

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

```
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

```
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 ApplicationDbContext _context;

    public ProductRepository(ApplicationDbContext 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

```
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

```
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。