# Repository Pattern in ASP.NET Core Web API

## 📌 What is the Repository Pattern?

The Repository Pattern is a design pattern used to separate the data access logic from the business logic. It provides an abstraction layer between the data source (database) and the application.  
  
✅ Centralizes data access logic in a single place.  
✅ Reduces code duplication.  
✅ Makes code easier to maintain and test.  
✅ Supports the \*\*Dependency Injection\*\* principle.

## 📌 Why is the Repository Pattern Important?

Without the Repository Pattern, business logic and data access logic are tightly coupled, making it hard to maintain and test.  
  
The Repository Pattern helps in:  
✅ \*\*Separation of Concerns\*\*: Keeps business logic separate from data access.  
✅ \*\*Improving Testability\*\*: Enables unit testing by allowing mock implementations.  
✅ \*\*Flexibility\*\*: Easily switch between different data sources (e.g., SQL Server, MongoDB).  
✅ \*\*Code Maintainability\*\*: Reduces code duplication and keeps data access in one place.

### 📝 Example Without Repository Pattern

In this example, the `ProductController` directly interacts with `AppDbContext` for data operations.

public class ProductController : ControllerBase  
{  
 private readonly AppDbContext \_context;  
  
 public ProductController(AppDbContext context)  
 {  
 \_context = context;  
 }  
  
 [HttpGet]  
 public async Task<ActionResult<IEnumerable<Product>>> GetProducts()  
 {  
 return await \_context.Products.ToListAsync();  
 }  
  
 [HttpPost]  
 public async Task<ActionResult<Product>> AddProduct(Product product)  
 {  
 \_context.Products.Add(product);  
 await \_context.SaveChangesAsync();  
 return CreatedAtAction(nameof(GetProducts), new { id = product.Id }, product);  
 }  
}

## 📌 Drawbacks of Not Using Repository Pattern

🚫 \*\*Tightly Coupled Code\*\*: Business logic is directly tied to Entity Framework, making it hard to switch databases.  
🚫 \*\*Difficult to Test\*\*: Requires a real database for unit testing instead of using a mock repository.  
🚫 \*\*Code Duplication\*\*: Data access logic is repeated in multiple controllers.  
🚫 \*\*Less Maintainability\*\*: Changing the data source or modifying queries affects all controllers.

### 📝 Implementing the Repository Pattern

To solve these issues, we implement the Repository Pattern.

### 📝 Step 1: Create a Repository Interface

Define an interface for data operations in `Repositories/IProductRepository.cs`:

public interface IProductRepository  
{  
 Task<IEnumerable<Product>> GetProductsAsync();

Task<Product> GetProductByIdAsync(int id);

Task<Product> AddProductAsync(Product product);

Task<Product> UpdateProductAsync(int id,Product product);

Task<string> DeleteProductAsync(int id);

}

### 📝 Step 2: Implement the Repository

Create a class that implements `IProductRepository`:

using EF\_Code\_first\_Demo1.Data;

using EF\_Code\_first\_Demo1.Models;

using Microsoft.AspNetCore.Http.HttpResults;

using Microsoft.EntityFrameworkCore;

namespace EF\_Code\_first\_Demo1.Repositories

{

public class ProductRepository:IProductRepository

{

private readonly AppDbContext \_context;

public ProductRepository(AppDbContext context)

{

\_context = context;

}

public async Task<IEnumerable<Product>> GetProductsAsync()

{

return await \_context.Products.ToListAsync();

}

public async Task<Product> GetProductByIdAsync(int id)

{

return await \_context.Products.FindAsync(id);

}

public async Task<Product> AddProductAsync(Product product)

{

\_context.Products.Add(product);

await \_context.SaveChangesAsync();

return product;

}

public async Task<Product> UpdateProductAsync(int id, Product product)

{

var updateProduct = await \_context.Products.FindAsync(id);

if (updateProduct == null)

{

return null; // Return null if product not found

}

updateProduct.Name = product.Name;

updateProduct.Price = product.Price;

await \_context.SaveChangesAsync();

return updateProduct; // Return updated product

}

public async Task<string> DeleteProductAsync(int id)

{

var prodcut = await \_context.Products.FindAsync(id);

if (prodcut != null)

{

\_context.Remove(prodcut);

await \_context.SaveChangesAsync();

return "Product Removed Successfully";

}

else

return "Product not found";

}

}

}

### 📝 Step 3: Modify the Controller to Use the Repository

Instead of directly interacting with `AppDbContext`, the controller now uses `IProductRepository`.

using EF\_Code\_first\_Demo1.Data;

using EF\_Code\_first\_Demo1.Models;

using EF\_Code\_first\_Demo1.Repositories;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

namespace EF\_Code\_first\_Demo1.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

private readonly IProductRepository \_productRepository;

public ProductsController(IProductRepository repo)

{

\_productRepository = repo;

}

[HttpGet]

public async Task<ActionResult<IEnumerable<Product>>> GetAllProducts()

{

return Ok(await \_productRepository.GetProductsAsync());

}

[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProduct(int id)

{

var product= await \_productRepository.GetProductByIdAsync(id);

if(product == null)

{

return NotFound();

}

return product;

}

[HttpPost]

public async Task<ActionResult<Product>> AddProduct(Product product)

{

var newProduct = await \_productRepository.AddProductAsync(product);

return CreatedAtAction(nameof(GetProduct), new { id = newProduct.Id }, newProduct);

}

[HttpPut("{id}")]

public async Task<ActionResult> UpdateProduct(int id, [FromBody] Product updatedProduct)

{

var product = await \_productRepository.UpdateProductAsync(id, updatedProduct);

if (product == null)

{

return NotFound("Product not found"); // Return 404 Not Found

}

return NoContent(); // Return 204 No Content if successful

}

[HttpDelete("{id}")]

public async Task<ActionResult> DeleteProduct(int id)

{

string result = await \_productRepository.DeleteProductAsync(id);

if (result == "Product not found")

{

return NotFound(result); // Return 404 Not Found if product does not exist

}

return Ok(result); // Return 200 OK with success message

}

}

}

### 📝 Step 4: Register the Repository in Program.cs

Modify `Program.cs` to add dependency injection for `IProductRepository`:

var builder = WebApplication.CreateBuilder(args);  
  
builder.Services.AddDbContext<AppDbContext>(options =>  
 options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));  
  
builder.Services.AddScoped<IProductRepository, ProductRepository>(); // Register Repository  
  
builder.Services.AddControllers();  
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();  
  
var app = builder.Build();  
  
app.UseHttpsRedirection();  
app.UseAuthorization();  
app.MapControllers();  
app.Run();

## 📌 Is Repository Pattern the Same as Dependency Injection?

No, the \*\*Repository Pattern and Dependency Injection (DI) are different concepts\*\* but work together.  
  
✅ \*\*Repository Pattern\*\* is a \*\*design pattern\*\* that abstracts data access logic from the business logic.  
✅ \*\*Dependency Injection (DI)\*\* is a \*\*technique\*\* to provide dependencies (e.g., `IProductRepository`) to classes.  
  
\*\*How They Work Together:\*\*  
- The Repository Pattern creates an abstraction for data access.  
- Dependency Injection provides an instance of the repository (`IProductRepository`) to controllers.  
- This ensures the controller is not tightly coupled to any specific data access implementation.

## 📌 Summary

✅ The \*\*Repository Pattern\*\* separates data access logic from the application.  
✅ Without the Repository Pattern, code is tightly coupled to Entity Framework.  
✅ Implementing the Repository Pattern improves \*\*maintainability, flexibility, and testability\*\*.  
✅ \*\*Repository Pattern is NOT the same as Dependency Injection\*\*, but they work together.  
✅ \*\*DI helps inject the repository into controllers\*\*, making the application loosely coupled.

## 🚀 Next Steps

Would you like to:

1. Implement Unit Testing for Repository and Controller?  
2. Use Generic Repository for Reusability?  
3. Implement Caching with Repository Pattern?