Repository and **Service** patterns in the context of **ASP.NET Web API** development

---

1. Repository Pattern

Definition

The **Repository Pattern** is a design pattern that abstracts the data access logic and provides a clean separation between the **business logic** and **data access layer**.

It acts as an **in-memory collection interface** to the database, so your business or API layers don't need to know *how* data is stored or retrieved (SQL, Entity Framework, MongoDB, etc.).

Key Idea

> "The Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection."

Typical Structure

```
Controllers → Services → Repositories → Data Source (EF, SQL, etc.)
```

Example

IProductRepository.cs

```
public interface IProductRepository
{
    Task<IEnumerable<Product>> GetAllAsync();
    Task<Product?> GetByIdAsync(int id);
    Task AddAsync(Product product);
    Task UpdateAsync(Product product);
    Task DeleteAsync(int id);
}
```

ProductRepository.cs

```
public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _context;
    public ProductRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<Product>> GetAllAsync() =>
        await _context.Products.ToListAsync();

    public async Task<Product?> GetByIdAsync(int id) =>
        await _context.Products.FindAsync(id);

    public async Task AddAsync(Product product)
    {
        _context.Products.Add(product);
        await _context.SaveChangesAsync();
    }

    public async Task UpdateAsync(Product product)
    {
        _context.Products.Update(product);
        await _context.SaveChangesAsync();
    }

    public async Task DeleteAsync(int id)
    {
        var product = await _context.Products.FindAsync(id);
        if (product != null)
        {
            _context.Products.Remove(product);
            await _context.SaveChangesAsync();
        }
    }
}
```

2. Service Pattern

Definition

The **Service Pattern** (or **Service Layer Pattern**) adds another layer on top of repositories to encapsulate **business logic** or **application-specific rules**.

While repositories focus on *data access*, services focus on *business operations* and orchestrating multiple repositories if needed.

Key Benefits

- Centralizes business logic.
- Keeps controllers lightweight.
- Improves testability and maintainability.
- Allows for combining data from multiple repositories.

Example

IProductService.cs

```
public interface IProductService
{
    Task<IEnumerable<Product>> GetAllProductsAsync();
    Task<Product?> GetProductDetailsAsync(int id);
    Task CreateProductAsync(Product product);
    Task UpdateProductAsync(Product product);
    Task DeleteProductAsync(int id);
}
```

ProductService.cs

```
public class ProductService : IProductService
{
    private readonly IProductRepository _repository;

    public ProductService(IProductRepository repository)
    {
        _repository = repository;
    }

    public async Task<IEnumerable<Product>> GetAllProductsAsync()
    {
        return await _repository.GetAllAsync();
    }

    public async Task<Product?> GetProductDetailsAsync(int id)
    {
        var product = await _repository.GetByIdAsync(id);
        if (product == null)
            throw new KeyNotFoundException("Product not found");
        return product;
    }

    public async Task CreateProductAsync(Product product)
    {
        // Example: Add business rules before saving
        if (string.IsNullOrEmpty(product.Name))
            throw new ArgumentException("Product name is required");

        await _repository.AddAsync(product);
    }

    public async Task UpdateProductAsync(Product product)
    {
        await _repository.UpdateAsync(product);
    }

    public async Task DeleteProductAsync(int id)
    {
        await _repository.DeleteAsync(id);
    }
}
```

3. Controller Example (How They Work Together)

ProductsController.cs

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly IProductService _service;

    public ProductsController(IProductService service)
    {
        _service = service;
    }

    [HttpGet]
    public async Task<IActionResult> GetAll()
    {
        var products = await _service.GetAllProductsAsync();
        return Ok(products);
    }

    [HttpPost]
    public async Task<IActionResult> Create(Product product)
    {
        await _service.CreateProductAsync(product);
        return CreatedAtAction(nameof(GetAll), new { id = product.Id }, product);
    }
}
```

Summary

Layer	Responsibility	Example
Controller	Handles HTTP requests and responses	`ProductsController`
Service	Contains business logic	`ProductService`
Repository	Handles database CRUD operations	`ProductRepository`

Layer	Responsibility	Example
DbContext	Entity Framework data context	`AppDbContext`

Why Use Both Patterns?

- **Repository Pattern** → isolates data access (makes swapping databases easy).
- **Service Pattern** → isolates business logic (keeps controllers clean).
- Together, they make your app more **modular**, **testable**, and **maintainable**.