**Why We Need Architecture in Any Project?**

Software architecture is essential in any project because it defines the structure and interaction of components. Here’s why it is crucial:

1. **Scalability** – A well-structured architecture allows applications to scale efficiently.
2. **Maintainability** – Organized code is easier to update and extend.
3. **Performance Optimization** – Ensures optimized resource usage.
4. **Security** – Defines clear boundaries for secure data flow.
5. **Reusability** – Enables component reuse across multiple projects.

**What is N-Tier Architecture?**

N-Tier architecture is a software design pattern that separates an application into multiple layers to improve maintainability and scalability. The most common layers include:

1. **Presentation Layer** – UI/UX handling.
2. **Business Logic Layer** – Core functionality and rules.
3. **Data Access Layer** – Database interactions.
4. **Infrastructure Layer** (optional) – Handles external dependencies like logging, caching, etc.

**Example of N-Tier Architecture:**

public class ProductService

{

private readonly ProductRepository \_repository;

public ProductService(ProductRepository repository)

{

\_repository = repository;

}

public Product GetProduct(int id)

{

return \_repository.GetById(id);

}

}

**What is Onion Architecture?**

Onion Architecture is a domain-centric approach that emphasizes maintainability, testability, and flexibility by organizing layers in concentric rings.

**Key Layers in Onion Architecture:**

1. **Domain Layer** – Core business logic (Entities, Interfaces, Services).
2. **Application Layer** – Use cases and application-specific logic.
3. **Infrastructure Layer** – Database, third-party services, APIs.
4. **Presentation Layer** – UI framework (MVC, Blazor, etc.).

**Example Structure:**

public interface IProductRepository

{

Product GetById(int id);

}

public class ProductService

{

private readonly IProductRepository \_repository;

public ProductService(IProductRepository repository)

{

\_repository = repository;

}

}

**Is LINQ Slow in Execution? Understanding Deferred vs. Eager Execution**

LINQ performance depends on whether it uses **Deferred Execution** or **Eager Execution**.

**1. Deferred Execution**

LINQ queries are not executed immediately. The query is evaluated only when iterated.

var numbers = new List<int> { 1, 2, 3, 4 };

var evenNumbers = numbers.Where(n => n % 2 == 0); // Not executed yet

foreach (var num in evenNumbers) // Execution happens here

{

Console.WriteLine(num);

}

**2. Eager Execution**

Forces immediate execution using methods like .ToList(), .ToArray(), or .FirstOrDefault().

var evenNumbers = numbers.Where(n => n % 2 == 0).ToList(); // Executes immediately

**When is LINQ slow?**

* **Large datasets with complex filtering**
* **Multiple iterations over deferred queries**
* **Inefficient queries affecting database performance (e.g., missing indexes in EF Core)**

**Optimizing LINQ Performance**

* Use **compiled queries** in EF Core.
* Convert deferred execution to eager execution when needed.
* Optimize database queries by reviewing generated SQL.
* Use AsNoTracking() in EF Core when tracking is unnecessary.

By understanding deferred vs. eager execution, developers can write efficient LINQ queries that balance performance and readability.