ASP.NET Core Programming:

Intermediate (Integrative-Generative AI Edition)

Contents:

Author: Student Price Book Center

คำนำ

โลกของการพัฒนาเว็บแอปพลิเคชันในยุคปัจจุบันมีการเปลี่ยนแปลงอย่างรวดเร็ว การเชื่อมต่อระหว่างระบบ การจัดการข้อมูล และความปลอดภัยของผู้ใช้งานถือเป็นปัจจัยสำคัญที่นักพัฒนาต้องให้ความสำคัญ ASP.NET Core ได้กลายเป็นเฟรมเวิร์กที่ทรงพลังและยืดหยุ่นสูงสำหรับนักพัฒนาสาย .NET โดยรองรับทั้งเว็บแอปพลิเคชัน, API, การทำงานร่วมกับฐานข้อมูล และการจัดการความปลอดภัย ซึ่งทำให้สามารถสร้างระบบที่มีคุณภาพ มั่นคง และสามารถปรับขยายได้ตามความต้องการ

หนังสือเล่มนี้ **ASP.NET Core Programming: Intermediate** ออกแบบมาเพื่อผู้ที่มีพื้นฐานการพัฒนา ASP.NET Core อยู่แล้ว และต้องการต่อยอดสู่ระดับ Intermediate โดยเน้นการใช้งานจริงควบคู่กับการเข้าใจเชิงลึก เพื่อให้ผู้อ่านสามารถสร้างแอปพลิเคชันที่มีความซับซ้อนและพร้อมใช้งานในสภาพแวดล้อมจริง สารบัญครอบคลุมตั้งแต่การจัดการฐานข้อมูลด้วย **EF Core**, การสร้างและใช้งาน **Web API**, การจัดการ **Authentication และ Authorization**, การปรับแต่ง **Middleware และ Filters** ไปจนถึงการใช้ **Configuration และ Dependency Injection (DI)** อย่างมีประสิทธิภาพ

บทที่ 6: *CRUD Operations with EF Core* เป็นจุดเริ่มต้นที่สำคัญสำหรับนักพัฒนาที่ต้องการเข้าใจการจัดการข้อมูลใน ASP.NET Core ผ่าน EF Core หนังสือจะอธิบายตั้งแต่พื้นฐานของการสร้าง อ่าน แก้ไข และลบข้อมูล (CRUD) ไปจนถึงการใช้ **LINQ Query** เพื่อสืบค้นข้อมูลอย่างมีประสิทธิภาพ รวมถึงการตรวจสอบความถูกต้องของข้อมูลด้วย **Data Annotation** และการนำข้อมูลไปแสดงผลใน **View** ของ ASP.NET Core MVC และ Razor Pages ตัวอย่างและคำอธิบายเชิงลึกในบทนี้จะช่วยให้ผู้อ่านเข้าใจการจัดการข้อมูลอย่างครบวงจร

ต่อเนื่องด้วยบทที่ 7: *ASP.NET Core Web API* ซึ่งเจาะลึกการสร้างและใช้งาน RESTful API ตั้งแต่ความแตกต่างระหว่าง ASP.NET Core MVC และ Web API การทำงานกับ **JSON** ผ่าน System.Text.Json และ Newtonsoft.Json ไปจนถึงการสร้างเอกสาร API ด้วย **Swagger/OpenAPI** รวมถึงตัวอย่างบูรณาการที่จะช่วยให้ผู้อ่านสามารถออกแบบ API ที่สามารถใช้งานร่วมกับแอปพลิเคชันหลายแพลตฟอร์มได้อย่างมืออาชีพ

บทที่ 8: *Authentication & Authorization* ให้ความสำคัญกับความปลอดภัยของระบบ อธิบายการใช้งาน **ASP.NET Core Identity** สำหรับจัดการผู้ใช้ การสร้างระบบ **Login, Register และ Roles** การยืนยันตัวตนด้วย **Cookie Authentication** และ **JWT Authentication** เนื้อหานี้ช่วยให้ผู้อ่านเข้าใจกลไกการควบคุมสิทธิ์ และสามารถสร้างระบบที่ปลอดภัย รองรับผู้ใช้งานหลายระดับ และสอดคล้องกับมาตรฐานสากล

ในบทที่ 9: *Middleware and Filters* ผู้อ่านจะได้เรียนรู้การจัดการกระบวนการทำงานของแอปพลิเคชันตั้งแต่ระดับแกนกลางของ ASP.NET Core การสร้าง **Custom Middleware**, การจัดการข้อผิดพลาดด้วย **Exception Handling Middleware**, การใช้ **Action Filter และ Authorization Filter** และการทำ **Logging และ Monitoring** ตัวอย่างบูรณาการในบทนี้จะช่วยให้นักพัฒนาสามารถ

เพิ่มความยืดหยุ่น ปรับแต่งการทำงานเฉพาะของแอปพลิเคชัน และติดตามการทำงานของระบบได้อย่างเป็นระบบ

สุดท้าย บทที่ 10: *Configuration and Dependency Injection (DI)* เจาะลึกการจัดการค่า Configuration และการใช้ DI เพื่อสร้างระบบที่ยืดหยุ่นและง่ายต่อการดูแลรักษา หัวข้อสำคัญประกอบด้วย **Service Lifetime** แบบ Transient, Scoped, Singleton การใช้ **Strongly Typed Configuration**, การ **Inject Service เข้า Controller**, และการสร้าง **Custom Service** ตัวอย่างและคำอธิบายเชิงลึกในบทนี้จะช่วยให้ผู้อ่านเข้าใจโครงสร้างของระบบ และสามารถออกแบบแอปพลิเคชันที่ขยายตัวได้อย่างมั่นคง

ด้วยเนื้อหาที่ครอบคลุมทั้งพื้นฐานและเชิงลึก พร้อมตัวอย่างการประยุกต์ใช้งานจริง หนังสือเล่มนี้จึงเป็นคู่มือสำหรับนักพัฒนาที่ต้องการยกระดับทักษะการพัฒนา ASP.NET Core ให้สามารถสร้างแอปพลิเคชันที่มีคุณภาพ ปลอดภัย และพร้อมรองรับการขยายตัวในอนาคต โดยผู้อ่านสามารถใช้เป็นแนวทางทั้งในการเรียนรู้ด้วยตนเองและในการทำงานจริงในโครงการระดับมืออาชีพ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 6

CRUD Operations with EF Core

(CRUD Operations with EF Core)

เนื้อหา

- CRUD Operations with EF Core
- CRUD Operations with EF Core – รายละเอียดเชิงลึก
- เจาะลึกหัวข้อ Create, Read, Update, Delete (CRUD) ใน EF Core
- การใช้ LINQ Query กับ EF Core
- Validation ด้วย Data Annotation ใน ASP.NET Core กับ EF Core
- การแสดงผลข้อมูลใน View ใน ASP.NET Core MVC / Razor Pages

บทนำ บทที่ 6: CRUD Operations with EF Core

ในการพัฒนาแอปพลิเคชันเชิงธุรกิจ ฐานข้อมูลถือเป็นองค์ประกอบสำคัญที่ช่วยให้สามารถจัดเก็บ จัดการ และเรียกใช้งานข้อมูลได้อย่างเป็นระบบ บทที่ 6 นี้จะมุ่งเน้นไปที่การทำงานกับ **Entity Framework Core (EF Core)** ซึ่งเป็น Object-Relational Mapper (ORM) ของ .NET ที่ช่วยให้นักพัฒนาสามารถเชื่อมต่อกับฐานข้อมูลและจัดการข้อมูลได้อย่างมีประสิทธิภาพ โดยไม่จำเป็นต้องเขียน SQL ที่ซับซ้อนในทุกขั้นตอน

หัวใจหลักของการทำงานกับฐานข้อมูลคือกระบวนการ **CRUD (Create, Read, Update, Delete)** ซึ่งเป็นพื้นฐานที่ทุกระบบสารสนเทศต้องมี บทนี้จะอธิบายอย่างละเอียดถึงการสร้างข้อมูลใหม่ การอ่านและดึงข้อมูลจากฐานข้อมูล การแก้ไขข้อมูลที่มีอยู่ และการลบข้อมูลออกจากระบบ โดยทั้งหมดนี้จะถูกอธิบายผ่าน EF Core เพื่อให้นักพัฒนาสามารถเข้าใจการทำงานได้ทั้งในเชิงทฤษฎีและเชิงปฏิบัติ

นอกจาก CRUD พื้นฐานแล้ว บทนี้ยังครอบคลุมถึงการใช้ **LINQ Query (Language Integrated Query)** ซึ่งเป็นคุณสมบัติเด่นของ .NET ที่ช่วยให้การสืบค้นข้อมูลในฐานข้อมูลสามารถทำได้ในรูปแบบที่เป็นมิตรต่อการเขียนโปรแกรมมากขึ้น นักพัฒนาจะได้เรียนรู้ทั้งการใช้ LINQ แบบ Method Syntax และ Query Syntax เพื่อนำไปประยุกต์ใช้ในการสร้างเงื่อนไข การกรอง การจัดเรียง และการรวมข้อมูลตามที่ต้องการ

การจัดการข้อมูลไม่ได้มีเพียงการสร้างหรืออ่านเท่านั้น แต่ยังต้องคำนึงถึง **ความถูกต้องของข้อมูล (Data Validation)** เพื่อให้มั่นใจว่าข้อมูลที่บันทึกลงในระบบมีคุณภาพและสอดคล้องกับ

ข้อกำหนดที่กำหนดไว้ ในบทนี้จะอธิบายการใช้ **Data Annotation** เพื่อกำหนดกฎเกณฑ์ เช่น ความยาวของข้อความ รูปแบบอีเมล ค่าที่ไม่เป็นค่าว่าง รวมถึงการสร้าง Validation แบบกำหนดเอง เพื่อเสริมความยืดหยุ่นในการใช้งาน

อีกประเด็นสำคัญคือการ **แสดงผลข้อมูลใน View** โดยจะอธิบายการทำงานร่วมกันระหว่าง EF Core และ ASP.NET Core MVC เพื่อดึงข้อมูลจากฐานข้อมูลมาแสดงผลในรูปแบบที่เข้าใจง่ายและสวยงาม นักพัฒนาจะได้เรียนรู้วิธีผูกข้อมูล (Data Binding) กับ View การจัดการ Model และ ViewModel รวมถึงการนำเสนอข้อมูลในรูปแบบตาราง รายการ หรือฟอร์มที่ผู้ใช้สามารถโต้ตอบได้ เพื่อให้ผู้เรียนเข้าใจการประยุกต์ใช้งานจริง เนื้อหาในบทนี้จะมีตัวอย่างโค้ดที่ครบถ้วน ตั้งแต่การตั้งค่า DbContext การสร้าง Entity Class ไปจนถึงการเขียน Controller และ View ที่ทำงานร่วมกัน การนำเสนอจะเน้นให้ผู้อ่านสามารถต่อยอดไปสร้างระบบ CRUD ที่สมบูรณ์และสามารถปรับแต่งตามความต้องการของโครงการจริงได้

กล่าวโดยสรุป บทที่ 6: *CRUD Operations with EF Core* จะปูพื้นฐานที่สำคัญในการพัฒนาเว็บแอปพลิเคชันที่ต้องใช้ฐานข้อมูล โดยรวมทั้งการสร้าง แก้ไข ลบ และเรียกดูข้อมูล พร้อมทั้งการใช้ LINQ Query, Validation ด้วย Data Annotation และการแสดงผลใน View เมื่อเข้าใจแนวคิดและฝึกปฏิบัติแล้ว ผู้อ่านจะมีทักษะที่พร้อมต่อยอดไปยังหัวข้อที่ซับซ้อนยิ่งขึ้นในงานพัฒนาซอฟต์แวร์ระดับมืออาชีพ

CRUD Operations with EF Core

- Create, Read, Update, Delete
- การใช้ LINQ Query
- Validation ด้วย Data Annotation
- การแสดงผลข้อมูลใน View

บทที่ 6 CRUD Operations with EF Core แบบละเอียดที่สุด แยกตามหัวข้อย่อยที่คุณระบุ พร้อมตัวอย่างแนวคิดเชิงลึก

1. Create, Read, Update, Delete (CRUD)

CRUD เป็น **พื้นฐานการจัดการข้อมูลใน Database**:

Operation	คำอธิบาย	EF Core Method
Create	สร้างข้อมูลใหม่ใน Database	DbSet.Add(), DbSet.AddAsync()
Read	อ่านข้อมูลจาก Database	DbSet.ToList(), DbSet.FirstOrDefault(), DbSet.Find()
Update	แก้ไขข้อมูลที่มีอยู่	ดึง Entity ⟶ แก้ไข Property ⟶ SaveChanges()
Delete	ลบข้อมูล	DbSet.Remove() หรือ DbSet.RemoveRange() ⟶ SaveChanges()

เชิงลึก:

- EF Core ใช้ **Change Tracker** เพื่อติดตามการเปลี่ยนแปลงของ Entity
- การบันทึกจะเกิดขึ้น **จริง** ก็ต่อเมื่อเรียก SaveChanges() หรือ SaveChangesAsync()

ตัวอย่าง

```
// Create
var product = new Product { Name = "Laptop", Price = 50000 };
_context.Products.Add(product);
_context.SaveChanges();

// Read
var products = _context.Products.ToList();
var single = _context.Products.FirstOrDefault(p => p.Id == 1);

// Update
var p = _context.Products.Find(1);
p.Price = 52000;
_context.SaveChanges();

// Delete
var del = _context.Products.Find(1);
_context.Products.Remove(del);
_context.SaveChanges();
```

---

2. การใช้ LINQ Query

LINQ (Language Integrated Query) ช่วย **Query ข้อมูลในรูปแบบ strongly-typed**

ตัวอย่าง LINQ Queries

```
// Filter
var expensiveProducts = _context.Products
    .Where(p => p.Price > 1000)
    .ToList();

// Order
var sortedProducts = _context.Products
    .OrderByDescending(p => p.Price)
```

```
    .ToList();

// Projection (select specific columns)
var productNames = _context.Products
    .Select(p => p.Name)
    .ToList();

// Join with Category
var query = from p in _context.Products
            join c in _context.Categories
            on p.CategoryId equals c.Id
            select new { p.Name, c.Name, p.Price };
```

เชิงลึก:

- EF Core จะ **แปลง LINQ เป็น SQL Query** อัตโนมัติ
- สนับสนุน **Eager Loading (Include)**, **Lazy Loading**, **Explicit Loading**

---

3. Validation ด้วย Data Annotation

Data Annotation ช่วยตรวจสอบข้อมูลบน **Model Level** ก่อนบันทึก

ตัวอย่าง

```
using System.ComponentModel.DataAnnotations;

public class Product
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Name is required")]
    [StringLength(100, MinimumLength = 3)]
    public string Name { get; set; }

    [Range(0, 100000, ErrorMessage = "Price must be between 0-100,000")]
    public decimal Price { get; set; }

    [Range(0, int.MaxValue)]
```

```
    public int Stock { get; set; }
}
```

ใน Controller

```
[HttpPost]
public IActionResult Create(Product product)
{
    if(!ModelState.IsValid)
    {
        return View(product); // แสดง Validation Errors ใน View
    }
    _context.Products.Add(product);
    _context.SaveChanges();
    return RedirectToAction("Index");
}
```

เชิงลึก:

- Validation ทำงานทั้ง **Client-Side (via Tag Helpers)** และ **Server-Side**
- สามารถเพิ่ม **Custom Validation Attribute** สำหรับกฎเฉพาะได้

---

4. การแสดงผลข้อมูลใน View

การส่งข้อมูลจาก Controller ⟶ View ใช้ **Model / ViewData / ViewBag / TempData**

ตัวอย่าง Razor View

```
@model List<Product>

<h2>Products List</h2>
<table border="1">
<tr>
    <th>Name</th>
    <th>Price</th>
    <th>Stock</th>
    <th>Actions</th>
</tr>
@foreach(var p in Model)
{
    <tr>
```

```
        <td>@p.Name</td>
        <td>@p.Price:C</td>
        <td>@p.Stock</td>
        <td>
            <a href="/Products/Edit/@p.Id">Edit</a> |
            <a href="/Products/Delete/@p.Id">Delete</a>
        </td>
    </tr>
}
</table>
<a href="/Products/Create">Add Product</a>
```

เชิงลึก:

- ใช้ **Tag Helpers** เช่น <form asp-action="Create"> เพื่อเชื่อม Model Binding
- การ Edit / Delete สามารถใช้ **ActionLink** หรือ **form + POST**

สรุปเชิงลึกของบท 6

1. **CRUD Operations** ⟶ EF Core ใช้ DbSet + Change Tracker + SaveChanges
2. **LINQ Query** ⟶ Query แบบ strongly-typed, แปลงเป็น SQL อัตโนมัติ
3. **Validation** ⟶ Data Annotation ตรวจสอบข้อมูลทั้ง Server & Client
4. **View Rendering** ⟶ Razor + Model Binding + Tag Helpers แสดงผลข้อมูล

ขยายความ **เชิงลึก** ของบทที่ 6 **CRUD Operations with EF Core** ให้ครบทุกมิติ ทั้ง **การทำงานภายใน EF Core, LINQ, Validation, และการแสดงผลใน View** พร้อมเชื่อมโยงกับ **ASP.NET Core MVC Architecture**

CRUD Operations with EF Core – รายละเอียดเชิงลึก

---

1. Create, Read, Update, Delete (CRUD)

หลักการทำงานภายใน EF Core

- EF Core ใช้ **DbContext** เป็นตัวกลางในการสื่อสารกับ Database
- **DbSet** แทนตาราง (Table) ใน Database
- ทุก Entity ที่ถูกดึงมาจาก DbContext จะอยู่ใน **Change Tracker**
    - EF Core ติดตามการเปลี่ยนแปลงของ Property

- เมื่อเรียก SaveChanges() ⟶ EF Core จะสร้าง SQL **INSERT / UPDATE / DELETE** ตามความเปลี่ยนแปลง

- สามารถทำงาน **Async** เพื่อเพิ่มประสิทธิภาพด้วย AddAsync(), SaveChangesAsync(), ToListAsync()

ตัวอย่าง

```
// Create
var product = new Product { Name="Laptop", Price=50000 };
_context.Products.Add(product);   // อยู่ใน Change Tracker
_context.SaveChanges();           // EF Core Generate INSERT

// Read
var products = _context.Products.ToList();       // SELECT * FROM Products
var single = _context.Products.FirstOrDefault(p => p.Id == 1);

// Update
var p = _context.Products.Find(1);
p.Price = 52000;                  // Change Tracker Detects Modification
_context.SaveChanges();           // EF Core Generate UPDATE

// Delete
var del = _context.Products.Find(1);
_context.Products.Remove(del);     // Change Tracker Marks as Deleted
_context.SaveChanges();           // EF Core Generate DELETE
```

เชิงลึกเพิ่มเติม:

- การทำ **Batch Operations** เช่น RemoveRange() ลดจำนวน SQL Round-trip
- การใช้ **AsNoTracking()** สำหรับ Read-only Query ⟶ ปรับปรุง Performance
- EF Core รองรับ **Soft Delete** โดยใช้ Property เช่น IsDeleted แทนลบจริง

---

2. การใช้ LINQ Query

ทำไมต้อง LINQ กับ EF Core

- LINQ เป็น **Query Language แบบ Strongly Typed**
- EF Core จะแปลง LINQ เป็น SQL อัตโนมัติ
- สนับสนุน:
  - **Filtering:** Where()

- **Sorting:** OrderBy(), OrderByDescending()
- **Projection:** Select()
- **Joining / Eager Loading:** Include()
- **Aggregation:** Count(), Sum(), Average()

ตัวอย่างเชิงลึก

```
// Eager Loading
var productsWithCategory = _context.Products
                                .Include(p => p.Category)
                                .Where(p => p.Price > 1000)
                                .OrderByDescending(p => p.Price)
                                .ToList();


// Projection
var productNames = _context.Products
                            .Select(p => new { p.Name, p.Price })
                            .ToList();


// Join แบบ LINQ Query Syntax
var query = from p in _context.Products
            join c in _context.Categories
            on p.CategoryId equals c.Id
            select new { p.Name, c.Name, p.Price };
```

เชิงลึกเพิ่มเติม:

- **Lazy Loading** ⟶ EF Core จะโหลด Navigation Property อัตโนมัติเมื่อเข้าถึง
- **Explicit Loading** ⟶ ใช้ Entry(entity).Collection(...).Load() เพื่อโหลดเฉพาะเมื่อจำเป็น
- LINQ Query ถูกแปลงเป็น **Parameterized SQL** ⟶ ลด SQL Injection Risk

---

3. Validation ด้วย Data Annotation

การตรวจสอบความถูกต้องของข้อมูล

- **Server-Side Validation**
  - ใช้ **Data Annotation Attributes** เช่น [Required], [Range], [EmailAddress], [StringLength]
  - ทำงานใน Controller ผ่าน ModelState.IsValid

- **Client-Side Validation**
    - Razor Page + Tag Helpers ⟶ แปลงเป็น HTML5 Validation + jQuery Validation
    - ลดการส่ง Request ผิดพลาดไปยัง Server

ตัวอย่าง

```
public class Product
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Name is required")]
    [StringLength(100, MinimumLength = 3)]
    public string Name { get; set; }

    [Range(0, 100000)]
    public decimal Price { get; set; }

    [Range(0, int.MaxValue)]
    public int Stock { get; set; }
}
```

Controller

```
[HttpPost]
public IActionResult Create(Product product)
{
    if(!ModelState.IsValid)
    {
        return View(product); // แสดง Validation Errors
    }
    _context.Products.Add(product);
    _context.SaveChanges();
    return RedirectToAction("Index");
}
```

เชิงลึกเพิ่มเติม:

- สามารถสร้าง **Custom Validation Attribute** เช่น ตรวจสอบ Stock > 0 เฉพาะบางเงื่อนไข
- ใช้ **IValidatableObject** สำหรับ Validation ระดับ Model

4. การแสดงผลข้อมูลใน View

การส่งข้อมูลจาก Controller ⟶ Razor View

- ใช้ **Model** หรือ **ViewData / ViewBag / TempData**
- Razor View ใช้ **@model** เพื่อรับ Data Strongly Typed
- ใช้ **foreach / table** แสดงข้อมูล

ตัวอย่าง

```
@model List<Product>
<h2>Products List</h2>

@if(TempData["Message"] != null)
{
    <div style="color:green">@TempData["Message"]</div>
}

<table border="1">
<tr><th>Name</th><th>Price</th><th>Stock</th></tr>
@foreach(var p in Model)
{
    <tr>
        <td>@p.Name</td>
        <td>@p.Price:C</td>
        <td>@p.Stock</td>
    </tr>
}
</table>
<a href="/Products/Create">Add Product</a>
```

เชิงลึก:

- ใช้ **Tag Helpers** เพื่อ Bind Form ⟶ Model
- <form asp-action="Create">
- <input asp-for="Name" />
-
- </form>

- Razor + EF Core ช่วยให้ **Data Binding + Validation + CRUD** ทำงานรวมกันได้อย่างราบรื่น
- TempData สามารถใช้ **Flash Message** แจ้งผลการบันทึก

สรุปเชิงลึก

หัวข้อ	ประเด็นสำคัญเชิงลึก
CRUD Operations	ใช้ **Change Tracker + DbSet + SaveChanges()**, รองรับ Async, Batch, Soft Delete
LINQ Query	Strongly Typed Query, SQL Injection Safe, รองรับ Eager / Lazy / Explicit Loading
Validation	Data Annotation + ModelState + Client-side Validation + Custom Validation
View Rendering	Razor + Model Binding + Tag Helpers + TempData Flash Message

Insight:

- EF Core ทำงานเป็น **ORM Full Stack** ตั้งแต่ **Model ⟶ DbContext ⟶ Database ⟶ Razor View**
- การใช้ LINQ + Validation + Razor View ร่วมกัน ทำให้ CRUD Operations **ปลอดภัย, Type-safe, และ Maintainable**

เจาะลึกหัวข้อ Create, Read, Update, Delete (CRUD) ใน EF Core

เจาะลึกหัวข้อ **Create, Read, Update, Delete (CRUD) ใน EF Core** โดยละเอียดทั้งเชิงแนวคิดและการทำงานภายใน

---

1. Create (สร้างข้อมูลใหม่)

หลักการ

- การสร้าง Entity ใหม่ใน EF Core ทำผ่าน **DbSet.Add()** หรือ **DbSet.AddAsync()**
- EF Core จะ **ติดตาม Entity** ใน **Change Tracker**
- การบันทึกจริงเกิดขึ้นเมื่อเรียก **SaveChanges()** หรือ **SaveChangesAsync()**

ตัวอย่าง

```
var product = new Product { Name = "Laptop", Price = 50000, Stock = 10 };

// Add Entity เข้า Change Tracker
_context.Products.Add(product);
```

```
// บันทึกลง Database
_context.SaveChanges();
```

เชิงลึก

- AddAsync() เหมาะกับการทำงานแบบ Async
- Change Tracker ช่วยให้ EF Core รู้ว่า Entity ใดเป็น **Added, Modified, Deleted**

---

2. Read (อ่านข้อมูล)

หลักการ

- ใช้ **LINQ Query** หรือ **DbSet.Find()** / **DbSet.FirstOrDefault()**
- สามารถทำ **Filtering, Sorting, Projection, Joining**
- EF Core จะแปลงเป็น **SQL SELECT** อัตโนมัติ

ตัวอย่าง

```
// ดึงข้อมูลทั้งหมด
var products = _context.Products.ToList();

// ดึงข้อมูลเฉพาะรายการแรก
var single = _context.Products.FirstOrDefault(p => p.Id == 1);

// ดึงข้อมูลตามเงื่อนไข
var expensiveProducts = _context.Products
                        .Where(p => p.Price > 1000)
                        .ToList();
```

เชิงลึก

- AsNoTracking() ⟶ ใช้สำหรับ Read-only Query ⟶ ลด Memory & Improve Performance
- สนับสนุน **Eager Loading** ด้วย Include() เพื่อโหลด Navigation Properties พร้อมกัน

---

3. Update (แก้ไขข้อมูล)

หลักการ

1. ดึง Entity จาก DbContext
2. แก้ไข Property ของ Entity
3. SaveChanges() ⟶ EF Core Generate UPDATE SQL

ตัวอย่าง

```
var product = _context.Products.Find(1); // ดึง Entity
```

```
product.Price = 52000;                  // แก้ไข Property
_context.SaveChanges();                 // EF Core ส่ง SQL UPDATE
```

เชิงลึก

- EF Core ใช้ **Change Tracker** ตรวจสอบว่า Property ไหนถูกแก้ไข
- สามารถทำ **Batch Update** ผ่าน ExecuteUpdate() ใน EF Core 7+

---

4. Delete (ลบข้อมูล)

หลักการ

- ใช้ **DbSet.Remove()** หรือ **DbSet.RemoveRange()**
- EF Core ติดตาม Entity และ Generate SQL DELETE เมื่อ SaveChanges()

ตัวอย่าง

```
var product = _context.Products.Find(1);
_context.Products.Remove(product);
_context.SaveChanges();
```

เชิงลึก

- **Soft Delete**: แทนที่จะลบจริง ใช้ Property เช่น IsDeleted และ Filter Query
- RemoveRange() ⟶ ลบหลาย Entity พร้อมกัน ลดจำนวน SQL Round-trip

---

Flow การทำงานภายใน EF Core

1. Controller รับ Request (Create/Read/Update/Delete)
2. ดึงหรือสร้าง Entity ผ่าน DbContext
3. EF Core **Change Tracker** ติดตามการเปลี่ยนแปลง
4. SaveChanges() ⟶ EF Core แปลงเป็น SQL (INSERT/SELECT/UPDATE/DELETE)
5. Database ทำงาน
6. Controller ส่งข้อมูลไปยัง Razor View ⟶ Response

สรุปเชิงลึกของ CRUD

Operation	Method (EF Core)	Change Tracker	SQL Generated
Create	Add / AddAsync	Added	INSERT
Read	Find / FirstOrDefault / ToList / LINQ	N/A	SELECT
Update	Modify Property ⟶ SaveChanges	Modified	UPDATE
Delete	Remove / RemoveRange	Deleted	DELETE

- CRUD คือ **หัวใจของ Data Management**
- EF Core ช่วยให้ **Type-safe, LINQ-integrated, Change Tracking** ทำงานครบวงจร

ตัวอย่าง **CRUD Operations with EF Core** ให้ครบทั้ง **พื้นฐาน 3 โปรแกรม** และ **แนวประยุกต์ 3 โปรแกรม** พร้อม **Project Structure, โค้ดเต็มไฟล์, คำอธิบาย, และผลการรัน**

ตัวอย่างพื้นฐาน CRUD (3 โปรแกรม)

ตัวอย่าง 1 – Basic Product CRUD (SQL Server)

Project Structure

```
/BasicProductCRUD
├── Controllers
│   └── ProductsController.cs
├── Models
│   └── Product.cs
├── Data
│   └── AppDbContext.cs
├── Views
│   └── Products
│       ├── Index.cshtml
│       ├── Create.cshtml
│       ├── Edit.cshtml
│       └── Delete.cshtml
├── Program.cs
└── appsettings.json
```

Model: Product.cs

```
using System.ComponentModel.DataAnnotations;

namespace BasicProductCRUD.Models
{
    public class Product
    {
        [Key]
```

```
        public int Id { get; set; }

        [Required]
        [StringLength(100)]
        public string Name { get; set; }

        [Range(0, 100000)]
        public decimal Price { get; set; }

        [Range(0, int.MaxValue)]
        public int Stock { get; set; }
    }
}
```

DbContext: AppDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using BasicProductCRUD.Models;

namespace BasicProductCRUD.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
        public DbSet<Product> Products { get; set; }
    }
}
```

Controller: ProductsController.cs

```
using Microsoft.AspNetCore.Mvc;
using BasicProductCRUD.Data;
using BasicProductCRUD.Models;

namespace BasicProductCRUD.Controllers
{
    public class ProductsController : Controller
    {
```

```
private readonly AppDbContext _context;
public ProductsController(AppDbContext context) => _context = context;

public IActionResult Index() => View(_context.Products.ToList());

public IActionResult Create() => View();

[HttpPost]
public IActionResult Create(Product product)
{
    if (!ModelState.IsValid) return View(product);
    _context.Products.Add(product);
    _context.SaveChanges();
    return RedirectToAction("Index");
}

public IActionResult Edit(int id)
{
    var product = _context.Products.Find(id);
    if (product == null) return NotFound();
    return View(product);
}

[HttpPost]
public IActionResult Edit(Product product)
{
    if (!ModelState.IsValid) return View(product);
    _context.Products.Update(product);
    _context.SaveChanges();
    return RedirectToAction("Index");
}

public IActionResult Delete(int id)
{
```

```
            var product = _context.Products.Find(id);
            if (product == null) return NotFound();
            return View(product);
        }

        [HttpPost, ActionName("Delete")]
        public IActionResult DeleteConfirmed(int id)
        {
            var product = _context.Products.Find(id);
            _context.Products.Remove(product);
            _context.SaveChanges();
            return RedirectToAction("Index");
        }
    }
}
```

Program.cs

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();
app.UseStaticFiles();
app.UseRouting();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Products}/{action=Index}/{id?}");
app.Run();
```

appsettings.json

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=localhost;Database=BasicProductDb;Trusted_Connection=True;"
```

```
  }
}
```

ผลการรัน

- /Products/Index ⟶ แสดงรายการ Product
- /Products/Create ⟶ เพิ่ม Product ใหม่
- /Products/Edit/{id} ⟶ แก้ไข Product
- /Products/Delete/{id} ⟶ ลบ Product

ตัวอย่าง 2 – SQLite CRUD

- เปลี่ยน Database เป็น SQLite
- Project Structure เหมือนตัวอย่าง 1
- Program.cs:

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection")));
```

- appsettings.json:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=Products.db"
  }
}
```

ผลการรัน

- EF Core สร้างไฟล์ Products.db
- CRUD เหมือน SQL Server
- /Products/Index ⟶ แสดง Product List
- Migration สร้างตารางและ Column อัตโนมัติ

ตัวอย่าง 3 – Product CRUD + TempData Notification

- เพิ่ม TempData แจ้งเตือนเมื่อ Create/Edit/Delete

Controller Update:

```
[HttpPost]
public IActionResult Create(Product product)
{
    if(!ModelState.IsValid) return View(product);
    _context.Products.Add(product);
```

```
    _context.SaveChanges();
    TempData["Message"] = "Product created successfully!";
    return RedirectToAction("Index");
}
```

Razor View Index.cshtml

```
@model List<Product>
@if(TempData["Message"] != null)
{
    <div style="color:green">@TempData["Message"]</div>
}
<table>...</table>
```

ผลการรัน

- เมื่อสร้าง/แก้ไข/ลบ Product ⟶ แสดงข้อความแจ้งเตือน
- CRUD + Razor + TempData ทำงานรวมกันอย่างราบรื่น

ตัวอย่างแนวประยุกต์ CRUD (3 โปรแกรม)

ตัวอย่าง 1 – E-Commerce Dashboard CRUD

- **Model:** Product + Category + Price + Stock
- **DbContext:** Track Products + Categories
- **Controller:** CRUD + LINQ Query
- **View:** Razor + Layout + TempData Notification

ผลการรัน

- Dashboard แสดง Product + Category + Stock
- TempData แจ้งเตือนหลังสร้าง/แก้ไข Product
- LINQ ใช้ Filter / Sort / Include Category

ตัวอย่าง 2 – User Management System

- **Model:** User (Username, Email, Password)
- **Controller:** Create + List + Edit
- **View:** Razor Form + TempData Notification
- **DbContext:** Track Users

ผลการรัน

- /Users/Index ⟶ แสดง User List
- /Users/Create ⟶ เพิ่ม User ⟶ TempData Message ⟶ Redirect

ตัวอย่าง 3 – Multi-Tenant Product App

- **Model:** Product + Category
- **DbContext:** Dynamic Connection String
- **Program.cs:** เลือก Database ตาม Tenant Query String

```
var dbType = HttpContext.Request.Query["db"];
options.UseSqlServer(dbType=="A"?sqlA:sqlB);
```

ผลการรัน

- /Products?db=A ⟶ ใช้ SQL Server Tenant A
- /Products?db=B ⟶ ใช้ SQL Server Tenant B
- รองรับ Multi-Tenant แบบ Dynamic

ตัวอย่าง **แนวประยุกต์ 3 ตัว** สำหรับ **CRUD Operations with EF Core** ให้ละเอียดครบทุกส่วน รวม **Project Structure, Models, DbContext, Controller, Razor Views, Migration, Program.cs** และ **ผลการรัน**

ตัวอย่างแนวประยุกต์ 1 – E-Commerce Dashboard CRUD

Project Structure

```
/ECommerceDashboard
│ -- Controllers
│    └-- ProductsController.cs
│ -- Models
│    ├-- Product.cs
│    └-- Category.cs
│ -- Data
│    └-- AppDbContext.cs
│ -- Views
│    ├-- Shared
│    │    └-- _Layout.cshtml
│    └-- Products
│         ├-- Index.cshtml
```

```
│       ├── Create.cshtml
│       ├── Edit.cshtml
│       └── Delete.cshtml
│   ── Program.cs
│   ── appsettings.json
```

Models

Category.cs

```
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Product> Products { get; set; }
}
```

Product.cs

```
using System.ComponentModel.DataAnnotations;

public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }

    [Range(0, 100000)]
    public decimal Price { get; set; }

    [Range(0, int.MaxValue)]
    public int Stock { get; set; }

    [Required]
    public int CategoryId { get; set; }
    public Category Category { get; set; }
}
```

DbContext: AppDbContext.cs

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) {}
    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }
}
```

Controller: ProductsController.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

public class ProductsController : Controller
{
    private readonly AppDbContext _context;
    public ProductsController(AppDbContext context) => _context = context;

    public IActionResult Index()
    {
        var products = _context.Products.Include(p => p.Category).ToList();
        return View(products);
    }

    public IActionResult Create()
    {
        ViewBag.Categories = _context.Categories.ToList();
        return View();
    }

    [HttpPost]
    public IActionResult Create(Product product)
    {
        if(!ModelState.IsValid)
```

```
        {
            ViewBag.Categories = _context.Categories.ToList();
            return View(product);
        }
        _context.Products.Add(product);
        _context.SaveChanges();
        TempData["Message"] = "Product created successfully!";
        return RedirectToAction("Index");
    }

    public IActionResult Edit(int id)
    {
        var product = _context.Products.Find(id);
        if(product == null) return NotFound();
        ViewBag.Categories = _context.Categories.ToList();
        return View(product);
    }

    [HttpPost]
    public IActionResult Edit(Product product)
    {
        if(!ModelState.IsValid)
        {
            ViewBag.Categories = _context.Categories.ToList();
            return View(product);
        }
        _context.Products.Update(product);
        _context.SaveChanges();
        TempData["Message"] = "Product updated successfully!";
        return RedirectToAction("Index");
    }

    public IActionResult Delete(int id)
    {
```

```
        var product = _context.Products.Find(id);
        if(product == null) return NotFound();
        return View(product);
    }

    [HttpPost, ActionName("Delete")]
    public IActionResult DeleteConfirmed(int id)
    {
        var product = _context.Products.Find(id);
        _context.Products.Remove(product);
        _context.SaveChanges();
        TempData["Message"] = "Product deleted successfully!";
        return RedirectToAction("Index");
    }
}
```

Program.cs

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();
app.UseStaticFiles();
app.UseRouting();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Products}/{action=Index}/{id?}");
app.Run();
```

ผลการรัน

- /Products/Index ⟶ แสดง Product + Category + Stock
- TempData แจ้งเตือนเมื่อ Create/Edit/Delete
- CRUD + LINQ + Razor View + Layout ทำงานครบวงจร

ตัวอย่างแนวประยุกต์ 2 – User Management System

Project Structure

```
/UserManagement
├── Controllers
│   └── UsersController.cs
├── Models
│   └── User.cs
├── Data
│   └── AppDbContext.cs
├── Views
│   └── Users
│       ├── Index.cshtml
│       ├── Create.cshtml
│       └── Edit.cshtml
├── Program.cs
└── appsettings.json
```

Model: User.cs

```
using System.ComponentModel.DataAnnotations;

public class User
{
    public int Id { get; set; }

    [Required, StringLength(50)]
    public string Username { get; set; }

    [Required, EmailAddress]
    public string Email { get; set; }

    [Required, StringLength(100)]
    public string Password { get; set; }
}
```

DbContext: AppDbContext.cs

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) {}
    public DbSet<User> Users { get; set; }
}
```

Controller: UsersController.cs

```
using Microsoft.AspNetCore.Mvc;

public class UsersController : Controller
{
    private readonly AppDbContext _context;
    public UsersController(AppDbContext context) => _context = context;

    public IActionResult Index() => View(_context.Users.ToList());

    public IActionResult Create() => View();

    [HttpPost]
    public IActionResult Create(User user)
    {
        if(!ModelState.IsValid) return View(user);
        _context.Users.Add(user);
        _context.SaveChanges();
        TempData["Message"] = "User created successfully!";
        return RedirectToAction("Index");
    }
}
```

ผลการรัน

- /Users/Index ⟶ แสดง User List
- /Users/Create ⟶ เพิ่ม User พร้อม TempData แจ้งเตือน
- CRUD แบบง่ายสำหรับ User Management

ตัวอย่างแนวประยุกต์ 3 – Multi-Tenant Product App

Project Structure

```
/MultiTenantProduct
├── Controllers
│   └── ProductsController.cs
├── Models
│   └── Product.cs
├── Data
│   └── AppDbContext.cs
├── Views
│   └── Products
│       ├── Index.cshtml
│       └── Create.cshtml
├── Program.cs
└── appsettings.json
```

Dynamic DbContext

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddDbContext<AppDbContext>((serviceProvider, options) =>
{
    var httpContext = serviceProvider.GetRequiredService<IHttpContextAccessor>().HttpContext;
    var dbType = httpContext.Request.Query["db"].ToString();
    var sqlA = builder.Configuration.GetConnectionString("TenantA");
    var sqlB = builder.Configuration.GetConnectionString("TenantB");
    options.UseSqlServer(dbType == "A" ? sqlA : sqlB);
});

builder.Services.AddHttpContextAccessor();

var app = builder.Build();
app.UseRouting();
```

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Products}/{action=Index}/{id?}");
app.Run();
```

Controller: ProductsController.cs

```
using Microsoft.AspNetCore.Mvc;

public class ProductsController : Controller
{
    private readonly AppDbContext _context;
    public ProductsController(AppDbContext context) => _context = context;

    public IActionResult Index() => View(_context.Products.ToList());
}
```

ผลการรัน

- /Products?db=A ⟶ ใช้ SQL Server Tenant A
- /Products?db=B ⟶ ใช้ SQL Server Tenant B
- รองรับ Multi-Tenant แบบ Dynamic

การใช้ LINQ Query กับ EF Core

---

1. LINQ (Language Integrated Query) คืออะไร

- LINQ เป็นเครื่องมือของ .NET สำหรับ **Query ข้อมูลจาก Collections, Database, XML, Objects** โดยใช้ **C# syntax**
- EF Core รองรับ LINQ สำหรับ **Query Database** ซึ่ง EF Core จะ **แปลงเป็น SQL** อัตโนมัติ
- ทำให้โค้ด **Type-safe, IntelliSense-friendly, readable** และสามารถใช้ **Lambda Expressions** ร่วมได้

---

2. รูปแบบ LINQ กับ EF Core

2.1 Query Syntax

```
var products = from p in _context.Products
               where p.Price > 1000
               orderby p.Name
```

```
        select p;
```

2.2 Method Syntax (Lambda Expressions)

```
var products = _context.Products
                    .Where(p => p.Price > 1000)
                    .OrderBy(p => p.Name)
                    .ToList();
```

ข้อสังเกต:

- Method syntax เป็นที่นิยมมากกว่าใน EF Core
- สามารถใช้ **Select, Where, OrderBy, Join, Include** ได้ครบ

---

3. ตัวอย่าง LINQ Query กับ EF Core

3.1 Filter ข้อมูล

```
var expensiveProducts = _context.Products
                            .Where(p => p.Price > 5000)
                            .ToList();
```

3.2 Sorting

```
var sortedProducts = _context.Products
                        .OrderBy(p => p.Name)  // Ascending
                        .ToList();
var sortedDesc = _context.Products
                    .OrderByDescending(p => p.Price)
                    .ToList();
```

3.3 Projection (เลือกเฉพาะ Column)

```
var productNames = _context.Products
                        .Select(p => new { p.Name, p.Price })
                        .ToList();
```

3.4 Join / Include (Eager Loading)

```
// Join Product กับ Category
var productsWithCategory = _context.Products
                            .Include(p => p.Category)
                            .ToList();
```

3.5 Grouping

```
var categoryGroup = _context.Products
                        .GroupBy(p => p.CategoryId)
```

```
                .Select(g => new { CategoryId = g.Key, Count = g.Count() })
                .ToList();
```

3.6 Aggregation

```
var totalStock = _context.Products.Sum(p => p.Stock);
var avgPrice = _context.Products.Average(p => p.Price);
```

3.7 Paging

```
var page1 = _context.Products
                .OrderBy(p => p.Name)
                .Skip(0)
                .Take(10)
                .ToList();
var page2 = _context.Products
                .OrderBy(p => p.Name)
                .Skip(10)
                .Take(10)
                .ToList();
```

---

4. LINQ + CRUD

- **Create:** LINQ ไม่ใช้สำหรับ Create แต่สามารถใช้ LINQ ตรวจสอบเงื่อนไขก่อน Add
- **Read:** LINQ ใช้ดึงข้อมูลจาก DbSet เช่น Where, Include, Select
- **Update:** LINQ ใช้ค้นหา Entity ก่อน Update
- **Delete:** LINQ ใช้ค้นหา Entity ก่อน Remove

ตัวอย่าง Update + LINQ

```
var product = _context.Products.FirstOrDefault(p => p.Name == "Laptop");
if(product != null)
{
    product.Price = 52000;
    _context.SaveChanges();
}
```

---

5. ข้อดีของ LINQ กับ EF Core

1. **Type-safe:** ตรวจสอบ Data Type ขณะ Compile
2. **Readable:** โค้ด Query เข้าใจง่าย
3. **SQL Generation:** EF Core แปลง LINQ เป็น SQL อัตโนมัติ

4. **Supports Relationships:** ใช้ Include, Join, GroupBy ได้
5. **Composable Queries:** สามารถต่อ Chain Methods ได้

สร้างตัวอย่าง **การใช้ LINQ Query กับ EF Core** ให้ครบทั้ง **พื้นฐาน 3 โปรแกรม** และ **แนวประยุกต์ 3 โปรแกรม** พร้อม **Project Structure, โค้ดเต็มไฟล์, คำอธิบายโค้ด และผลการรัน**

ตัวอย่างพื้นฐาน LINQ Query (3 โปรแกรม)

ตัวอย่าง 1 – Filter และ Sort Products

Project Structure

```
/LINQBasic1
├── Controllers
│   └── ProductsController.cs
├── Models
│   └── Product.cs
├── Data
│   └── AppDbContext.cs
├── Views
│   └── Products
│       └── Index.cshtml
├── Program.cs
└── appsettings.json
```

Model: Product.cs

```
using System.ComponentModel.DataAnnotations;

namespace LINQBasic1.Models
{
    public class Product
    {
        public int Id { get; set; }

        [Required]
        public string Name { get; set; }
```

```
        public decimal Price { get; set; }

        public int Stock { get; set; }
    }
}
```

DbContext: AppDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using LINQBasic1.Models;

namespace LINQBasic1.Data
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
        public DbSet<Product> Products { get; set; }
    }
}
```

Controller: ProductsController.cs

```
using Microsoft.AspNetCore.Mvc;
using LINQBasic1.Data;
using System.Linq;

namespace LINQBasic1.Controllers
{
    public class ProductsController : Controller
    {
        private readonly AppDbContext _context;
        public ProductsController(AppDbContext context) => _context = context;

        public IActionResult Index()
        {
            // LINQ Query: Filter Price > 1000 and Sort by Name
            var products = _context.Products
```

```
                .Where(p => p.Price > 1000)
                .OrderBy(p => p.Name)
                .ToList();
            return View(products);
        }
    }
}
```

Razor View: Index.cshtml

```
@model List<LINQBasic1.Models.Product>

<h2>Products (Price > 1000, Sorted by Name)</h2>
<table>
    <tr><th>Name</th><th>Price</th><th>Stock</th></tr>
@foreach(var p in Model)
{
    <tr>
        <td>@p.Name</td>
        <td>@p.Price</td>
        <td>@p.Stock</td>
    </tr>
}
</table>
```

ผลการรัน

- แสดงเฉพาะ Product ที่ราคา > 1000
- เรียงตามชื่อ

ตัวอย่าง 2 – Projection และ Aggregation

- **Query:** ดึงเฉพาะ Name และ Price, หาค่าเฉลี่ยราคา และจำนวนสินค้า

Controller

```
public IActionResult Summary()
{
    var productInfo = _context.Products
                .Select(p => new { p.Name, p.Price })
                .ToList();
```

```
    var avgPrice = _context.Products.Average(p => p.Price);
    var totalStock = _context.Products.Sum(p => p.Stock);

    ViewBag.AvgPrice = avgPrice;
    ViewBag.TotalStock = totalStock;

    return View(productInfo);
}
```

Razor View: Summary.cshtml

```
@model List<dynamic>
<h2>Product Summary</h2>
<p>Average Price: @ViewBag.AvgPrice</p>
<p>Total Stock: @ViewBag.TotalStock</p>

<table>
    <tr><th>Name</th><th>Price</th></tr>
@foreach(var p in Model)
{
    <tr>
        <td>@p.Name</td>
        <td>@p.Price</td>
    </tr>
}
</table>
```

ผลการรัน

- แสดงเฉพาะชื่อและราคา
- แสดงค่าเฉลี่ยราคาและจำนวน Stock รวม

ตัวอย่าง 3 – Grouping และ Paging

Controller

```
public IActionResult GroupByCategory()
{
    // Group by CategoryId
```

```
    var grouped = _context.Products
                          .GroupBy(p => p.CategoryId)
                          .Select(g => new { CategoryId = g.Key, Count = g.Count() })
                          .ToList();
    return View(grouped);
}

public IActionResult Page(int page = 1)
{
    int pageSize = 5;
    var pagedProducts = _context.Products
                                .OrderBy(p => p.Name)
                                .Skip((page-1)*pageSize)
                                .Take(pageSize)
                                .ToList();
    return View(pagedProducts);
}
```

Razor Views

- GroupByCategory.cshtml ⟶ แสดง CategoryId + Count
- Page.cshtml ⟶ แสดง 5 รายการต่อหน้า

ผลการรัน

- แสดงจำนวนสินค้าแต่ละ Category
- Paging แสดงข้อมูลเป็นหน้า ๆ

ตัวอย่างแนวประยุกต์ LINQ Query (3 โปรแกรม)

ตัวอย่าง 1 – E-Commerce Dashboard + LINQ

- **Filter:** Price > 1000
- **Sort:** Name
- **Include:** Category
- **Projection:** Name, CategoryName, Price

Controller

```
var products = _context.Products
                       .Include(p => p.Category)
```

```
        .Where(p => p.Price > 1000)
        .OrderBy(p => p.Name)
        .Select(p => new { p.Name, Category = p.Category.Name, p.Price })
        .ToList();
```

ผลการรัน

- แสดง Product Name, Category Name, Price
- Filter และ Sort ทำงานครบ

ตัวอย่าง 2 – User Management + LINQ

- **Filter:** Email contains "gmail.com"
- **Sort:** Username ascending

Controller

```
var gmailUsers = _context.Users
        .Where(u => u.Email.Contains("gmail.com"))
        .OrderBy(u => u.Username)
        .ToList();
```

ผลการรัน

- แสดงเฉพาะ User Gmail
- เรียงตาม Username

ตัวอย่าง 3 – Multi-Tenant Product + LINQ

- **Dynamic DbContext** ตาม Tenant
- **Filter:** Price > 5000
- **Paging:** 10 รายการต่อหน้า

Controller

```
var dbType = HttpContext.Request.Query["db"];
var products = _context.Products
        .Where(p => p.Price > 5000)
        .OrderBy(p => p.Name)
        .Skip((page-1)*10)
        .Take(10)
        .ToList();
```

ผลการรัน

- /Products?db=A ⟶ ใช้ Tenant A

- /Products?db=B ⟶ ใช้ Tenant B
- Filter + Paging ทำงานครบ

ตัวอย่างแนวประยุกต์ LINQ + EF Core ทั้ง 3 แบบแบบเต็ม Project พร้อม **Razor Views, Layout, Form, TempData, Migration** และสรุป **ผลการรันครบวงจร** ให้คุณเห็นภาพชัดเจน

ตัวอย่างแนวประยุกต์ 1 – E-Commerce Dashboard + LINQ CRUD

Project Structure

```
/ECommerceDashboard
├── Controllers
│   └── ProductsController.cs
├── Models
│   ├── Product.cs
│   └── Category.cs
├── Data
│   └── AppDbContext.cs
├── Views
│   ├── Shared
│   │   └── _Layout.cshtml
│   └── Products
│       ├── Index.cshtml
│       ├── Create.cshtml
│       ├── Edit.cshtml
│       └── Delete.cshtml
├── Program.cs
└── appsettings.json
```

Models

Category.cs

```
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Product> Products { get; set; }
```