new external integrations or data storage solutions, without disrupting the overall organization.

**Conclusion:**

It’s highly recommended to separate different concerns within the **Infrastructure layer** into dedicated subfolders such as:

### Repositories: Repository implementations

### Persistence: Database contexts, ORM, migration scripts

### Logging: Logging services (e.g., Serilog, NLog)

### Frameworks: Framework-specific code like DI and application setup

### File/Cloud Storage: File storage systems like AWS S3, Azure Blob

### ExternalServices (optional): Integrations with third-party services like email, payments, etc.

This **folder structure** makes your infrastructure code **organized**, **maintainable**, and **easier to manage** as your application scales. It also helps in keeping the **domain layer clean** and independent of technical concerns.

Get smarter responses, upload files and images, and more.

Log in

Sign up for free

Top of Form

Attach

Search

Voice

Bottom of Form

ChatGPT can make mistakes. Check impo