在C# MVC的專案中 寫一個進銷存系統 但Controller中如果直接把邏輯寫在裡面 感覺又會不太好 但想抽出來好像又礙於對於View有時總需要一些其他資料 所以通常MVC專案中會怎麼做去處理

在C# MVC專案中，確實不建議將業務邏輯直接寫在Controller中。這會導致Controller變得過於臃腫且難以維護。通常的做法是將業務邏輯抽取到服務層(Service Layer)或使用應用程式服務(Application Services)，這樣可以保持Controller的簡潔，並使代碼更加模組化和可測試。

以下是一般的做法：

1. **服務層(Service Layer)**：
   建立一個服務層來處理業務邏輯。服務層中的類別負責具體的業務邏輯和數據處理。

2. **資料訪問層(Repository Layer)**：
   使用資料訪問層來處理數據庫操作。這層將數據庫操作從業務邏輯中分離出來，使代碼更加清晰。

3. **依賴注入(Dependency Injection)**：
   使用依賴注入將服務層和資料訪問層注入到Controller中，這樣可以提高代碼的可測試性和靈活性。

4. **ViewModel**：
   使用ViewModel來傳遞視圖需要的數據。ViewModel可以包含來自多個服務或資料源的數據，這樣可以保持視圖與業務邏輯的分離。

以下是一個示範範例：

1. 服務層定義

csharp 複製程式碼

```csharp
public interface IInventoryService
{
    Task<Product> GetProductByIdAsync(int id);
    Task<IEnumerable<Product>> GetAllProductsAsync();
    Task AddProductAsync(Product product);
    // 其他進銷存相關的方法
}

public class InventoryService : IInventoryService
{
    private readonly IProductRepository _productRepository;

    public InventoryService(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }

    public async Task<Product> GetProductByIdAsync(int id)
    {
        return await _productRepository.GetByIdAsync(id);
    }

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

    public async Task AddProductAsync(Product product)
    {
        await _productRepository.AddAsync(product);
    }

    // 其他進銷存相關的方法實現
}
```

2. 資料訪問層定義

csharp 複製程式碼

```csharp
public interface IProductRepository
{
    Task<Product> GetByIdAsync(int id);
    Task<IEnumerable<Product>> GetAllAsync();
    Task AddAsync(Product product);
    // 其他數據庫操作方法
}

public class ProductRepository : IProductRepository
{
    private readonly AppDbContext _context;

    public ProductRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task<Product> GetByIdAsync(int id)
    {
        return await _context.Products.FindAsync(id);
    }

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

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

    // 其他數據庫操作方法實現
}
```

3. 依賴注入設定 (在 `Startup.cs` 中)

csharp 複製程式碼

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AppDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnec

    services.AddScoped<IProductRepository, ProductRepository>();
    services.AddScoped<IInventoryService, InventoryService>();

    services.AddControllersWithViews();
}
```

4. Controller 使用服務層

csharp 複製程式碼

```csharp
public class ProductController : Controller
{
    private readonly IInventoryService _inventoryService;

    public ProductController(IInventoryService inventoryService)
    {
        _inventoryService = inventoryService;
    }

    public async Task<IActionResult> Index()
    {
        var products = await _inventoryService.GetAllProductsAsync();
        return View(products);
    }

    public async Task<IActionResult> Details(int id)
    {
        var product = await _inventoryService.GetProductByIdAsync(id);
        if (product == null)
        {
            return NotFound();
        }
        return View(product);
    }

    public IActionResult Create()
    {
        return View();
    }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(Product product)
    {
        if (ModelState.IsValid)
        {
            await _inventoryService.AddProductAsync(product);
            return RedirectToAction(nameof(Index));
        }
        return View(product);
    }
```

```
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(Product product)
    {
        if (ModelState.IsValid)
        {
            await _inventoryService.AddProductAsync(product);
            return RedirectToAction(nameof(Index));
        }
        return View(product);
    }
}
```

這種方式使得Controller保持簡潔，只負責處理HTTP請求和返回視圖。所有的業務邏輯都在服務層中處理，數據庫操作則在資料訪問層中處理。這樣的架構使代碼更加模組化、可維護性更高。

在資料層 如果需要Order勢必要動到產品proudct表 那該如何寫會比較好 在repository裡面 還是抽出？ 因為可能有時會需要join資料

在資料層處理需要更新多個表的操作時，有幾種常見的設計模式和策略。這些策略可以幫助你保持代碼的清晰和可維護性，同時確保數據的一致性。以下是一些具體的實現方法：

1. 將操作集中在 Repository 層

如果需要進行複雜的數據操作，包括查詢和更新多個表，你可以在 Repository 層中完成這些操作。這樣可以保持數據訪問邏輯的集中性。

OrderRepository.cs

csharp 複製程式碼

```csharp
public class OrderRepository : IOrderRepository
{
    private readonly AppDbContext _context;

    public OrderRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task<Order> GetOrderWithProductsAsync(int orderId)
    {
        return await _context.Orders
            .Include(o => o.OrderItems)
            .ThenInclude(oi => oi.Product)
            .FirstOrDefaultAsync(o => o.Id == orderId);
    }

    public async Task UpdateOrderAndProductsAsync(Order order)
    {
        // 這裡假設 OrderItems 是包含產品數據的
        foreach (var item in order.OrderItems)
        {
            _context.Products.Update(item.Product);
        }
        _context.Orders.Update(order);
        await _context.SaveChangesAsync();
    }

    // 其他訂單相關的操作
}
```

2. 使用單一責任原則和多個 Repository

將業務邏輯放在服務層，服務層調用多個 Repository 來完成操作。這樣可以將數據操作和業務邏輯分離，使代碼更加模塊化和易於維護。

OrderService.cs

csharp 複製程式碼

```csharp
public class OrderService : IOrderService
{
    private readonly IOrderRepository _orderRepository;
    private readonly IProductRepository _productRepository;
    private readonly AppDbContext _context;

    public OrderService(IOrderRepository orderRepository, IProductRepository
    {
        _orderRepository = orderRepository;
        _productRepository = productRepository;
        _context = context;
    }

    public async Task UpdateOrderAsync(Order order)
    {
        using var transaction = await _context.Database.BeginTransactionAsync
        try
        {
            foreach (var item in order.OrderItems)
            {
                var product = await _productRepository.GetByIdAsync(item.Prod
                product.Quantity -= item.Quantity;
                await _productRepository.UpdateAsync(product);
            }
            await _orderRepository.UpdateAsync(order);
            await transaction.CommitAsync();
        }
        catch
        {
            await transaction.RollbackAsync();
            throw;
        }
    }
}
```

3. 使用 Unit of Work 模式

Unit of Work 模式可以幫助你管理多個 Repository 的操作，確保在一次業務操作中涉及的多個資料庫操作要麼全部成功，要麼全部失敗。這樣可以提高數據的一致性。

UnitOfWork.cs

csharp 複製程式碼

```csharp
public class UnitOfWork : IUnitOfWork
{
    private readonly AppDbContext _context;

    public UnitOfWork(AppDbContext context)
    {
        _context = context;
    }

    public IOrderRepository Orders => new OrderRepository(_context);
    public IProductRepository Products => new ProductRepository(_context);

    public async Task<int> CompleteAsync()
    {
        return await _context.SaveChangesAsync();
    }

    public void Dispose()
    {
        _context.Dispose();
    }
}
```

使用 Unit of Work 的 OrderService.cs

csharp 複製程式碼

```csharp
public class OrderService : IOrderService
{
    private readonly IUnitOfWork _unitOfWork;

    public OrderService(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

    public async Task UpdateOrderAsync(Order order)
    {
        using var transaction = await _unitOfWork.BeginTransactionAsync();
        try
        {
            foreach (var item in order.OrderItems)
            {
                var product = await _unitOfWork.Products.GetByIdAsync(item.Pr
                product.Quantity -= item.Quantity;
                await _unitOfWork.Products.UpdateAsync(product);
            }
            await _unitOfWork.Orders.UpdateAsync(order);
            await _unitOfWork.CompleteAsync();
            await transaction.CommitAsync();
        }
        catch
        {
            await transaction.RollbackAsync();
            throw;
        }
    }
}
```

關鍵點

1. **將數據操作集中在 Repository 層**：
    - 可以在 Repository 層中實現複雜的數據操作，包括連接查詢和更新多個表。
2. **將業務邏輯放在服務層**：
    - 服務層調用多個 Repository，並處理業務邏輯，這樣可以保持代碼的模塊化和可維護性。
3. **使用 Unit of Work 模式**：
    - Unit of Work 模式幫助管理多個 Repository 的操作，確保數據一致性和事務性。

選擇適合你應用場景的方法，根據具體需求進行實現。這樣可以確保代碼的清晰和可維護性，同時保證數據的一致性和完整性。