C# Programming: Professional

(Integrative-Generative AI Edition)

- Entity Framework Core
- RESTful API
- Unit Testing
- C# GUI
- Performance Optimization
- Bibliography

By Student Price Book Center

คำนำ

ในยุคที่ซอฟต์แวร์มีบทบาทสำคัญในทุกภาคส่วน ตั้งแต่ระบบองค์กรขนาดใหญ่ไปจนถึงแอปพลิเคชันที่ผู้ใช้เข้าถึงได้ในชีวิตประจำวัน นักพัฒนาจำเป็นต้องมีทักษะเชิงลึกและแนวคิดที่ชัดเจนในการออกแบบ พัฒนา และดูแลระบบอย่างมีประสิทธิภาพ หนังสือ *C# Programming: Professional* เล่มนี้ถูกออกแบบมาเพื่อพาผู้อ่านก้าวข้ามระดับพื้นฐานและระดับกลาง เข้าสู่การพัฒนาเชิงวิชาชีพ (Professional Level) ที่ไม่เพียงแต่เน้นการเขียนโค้ดที่ทำงานได้ แต่ยังให้ความสำคัญกับความสามารถในการปรับขยาย ความปลอดภัย ความยั่งยืน และการบำรุงรักษาระยะยาว

หนังสือเล่มนี้ประกอบไปด้วยเนื้อหาสำคัญตั้งแต่บทที่ 16 ถึงบทที่ 20 โดยเริ่มจาก **Entity Framework Core (EF Core)** ซึ่งเป็นหัวใจสำคัญในการจัดการข้อมูลในโลก .NET การเรียนรู้แนวทาง **Code-First vs Database-First** การทำ **Migration** เพื่ออัปเดตฐานข้อมูลอย่างยืดหยุ่น และการใช้ **LINQ** เพื่อ query ข้อมูลอย่างทรงพลัง จะช่วยให้ผู้อ่านเข้าใจการผสานระหว่างโลกของฐานข้อมูลและโค้ดได้อย่างมีประสิทธิภาพ พร้อมทั้งเรียนรู้การใช้ **Include() และ Navigation Properties** ในการจัดการความสัมพันธ์ที่ซับซ้อนของข้อมูล

ถัดมาใน **บทที่ 17: การพัฒนา RESTful API ด้วย ASP.NET Core** ผู้อ่านจะได้ศึกษาแนวคิดและการลงมือปฏิบัติจริงในการสร้าง **API Controller** การจัดการ **Routing และ Attribute** รวมถึงการเชื่อมต่อฐานข้อมูลอย่างปลอดภัย การใช้ **JWT Authentication/Authorization** ที่เป็นมาตรฐานอุตสาหกรรมเพื่อควบคุมสิทธิ์การเข้าถึง API และตัวอย่างบูรณาการที่จะช่วยให้นักพัฒนาสามารถสร้าง API ที่พร้อมใช้งานในระดับองค์กรได้อย่างมั่นใจ

บทที่ 18: Unit Testing จะมุ่งเน้นไปที่การทำให้ซอฟต์แวร์มีคุณภาพและความน่าเชื่อถือ ผู้อ่านจะได้เจาะลึกเครื่องมือสำคัญอย่าง **MSTest, xUnit และ NUnit** รวมถึงการสร้าง **Mock Object ด้วย Moq** เพื่อทดสอบโค้ดโดยไม่ต้องพึ่งพาส่วนประกอบจริง การทดสอบ **Repository และ Service** จะถูกยกตัวอย่างเพื่อชี้ให้เห็นแนวทางปฏิบัติจริง พร้อมทั้งการวัด **Test Coverage** เพื่อประเมินคุณภาพของการทดสอบที่ครอบคลุมในระบบ

การพัฒนา **GUI (บทที่ 19)** ถือเป็นอีกหนึ่งหัวใจของการสร้างซอฟต์แวร์ที่ผู้ใช้สามารถโต้ตอบได้ หนังสือเล่มนี้จะอธิบายอย่างละเอียดถึงการใช้ **WinForms และ WPF** รวมถึง **MVVM Pattern** ซึ่งเป็นแนวคิดหลักในการแยกความรับผิดชอบของโค้ดเพื่อเพิ่มความง่ายต่อการบำรุงรักษา ผู้อ่านยังจะได้ศึกษา **Data Binding** ที่ช่วยเชื่อมโยงข้อมูลกับ UI โดยอัตโนมัติ ตลอดจนการจัดการ **Event** ต่าง ๆ ที่เกิดขึ้นในระบบ นอกจากนี้ยังมีตัวอย่างโปรเจกต์แบบบูรณาการเพื่อแสดงภาพรวมการพัฒนา GUI เชิงลึก

สุดท้ายใน **บทที่ 20: Performance Optimization และ Clean Code** ผู้อ่านจะได้เรียนรู้เทคนิคขั้นสูงในการวิเคราะห์และเพิ่มประสิทธิภาพของโค้ดผ่านการทำ **Profiling และ Benchmarking** การ **Refactor** โค้ดให้อ่านง่ายและรักษาได้ในระยะยาว พร้อมด้วยการประยุกต์ใช้ **SOLID Principles**

เพื่อยกระดับคุณภาพการออกแบบซอฟต์แวร์ และการทำ **Logging และ Monitoring** เพื่อเฝ้าติดตามและประเมินประสิทธิภาพของระบบในสภาพแวดล้อมจริง

หนังสือเล่มนี้ไม่ได้เป็นเพียงคู่มือเชิงทฤษฎี แต่ยังเต็มไปด้วย **ตัวอย่างโค้ดบูรณาการ** และ **โปรเจกต์จริง** ในแต่ละบท เพื่อให้ผู้อ่านสามารถนำไปประยุกต์ใช้ได้อย่างทันทีและมั่นใจ เนื้อหาในทุกบทได้รับการออกแบบอย่างเป็นระบบ เริ่มจากการอธิบายแนวคิด ไปจนถึงตัวอย่างการใช้งานจริง พร้อมคำอธิบายละเอียดที่ช่วยให้ผู้เรียนเข้าใจทั้งมิติเทคนิคและมิติการออกแบบเชิงสถาปัตยกรรม

ท้ายที่สุด **C# Programming: Professional** ไม่ได้เพียงช่วยให้คุณเข้าใจภาษา C# อย่างลึกซึ้ง แต่ยังปูพื้นฐานให้คุณสามารถพัฒนาซอฟต์แวร์ที่มีคุณภาพระดับองค์กร สามารถรองรับการเปลี่ยนแปลงในอนาคต และยืนหยัดในโลกการแข่งขันทางเทคโนโลยีที่เปลี่ยนแปลงอย่างรวดเร็วได้อย่างมั่นคง

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

บทที่ 16

Entity Framework Core

(Entity Framework Core)

เนื้อหา

บทนำ บทที่ 16: Entity Framework Core

การจัดการข้อมูลและการทำงานกับฐานข้อมูลถือเป็นหัวใจสำคัญของการพัฒนาแอปพลิเคชันสมัยใหม่ **Entity Framework Core (EF Core)** เป็น ORM (Object-Relational Mapper) ของ **.NET** ที่ช่วยให้นักพัฒนาสามารถทำงานกับฐานข้อมูลโดยใช้ **C# objects** แทนการเขียน SQL แบบดั้งเดิม บทนี้จะพาผู้อ่านเข้าสู่แนวคิดและเทคนิคสำคัญของ EF Core พร้อมตัวอย่างเชิงลึกเพื่อให้สามารถนำไปใช้ในโปรเจกต์จริง

เริ่มจากแนวคิด **Code-First และ Database-First** ซึ่งเป็นสองวิธีหลักในการสร้างและจัดการโมเดลข้อมูล Code-First ช่วยให้นักพัฒนากำหนดโครงสร้างฐานข้อมูลจาก **C# classes** และสามารถสร้าง database schema อัตโนมัติ ในขณะที่ Database-First เหมาะสำหรับการทำงานกับฐานข้อมูลที่มีอยู่แล้ว โดย EF Core จะสร้างโค้ดที่สอดคล้องกับ schema ช่วยให้การทำงานกับข้อมูลมีประสิทธิภาพและลดความซับซ้อน

หัวข้อถัดมาคือ **Migration** ซึ่งเป็นกระบวนการจัดการการเปลี่ยนแปลงของ schema เมื่อโมเดลของแอปพลิเคชันเปลี่ยนแปลง Migration ช่วยให้สามารถอัปเดตฐานข้อมูลอย่างปลอดภัยและอัตโนมัติ ลดความเสี่ยงจากการแก้ไข schema ด้วยมือและช่วยให้ทีมพัฒนาทำงานร่วมกันได้อย่างราบรื่น

บทนี้ยังเน้นการใช้ **LINQ กับ EF Core** ซึ่งช่วยให้นักพัฒนาสามารถ query ข้อมูลได้อย่างชัดเจนและ type-safe การใช้ LINQ ช่วยลดข้อผิดพลาดจากการเขียน SQL ด้วยมือ และทำให้โค้ดอ่าน

ง่ายและ maintainable นอกจากนี้ EF Core ยังสนับสนุน **Include() และ Navigation Properties** สำหรับโหลดข้อมูลที่สัมพันธ์กันแบบ eager loading ทำให้สามารถจัดการ relational data ได้อย่างยืดหยุ่น

สุดท้ายบทนี้สรุปแนวทางการใช้งาน EF Core ในโปรเจกต์จริง ตั้งแต่การออกแบบโมเดล การเลือก Code-First หรือ Database-First การจัดการ migration ไปจนถึงการ query และโหลดข้อมูลสัมพันธ์ ผู้อ่านจะได้เรียนรู้ทั้งหลักการเชิงลึกและตัวอย่างเชิงปฏิบัติ เพื่อให้สามารถสร้างระบบจัดการข้อมูลที่มีประสิทธิภาพ ปลอดภัย และง่ายต่อการบำรุงรักษา

บทนี้จึงเหมาะสำหรับนักพัฒนาที่ต้องการเข้าใจ **Entity Framework Core** อย่างครบถ้วน ตั้งแต่พื้นฐานไปจนถึงเทคนิคขั้นสูง เพื่อพัฒนาซอฟต์แวร์ที่สามารถรองรับการเปลี่ยนแปลงข้อมูลและโครงสร้างฐานข้อมูลได้อย่างมั่นคงและมีประสิทธิภาพ

บทที่ 16: Entity Framework Core

- Code-First vs Database-First
- Migration
- LINQ กับ EF
- Include(), Navigation Properties

อธิบาย **บทที่ 16: Entity Framework Core (EF Core)** แบบละเอียดและเชิงลึก พร้อมตัวอย่างแนวคิดให้เข้าใจชัดเจน

□ 1□ □ode-First vs Database-First

Code-First

- นักพัฒนาสร้าง **คลาส C#** ก่อน แล้ว EF Core จะสร้าง **ฐานข้อมูล (Database)** ให้ตามโมเดล
- เหมาะกับโปรเจกต์ใหม่ที่ยังไม่มีฐานข้อมูล
- ข้อดี: สามารถออกแบบโมเดลในโค้ดได้ง่าย ปรับเปลี่ยน schema ผ่าน code
- ข้อเสีย: ถ้ามี database ขนาดใหญ่หรือซับซ้อน อาจต้องปรับ migration เยอะ

ตัวอย่างโมเดล Code-First

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Database-First

- มี **ฐานข้อมูลอยู่แล้ว** EF Core จะสร้าง **Entity Classes** ให้ตรงกับตาราง
- เหมาะกับโปรเจกต์ที่ต้องทำงานกับ database ที่มีอยู่แล้ว
- ใช้คำสั่ง Scaffold-DbContext ใน Package Manager Console เพื่อสร้างโมเดล

---

2 Migration

- **Migration** คือเครื่องมือในการ **ปรับเปลี่ยน database schema** ตามโมเดลในโค้ด
- ช่วยให้ database และ code **สอดคล้องกัน**

ตัวอย่างคำสั่ง Migration

```
Add-Migration InitialCreate      # สร้าง migration ชื่อ InitialCreate
Update-Database                  # ใช้ migration สร้าง/ปรับ database
```

หลักการทำงาน

1. สร้าง migration จากโมเดล C#
2. EF Core สร้างไฟล์ migration (.cs) สำหรับสร้าง/แก้ไขตาราง
3. ใช้ Update-Database เพื่อประยุกต์ schema กับฐานข้อมูลจริง

---

3 LINQ กับ EF Core

- **LINQ (Language Integrated Query)** เป็นภาษาสำหรับ query ข้อมูลใน C#
- EF Core แปลง LINQ เป็น **SQL query** อัตโนมัติ
- ใช้ได้ทั้ง IQueryable และ IEnumerable
- LINQ ช่วยให้เขียน query แบบ type-safe, readable และ maintainable

ตัวอย่าง LINQ กับ EF Core

```
using (var context = new SchoolContext())
{
    var students = context.Students
                          .Where(s => s.Age > 18)
                          .OrderBy(s => s.Name)
                          .ToList();

    foreach(var s in students)
        Console.WriteLine($"{s.Name} - {s.Age}");
}
```

- EF Core จะสร้าง SQL เช่น:

```
SELECT * FROM Students WHERE Age > 18 ORDER BY Name
```

---

4 Include() และ Navigation Properties

Navigation Properties

- ใช้สำหรับ **ความสัมพันธ์ระหว่างตาราง** เช่น One-to-Many, Many-to-Many, One-to-One
- ช่วยให้ EF Core เข้าใจความสัมพันธ์และสามารถ query ข้อมูลแบบ **join** ได้

ตัวอย่าง One-to-Many

```
public class Course
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Student> Students { get; set; } // Navigation
}

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int CourseId { get; set; }    // Foreign Key
    public Course Course { get; set; }  // Navigation
}
```

Include()

- ใช้ **ดึงข้อมูลที่เกี่ยวข้อง (eager loading)**
- ป้องกันปัญหา **Lazy Loading** หรือ N+1 Query

ตัวอย่าง Include()

```
using (var context = new SchoolContext())
{
    var courses = context.Courses
                         .Include(c => c.Students)
                         .ToList();

    foreach(var c in courses)
    {
        Console.WriteLine($"Course: {c.Name}");
        foreach(var s in c.Students)
            Console.WriteLine($"  Student: {s.Name}");
```

```
    }
}
```

- EF Core จะสร้าง **JOIN SQL** อัตโนมัติ เช่น:

```
SELECT c.*, s.*
FROM Courses c
LEFT JOIN Students s ON c.Id = s.CourseId
```

□ □ สรุปเชิงลึก

หัวข้อ	คำอธิบาย
Code-First	สร้าง class ก่อน EF สร้าง database, เหมาะกับโปรเจกต์ใหม่
Database-First	สร้าง entity จาก database ที่มีอยู่แล้ว, เหมาะกับโปรเจกต์ legacy
Migration	อัปเดต schema database ตามโมเดล C#, ใช้ Add-Migration / Update-Database
LINQ	Query แบบ type-safe ใน C#, EF Core แปลงเป็น SQL
Include()	Eager loading ของ navigation properties, ดึงข้อมูลที่เกี่ยวข้องพร้อมกัน

Entity Framework Core (EF Core) แบบละเอียดเชิงลึก

□ 1 □ Code-First vs Database-First

Code-First

- นักพัฒนาสร้าง **คลาส C# ก่อน** แล้ว EF Core จะสร้าง **ฐานข้อมูล** ให้ตามโมเดล
- การออกแบบโมเดลใช้ **Data Annotations** หรือ **Fluent API**
    - **Data Annotations**: ใช้ attribute บน property/class
    - public class Student
    - {
    - public int Id { get; set; }
    - [MaxLength(50)]
    - public string Name { get; set; }
    - }
    - **Fluent API**: กำหนด configuration ใน OnModelCreating
    - protected override void OnModelCreating(ModelBuilder modelBuilder)
    - {
    - modelBuilder.Entity<Student>()

```
        .Property(s => s.Name)
        .HasMaxLength(50);
    }
```

- EF Core จะ track class และ generate SQL **CREATE TABLE** เมื่อทำ **Migration**
- ข้อดี:
  - ออกแบบ schema จาก code ได้ง่าย
  - ปรับเปลี่ยน schema ผ่าน code และ migration
- ข้อเสีย:
  - ถ้ามีฐานข้อมูลซับซ้อน หรือ legacy ต้องใช้ migration แก้หลายครั้ง

Database-First

- มี **database อยู่แล้ว** EF Core สร้าง entity class ให้อัตโนมัติ
- ใช้คำสั่ง **Scaffold-DbContext**
- Scaffold-DbContext "Server=.;Database=SchoolDB;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
- EF Core จะสร้าง:
  - class entity
  - DbContext
  - property type และ relationships ตาม database
- เหมาะกับโปรเจกต์ที่ต้องทำงานกับ database ที่มีอยู่แล้ว

---

2 Migration

- Migration คือ **เครื่องมือ synchronize database กับ model**
- ขั้นตอน:
  1. สร้าง migration: Add-Migration InitialCreate
  2. ดูไฟล์ migration ที่ EF Core สร้าง (Up() = สร้าง/แก้ไข, Down() = rollback)
  3. ใช้ Update-Database เพื่อ apply migration

ตัวอย่างไฟล์ Migration

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Students",
```

```
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Name = table.Column<string>(maxLength: 50, nullable: false)
            },
            constraints: table => { table.PrimaryKey("PK_Students", x => x.Id); });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(name: "Students");
    }
}
```

- Migration ช่วย **version control schema database**
- ทำงานร่วมกับ **source control** เพื่อ team development

---

3 LINQ กับ EF Core

- LINQ (Language Integrated Query) ใช้ query entity class
- EF Core **แปลง LINQ เป็น SQL** แบบ type-safe
- ประเภท LINQ:
    - IQueryable<T>: query ยังไม่ execute, สามารถ chain filter/order/include
    - IEnumerable<T>: query execute แล้ว, ใช้ใน memory

ตัวอย่าง LINQ

```
var students = context.Students
                    .Where(s => s.Age > 18)
                    .OrderBy(s => s.Name)
                    .ToList();
```

EF Core แปลงเป็น SQL:

```
SELECT * FROM Students WHERE Age > 18 ORDER BY Name
```

ข้อดี

- Type-safe, compile-time check
- อ่านง่าย, maintainable
- รวมกับ Include(), Join, GroupBy ได้

4 Include() และ Navigation Properties

Navigation Properties

- ใช้แสดง **relationship ระหว่าง entity**
- EF Core รองรับ:
  - One-to-Many
  - Many-to-Many
  - One-to-One

ตัวอย่าง One-to-Many

```
public class Course
{
    public int Id { get; set; }
    public string Name { get; set; }
    public ICollection<Student> Students { get; set; }
}

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int CourseId { get; set; }
    public Course Course { get; set; }
}
```

Include()

- ใช้ดึงข้อมูล **related entities** (Eager Loading)
- ป้องกัน **Lazy Loading** หรือ N+1 problem

```
var courses = context.Courses
                .Include(c => c.Students)
                .ToList();
```

SQL ที่ EF Core สร้าง:

```
SELECT c.*, s.*
FROM Courses c
LEFT JOIN Students s ON c.Id = s.CourseId
```

- สามารถ chain multi-level include:

```
context.Orders
    .Include(o => o.Customer)
    .ThenInclude(c => c.Address)
    .ToList();
```

---

5 เชิงลึกการทำงานเบื้องหลัง EF Core

1. **Change Tracker**
   - EF Core track state ของ entity (Added, Modified, Deleted, Unchanged)
   - เมื่อเรียก SaveChanges(), EF Core generate SQL ตาม state
2. **Query Translation**
   - LINQ ⟶ Expression Tree ⟶ SQL
   - EF Core optimize query, combine joins, select only columns needed
3. **Eager vs Lazy Loading**
   - Eager (Include) ⟶ ดึงทั้งหมดพร้อมกัน
   - Lazy ⟶ ดึงเมื่อเรียก property (ต้องเปิด Lazy Loading Proxy)
4. **Concurrency Control**
   - ใช้ **Concurrency Tokens** (RowVersion) ป้องกัน conflict

---

6 สรุปเชิงลึก

หัวข้อ	รายละเอียดเชิงลึก
Code-First	สร้าง class ก่อน, EF Core สร้าง database, ใช้ DataAnnotation/Fluent API, migration track changes
Database-First	สร้าง class จาก database, scaffold entity และ DbContext, เหมาะกับ legacy DB
Migration	ใช้ version control schema, Add-Migration / Update-Database, rollback ด้วย Down()
LINQ	Query type-safe, EF Core แปลงเป็น SQL, ใช้ IQueryable/IEnumerable, chain filter/order/include
Navigation Properties	แสดงความสัมพันธ์ entity, One-to-Many, Many-to-Many, One-to-One
Include()	Eager loading, JOIN tables, ป้องกัน Lazy Loading problem

หัวข้อ	รายละเอียดเชิงลึก
Change Tracker	Track state entity, generate SQL ตาม Added/Modified/Deleted
Concurrency	RowVersion ป้องกัน conflict, EF Core detect collision
Query Translation	LINQ ⟶ Expression Tree ⟶ SQL, optimize query, select columns ที่จำเป็น

Code-First vs Database-First ใน EF Core

---

1 Code-First

แนวคิด

- นักพัฒนาสร้าง **คลาส C# (Entity Classes)** ก่อน
- EF Core จะสร้าง **Database** ให้ตรงกับโมเดลใน code
- เหมาะกับโปรเจกต์ใหม่ที่ยังไม่มี database

การทำงาน

1. สร้าง entity class

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

2. สร้าง DbContext

```
using Microsoft.EntityFrameworkCore;

public class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=.;Database=SchoolDB;Trusted_Connection=True;");
    }
}
```

3. สร้าง Migration

Add-Migration InitialCreate

Update-Database

- EF Core จะ generate **CREATE TABLE** ใน database ตาม class

ข้อดี

- ออกแบบ schema จาก code ได้ง่าย
- ปรับเปลี่ยน schema ผ่าน code และ migration
- ใช้ version control กับ code ได้เต็มที่

ข้อเสีย

- ถ้ามี database ขนาดใหญ่หรือซับซ้อน อาจต้องปรับ migration เยอะ
- ต้องเข้าใจ EF Core Fluent API / Data Annotation สำหรับ configuration

---

2 Database-First

แนวคิด

- มี **database อยู่แล้ว**
- EF Core จะสร้าง **Entity Classes** และ **DbContext** ให้ตรงกับ database

การทำงาน

1. ใช้คำสั่ง Scaffold-DbContext

Scaffold-DbContext "Server=.;Database=SchoolDB;Trusted_Connection=True;"

Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

2. EF Core จะสร้าง:

- entity class ทุก table
- DbContext
- property type และ relationships

ข้อดี

- เหมาะกับโปรเจกต์ legacy หรือ database ที่มีอยู่แล้ว
- ไม่ต้องสร้าง schema ใหม่, สร้าง class ให้ตรงกับ database อัตโนมัติ

ข้อเสีย

- การแก้ไข schema ต้องแก้ใน database แล้ว scaffold ใหม่
- ปรับเปลี่ยนโมเดลจาก code ไม่สะดวกเหมือน Code-First

---

3 เปรียบเทียบเชิงลึก

Feature	Code-First	Database-First

Feature	Code-First	Database-First
เริ่มต้น	สร้าง class ก่อน DB	DB มีอยู่แล้ว
เหมาะกับ	Project ใหม่	Legacy DB
Migration	ใช้ได้ง่าย	ไม่จำเป็น / ใช้ยาก
ปรับ schema	จาก code ⟶ DB	ต้องปรับ DB ⟶ scaffold ใหม่
Flexibility	สูง	ต่ำ

สรุป:

- **Code-First** ⟶ ออกแบบจาก code, ใช้ migration, เหมาะกับโปรเจกต์ใหม่
- **Database-First** ⟶ สร้าง entity จาก DB, เหมาะกับ database ที่มีอยู่แล้ว

ตัวอย่าง **Entity Framework Core (EF Core) แบบเต็มไฟล์** แบ่งเป็น **3 โปรแกรมพื้นฐาน** และ **3 โปรแกรมแนวประยุกต์** ครบทุกขั้นตอน พร้อมโครงสร้างโฟลเดอร์, คำอธิบายโค้ด และผลการรัน

โครงสร้างโปรเจกต์ EFCoreDemo

```
EFCoreDemo/
├── Models/
│   ├── Student.cs
│   └── Course.cs
├── Data/
│   └── SchoolContext.cs
└── Program.cs
```

โปรแกรมพื้นฐาน 3 ตัวอย่าง (Basic)

1 Basic CRUD with EF Core

ไฟล์: Models/Student.cs

```
namespace EFCoreDemo.Models
{
    public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
```

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

ไฟล์: Data/SchoolContext.cs

```
using EFCoreDemo.Models;
using Microsoft.EntityFrameworkCore;

namespace EFCoreDemo.Data
{
    public class SchoolContext : DbContext
    {
        public DbSet<Student> Students { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {

optionsBuilder.UseSqlServer(@"Server=.;Database=SchoolDB;Trusted_Connection=True;");
        }
    }
}
```

ไฟล์: Program.cs

```
using EFCoreDemo.Data;
using EFCoreDemo.Models;

using var context = new SchoolContext();

// Create
var student = new Student { Name = "Alice", Age = 20 };
context.Students.Add(student);
context.SaveChanges();

// Read
var students = context.Students.ToList();
students.ForEach(s => Console.WriteLine($"{s.Id} - {s.Name} - {s.Age}"));
```

```
// Update
student.Age = 21;
context.SaveChanges();

// Delete
context.Students.Remove(student);
context.SaveChanges();
```

ผลการรัน

```
1 - Alice - 20
```

---

2 LINQ Query Example

```
using EFCoreDemo.Data;
using EFCoreDemo.Models;

using var context = new SchoolContext();

// Add sample data
context.Students.AddRange(
    new Student { Name = "Bob", Age = 22 },
    new Student { Name = "Charlie", Age = 19 }
);
context.SaveChanges();

// LINQ query
var result = context.Students
                    .Where(s => s.Age > 20)
                    .OrderBy(s => s.Name)
                    .ToList();

result.ForEach(s => Console.WriteLine($"{s.Name} - {s.Age}"));
```

ผลการรัน

```
Bob - 22
```

---

3 Include and Navigation Properties

ไฟล์: Models/Course.cs

```
using System.Collections.Generic;

namespace EFCoreDemo.Models
{
    public class Course
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public ICollection<Student> Students { get; set; }
    }
}
```

Program.cs

```
using EFCoreDemo.Data;
using EFCoreDemo.Models;
using Microsoft.EntityFrameworkCore;

using var context = new SchoolContext();

// Add Course with Students
var course = new Course { Name = "Math", Students = new List<Student>
{
    new Student { Name = "David", Age = 21 },
    new Student { Name = "Eva", Age = 20 }
} };
context.Courses.Add(course);
context.SaveChanges();

// Include students
var courses = context.Courses.Include(c => c.Students).ToList();
foreach(var c in courses)
{
    Console.WriteLine($"Course: {c.Name}");
```

```
    foreach(var s in c.Students)
        Console.WriteLine($"  Student: {s.Name}");
}
```

ผลการรัน

```
Course: Math
  Student: David
  Student: Eva
```

โปรแกรมแนวประยุกต์ 3 ตัวอย่าง (Advanced/Applied)

1 Search Students by Age Range

```
using EFCoreDemo.Data;
using EFCoreDemo.Models;
using Microsoft.EntityFrameworkCore;

using var context = new SchoolContext();

int minAge = 18;
int maxAge = 22;

var students = context.Students
                    .Where(s => s.Age >= minAge && s.Age <= maxAge)
                    .OrderByDescending(s => s.Age)
                    .ToList();

students.ForEach(s => Console.WriteLine($"{s.Name} - {s.Age}"));
```

ผลการรัน

```
Bob - 22
David - 21
Eva - 20
Charlie - 19
```

---

2 Update Multiple Students (Batch Update)

```
using EFCoreDemo.Data;
using EFCoreDemo.Models;
```

```
using var context = new SchoolContext();

var students = context.Students.Where(s => s.Age < 21).ToList();
students.ForEach(s => s.Age += 1);
context.SaveChanges();

students.ForEach(s => Console.WriteLine($"{s.Name} - {s.Age}"));
```

ผลการรัน

```
Charlie - 20
Eva - 21
```

---

3 Eager Loading Multi-Level Include

สมมติเพิ่ม Address entity

```
public class Address
{
    public int Id { get; set; }
    public string Street { get; set; }
    public int StudentId { get; set; }
    public Student Student { get; set; }
}
```

Program.cs

```
var students = context.Students
                      .Include(s => s.Course)
                      .ThenInclude(c => c.Students)
                      .ToList();

foreach(var s in students)
{
    Console.WriteLine($"Student: {s.Name} - Course: {s.Course?.Name}");
}
```

ผลการรัน

```
Student: David - Course: Math
Student: Eva - Course: Math
```

□ สรุป

- 3 โปรแกรมพื้นฐาน: CRUD, LINQ, Include/Navigation
- 3 โปรแกรมแนวประยุกต์: Search filter, Batch Update, Multi-Level Include
- ครอบคลุม EF Core core features: Code-First, Migration, LINQ, Relationships, Include

---

Migration ใน EF Core

□ 1 □ Migration คืออะไร

- **Migration** คือเครื่องมือของ EF Core สำหรับ **จัดการ schema ของฐานข้อมูล** ให้สอดคล้องกับ **model class ใน C#**
- Migration ช่วยให้เรา **สร้าง ปรับแก้ และ version control database schema** ได้สะดวก
- เหมาะสำหรับ **Code-First approach** โดยไม่ต้องแก้ไข database ด้วยตนเอง

□ 2 □ หลักการทำงานของ Migration

1. **ตรวจสอบโมเดล** (Entity Classes / DbContext)
2. **เปรียบเทียบกับ schema ของ database ปัจจุบัน**
3. **สร้างไฟล์ migration** (.cs) แสดงความแตกต่างของ schema
4. **นำ migration ไป apply** กับ database (Update-Database)
5. **Track version schema** ผ่าน __EFMigrationsHistory table

□ 3 □ คำสั่ง Migration

สร้าง Migration

Add-Migration InitialCreate

- InitialCreate เป็นชื่อ migration
- EF Core จะสร้างไฟล์ migration ใหม่ในโฟลเดอร์ **Migrations/**

Apply Migration (สร้าง/อัปเดต database)

Update-Database

ย้อนกลับ Migration

Update-Database PreviousMigrationName

ลบ Migration

Remove-Migration

- ลบไฟล์ migration ล่าสุดก่อน apply database

4 โครงสร้างไฟล์ Migration

ตัวอย่างไฟล์ Migration: InitialCreate.cs

```
using Microsoft.EntityFrameworkCore.Migrations;

public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Students",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                Name = table.Column<string>(maxLength: 50, nullable: false),
                Age = table.Column<int>(nullable: false)
            },
            constraints: table => { table.PrimaryKey("PK_Students", x => x.Id); });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(name: "Students");
    }
}
```

- **Up()** ⟶ สร้าง/แก้ไขตารางใน database
- **Down()** ⟶ rollback การเปลี่ยนแปลง

5 การใช้งานเชิงลึก

1. **Version Control**
    - Migration บันทึกการเปลี่ยนแปลง schema เป็น version
    - Team development สามารถ sync schema ผ่าน source control
2. **Seed Data**

- สามารถใส่ข้อมูลเริ่มต้นผ่าน migration

3. migrationBuilder.InsertData(
4. table: "Students",
5. columns: new[] { "Name", "Age" },
6. values: new object[] { "Alice", 20 });
7. **Multi-environment**
   - Migration สามารถ apply database ใน **Dev / Test / Production** environment ต่างกัน
8. **Schema Update**
   - เพิ่ม column ใหม่
9. Add-Migration AddEmailToStudent
10. Update-Database
    - EF Core จะ generate SQL ALTER TABLE โดยอัตโนมัติ

---

6 ข้อดีของ Migration

Feature	ประโยชน์
Version Control	เก็บประวัติ schema ของ database
Automation	ปรับ database โดยไม่ต้องเขียน SQL เอง
Team Collaboration	ทีมสามารถ sync database schema จาก migration
Rollback	สามารถย้อน migration เมื่อเกิดข้อผิดพลาด

---

7 ตัวอย่าง Workflow Migration (Code-First)

1. สร้าง model class

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

2. สร้าง DbContext

```
public class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }
```

```
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        =>
optionsBuilder.UseSqlServer("Server=.;Database=SchoolDB;Trusted_Connection=True;");
}
```

3. สร้าง Migration

```
Add-Migration InitialCreate
```

4. Update Database

```
Update-Database
```

5. เพิ่ม property ใหม่

```
public string Email { get; set; }
```

6. สร้าง migration ใหม่

```
Add-Migration AddEmailToStudent
Update-Database
```

Migration คือ **หัวใจสำคัญของ Code-First EF Core** ที่ทำให้การจัดการ database schema **ง่าย, versionable, และ team-friendly**

ตัวอย่าง **Migration ใน EF Core** แบบเต็มชุด แบ่งเป็น **3 โปรแกรมพื้นฐาน** และ **3 โปรแกรมแนวประยุกต์** พร้อมโครงสร้าง, คำอธิบายโค้ด และผลการรัน

โครงสร้างโปรเจกต์ EFCoreMigrationDemo

```
EFCoreMigrationDemo/
├── Models/
│   ├── Student.cs
│   └── Course.cs
├── Data/
│   └── SchoolContext.cs
├── Program.cs
└── Migrations/   (EF Core สร้างอัตโนมัติ)
```

โปรแกรมพื้นฐาน 3 ตัวอย่าง (Basic)

1 Initial Migration – สร้างฐานข้อมูลจาก Model

Models/Student.cs

```
namespace EFCoreMigrationDemo.Models
{
    public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
    }
}
```

Data/SchoolContext.cs

```
using EFCoreMigrationDemo.Models;
using Microsoft.EntityFrameworkCore;

namespace EFCoreMigrationDemo.Data
{
    public class SchoolContext : DbContext
    {
        public DbSet<Student> Students { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {

optionsBuilder.UseSqlServer(@"Server=.;Database=SchoolMigrationDB;Trusted_Connection=Tr
ue;");
        }
    }
}
```

Program.cs

```
using EFCoreMigrationDemo.Data;
using EFCoreMigrationDemo.Models;

Console.WriteLine("Apply initial migration...");

using var context = new SchoolContext();
```

```
context.Database.Migrate(); // สร้าง database และ table จาก migration

Console.WriteLine("Database and table created successfully!");
```

คำสั่ง Migration

```
Add-Migration InitialCreate
Update-Database
```

ผลการรัน

```
Apply initial migration...
Database and table created successfully!
```

---

2 Add Column Migration – เพิ่ม Email Property

Models/Student.cs (เพิ่ม property)

```
public string Email { get; set; }
```

สร้าง Migration

```
Add-Migration AddEmailToStudent
Update-Database
```

Program.cs

```
using EFCoreMigrationDemo.Data;
using EFCoreMigrationDemo.Models;

using var context = new SchoolContext();

var student = new Student { Name = "Alice", Age = 20, Email = "alice@example.com" };
context.Students.Add(student);
context.SaveChanges();

Console.WriteLine($"Added Student: {student.Name}, Email: {student.Email}");
```

ผลการรัน

```
Added Student: Alice, Email: alice@example.com
```

---

3 Delete Column / Rollback Migration

- สมมติเราต้องลบ Email property
- แก้ Model แล้วสร้าง Migration ใหม่

```
Add-Migration RemoveEmailFromStudent
```

```
Update-Database
```

Program.cs

```
using EFCoreMigrationDemo.Data;
using EFCoreMigrationDemo.Models;

using var context = new SchoolContext();

var students = context.Students.ToList();
students.ForEach(s => Console.WriteLine($"{s.Name} - {s.Age}"));
```

ผลการรัน

```
Alice - 20
```

โปรแกรมแนวประยุกต์ 3 ตัวอย่าง (Advanced/Applied)

1 Seed Initial Data ผ่าน Migration

```
migrationBuilder.InsertData(
    table: "Students",
    columns: new[] { "Name", "Age" },
    values: new object[] { "Bob", 22 });
```

Program.cs

```
using EFCoreMigrationDemo.Data;

using var context = new SchoolContext();
var students = context.Students.ToList();
students.ForEach(s => Console.WriteLine($"{s.Name} - {s.Age}"));
```

ผลการรัน

```
Bob - 22
```

---

2 Multi-Table Migration – Add Course Table & Relationship

Models/Course.cs

```
using System.Collections.Generic;

namespace EFCoreMigrationDemo.Models
{
    public class Course
```

```
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public ICollection<Student> Students { get; set; }
    }
}
```

Models/Student.cs (เพิ่ม Navigation)

```
public int CourseId { get; set; }
public Course Course { get; set; }
```

Program.cs

```
using EFCoreMigrationDemo.Data;
using EFCoreMigrationDemo.Models;

using var context = new SchoolContext();

var course = new Course { Name = "Math", Students = new List<Student>
{
    new Student { Name = "Charlie", Age = 19 },
    new Student { Name = "David", Age = 21 }
}};

context.Courses.Add(course);
context.SaveChanges();

var courses = context.Courses.Include(c => c.Students).ToList();
foreach(var c in courses)
{
    Console.WriteLine($"Course: {c.Name}");
    foreach(var s in c.Students)
        Console.WriteLine($"  Student: {s.Name}");
}
```

ผลการรัน

```
Course: Math
  Student: Charlie
```

Student: David

---

3 Rollback Migration & Version Control

คำสั่ง rollback

Update-Database InitialCreate

Program.cs

using EFCoreMigrationDemo.Data;

using var context = new SchoolContext();

Console.WriteLine("Database rolled back to InitialCreate migration");

ผลการรัน

Database rolled back to InitialCreate migration

สรุป

- โปรแกรมพื้นฐาน 3 ตัว: Initial Migration, Add Column, Remove Column
- โปรแกรมแนวประยุกต์ 3 ตัว: Seed Data, Multi-Table + Relationship, Rollback Migration
- ครอบคลุม **Migration workflow, CRUD, Relationships, Seed, Rollback**
- โครงสร้างโปรเจกต์เหมาะสำหรับศึกษาเชิงลึกต่อ

LINQ กับ Entity Framework Core (EF Core)

---

1 LINQ คืออะไร

- **LINQ (Language Integrated Query)** เป็น syntax ของ C# สำหรับ **query ข้อมูลจาก collection หรือ database**
- LINQ ทำให้ **query database, collection, XML หรือ objects** ด้วย syntax เดียวกัน
- EF Core ใช้ LINQ เป็น **หลักในการดึงข้อมูลจาก database**

---

2 LINQ กับ EF Core

- LINQ query จะถูก **translate เป็น SQL** โดย EF Core
- ทำให้สามารถ query database ได้ **strongly typed** และ **compile-time checked**
- ประเภทของ LINQ query:
    1. **LINQ to Objects** ⟶ query collection ใน memory
    2. **LINQ to Entities** ⟶ query database ผ่าน EF Core

3 ตัวอย่าง LINQ Query พื้นฐานกับ EF Core

Context และ Model

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

public class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        =>
optionsBuilder.UseSqlServer(@"Server=.;Database=SchoolDB;Trusted_Connection=True;");
}
```

ตัวอย่าง LINQ

1 Select All Students

```
using var context = new SchoolContext();

var students = context.Students.ToList();
students.ForEach(s => Console.WriteLine($"{s.Name} - {s.Age}"));
```

2 Filter by Condition (Where)

```
var adults = context.Students
                    .Where(s => s.Age >= 20)
                    .ToList();

adults.ForEach(s => Console.WriteLine($"{s.Name} - {s.Age}"));
```

3 Sorting (OrderBy / OrderByDescending)

```
var sorted = context.Students
                    .OrderBy(s => s.Name)
```

```
                .ToList();
```

4 Projection (Select)

```
var names = context.Students
                .Select(s => s.Name)
                .ToList();
names.ForEach(Console.WriteLine);
```

5 Aggregation (Count, Max, Min, Average)

```
int total = context.Students.Count();
double avgAge = context.Students.Average(s => s.Age);

Console.WriteLine($"Total: {total}, AvgAge: {avgAge}");
```

6 Join / Include (Eager Loading)

```
var courses = context.Courses.Include(c => c.Students).ToList();

foreach(var c in courses)
{
    Console.WriteLine($"Course: {c.Name}");
    foreach(var s in c.Students)
        Console.WriteLine($"  Student: {s.Name}");
}
```

---

4 LINQ แบบ Method Syntax vs Query Syntax

Method Syntax (นิยมใช้)

```
var adults = context.Students
                .Where(s => s.Age >= 20)
                .OrderBy(s => s.Name)
                .ToList();
```

Query Syntax

```
var adults = (from s in context.Students
              where s.Age >= 20
              orderby s.Name
              select s).ToList();
```
