ASP.NET Core Programming: ADVANCE

(Integrative-Generative AI Edition)

Advanced EF Core
SignalR and Real-Time Communication
File Handling and Uploads
Caching and Performance Optimization
Security Best Practices
Bibliography

Student Price Book Center

ai

คำนำ

การพัฒนาแอปพลิเคชันด้วย **ASP.NET Core** ก้าวหน้าไปอย่างต่อเนื่องตามความต้องการของโลกซอฟต์แวร์ยุคใหม่ ที่ไม่เพียงแต่ต้องการประสิทธิภาพ ความเร็ว และการรองรับผู้ใช้งานจำนวนมากเท่านั้น แต่ยังต้องคำนึงถึงความปลอดภัย ความสามารถในการปรับขยาย (scalability) และการบำรุงรักษาในระยะยาว หนังสือเล่มนี้ **ASP.NET Core Programming: Advance** ถูกเขียนขึ้นเพื่อเติมเต็มความรู้เชิงลึกสำหรับนักพัฒนาและสถาปนิกซอฟต์แวร์ที่ต้องการก้าวข้ามจากระดับพื้นฐานไปสู่การประยุกต์ใช้ ASP.NET Core ในเชิงลึกและระดับองค์กร

ในบทที่ 11 ผู้อ่านจะได้เจาะลึก **Advanced EF Core** ซึ่งเป็นหัวใจสำคัญของการทำงานกับฐานข้อมูลใน .NET การเข้าใจ **Eager vs Lazy Loading** การสร้างความสัมพันธ์แบบ **One-to-Many และ Many-to-Many**, การใช้งาน **Stored Procedures**, และเทคนิคการ **Query Optimization** จะทำให้นักพัฒนาสามารถสร้างระบบที่ตอบสนองรวดเร็วและจัดการข้อมูลได้อย่างมีประสิทธิภาพยิ่งขึ้น
บทที่ 12 มุ่งเน้นไปที่ **SignalR and Real-Time Communication** เทคโนโลยีที่ช่วยให้การสื่อสารแบบเรียลไทม์เป็นเรื่องง่าย ไม่ว่าจะเป็นการสร้าง **Real-Time Chat**, การเชื่อมต่อ **Client ด้วย JavaScript**, หรือการปรับขยายระบบด้วย **Redis และ Azure SignalR** ผู้อ่านจะได้เรียนรู้แนวคิดเชิงลึกพร้อมตัวอย่างจริงที่สามารถนำไปต่อยอดได้ทันที

หัวข้อสำคัญอีกประการที่นักพัฒนาไม่ควรมองข้ามคือ **File Handling and Uploads** ซึ่งครอบคลุมอยู่ในบทที่ 13 การเรียนรู้การจัดการไฟล์ การอัปโหลด การเก็บไฟล์ใน **wwwroot** หรือ **Cloud Storage** และการสร้าง **File API** ถือเป็นทักษะที่จำเป็นสำหรับระบบที่มีการรับส่งไฟล์จากผู้ใช้จำนวนมาก เนื้อหาในบทนี้จะช่วยให้ผู้อ่านสามารถออกแบบระบบจัดการไฟล์ที่ปลอดภัย มีมาตรฐาน และรองรับการใช้งานจริง

ต่อมาในบทที่ 14 ผู้อ่านจะได้เรียนรู้เรื่อง **Caching and Performance Optimization** ซึ่งเป็นหัวใจของการสร้างระบบที่มีประสิทธิภาพสูง ทั้งการใช้งาน **In-Memory Cache**, **Distributed Cache (Redis, SQL Server)**, **Response/Output Caching**, ตลอดจนการ **Optimize Pipeline** เพื่อปรับปรุงประสิทธิภาพของ Middleware และกระบวนการทำงานโดยรวม เทคนิคเหล่านี้จะทำให้ระบบสามารถตอบสนองได้รวดเร็วขึ้น รองรับผู้ใช้งานจำนวนมาก และลดภาระของเซิร์ฟเวอร์ได้อย่างมีประสิทธิผล

บทที่ 15 ว่าด้วย **Security Best Practices** ซึ่งเป็นสิ่งที่นักพัฒนาทุกคนควรให้ความสำคัญสูงสุด เนื้อหาครอบคลุมการป้องกัน **CSRF, XSS และ SQL Injection**, การใช้ **Data Protection API** เพื่อรักษาข้อมูลสำคัญ, การควบคุมคำขอด้วย **Rate Limiting และ Throttling**, และการบังคับใช้งาน **HTTPS และ HSTS** เพื่อความปลอดภัยของการสื่อสารบนเครือข่าย ผู้อ่านจะได้เรียนรู้แนวทางปฏิบัติที่สามารถนำไปใช้กับโปรเจกต์จริงเพื่อยกระดับความปลอดภัยของระบบ

สิ่งที่ทำให้หนังสือเล่มนี้มีความพิเศษคือการนำเสนอทั้ง **แนวคิดเชิงทฤษฎีและตัวอย่างบูรณาการ** ในแต่ละบท เพื่อให้ผู้อ่านไม่เพียงแต่เข้าใจกลไกของเทคโนโลยี แต่ยังสามารถนำความรู้ไปประยุกต์ใช้ได้จริงกับโครงการพัฒนาแอปพลิเคชันในระดับองค์กร

หวังเป็นอย่างยิ่งว่าหนังสือเล่มนี้จะเป็นคู่มือที่มีคุณค่า ช่วยให้นักพัฒนา ASP.NET Core สามารถยกระดับความรู้และทักษะไปสู่ระดับที่สูงขึ้น พร้อมสร้างแอปพลิเคชันที่มี **ประสิทธิภาพสูง มั่นคงปลอดภัย และรองรับการขยายตัว** ได้อย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 11

Advanced EF Core

(Advanced EF Core)

เนื้อหา

- Advanced EF Core
- Advanced EF Core (เชิงลึก)
- Eager vs Lazy Loading ใน Entity Framework Core (EF Core)
- Relationships (One-to-Many, Many-to-Many)
- Stored Procedures ใน EF Core
- Query Optimization ใน EF Core
- ตัวอย่างบูรณาการ

บทนำ: Advanced EF Core

Entity Framework Core (EF Core) ได้กลายเป็นเครื่องมือสำคัญสำหรับนักพัฒนา .NET ที่ต้องการสร้างแอปพลิเคชันที่มีการจัดการข้อมูลอย่างมีประสิทธิภาพ ด้วยความสามารถในการเชื่อมต่อกับฐานข้อมูลหลายประเภทและการทำงานแบบ Object-Relational Mapping (ORM) EF Core ช่วยลดความซับซ้อนของการเขียน SQL ดิบและมุ่งเน้นไปที่การพัฒนาโค้ดที่ชัดเจนและบำรุงรักษาง่าย

ในบทนี้ เราจะเริ่มต้นด้วยแนวคิดเรื่อง **Eager Loading และ Lazy Loading** ซึ่งเป็นเทคนิคในการโหลดข้อมูลที่เกี่ยวข้องกับ Entity การเข้าใจความแตกต่างระหว่างสองวิธีนี้มีความสำคัญต่อประสิทธิภาพของแอปพลิเคชัน โดย Eager Loading จะดึงข้อมูลทั้งหมดพร้อมกันในครั้งเดียว ขณะที่ Lazy Loading จะดึงข้อมูลเมื่อจำเป็นเท่านั้น การเลือกใช้วิธีที่เหมาะสมช่วยลดปัญหาการโหลดซ้ำซ้อนและเพิ่มความเร็วในการเข้าถึงข้อมูล

ต่อมา เราจะเจาะลึกเรื่อง **Relationships** ใน EF Core ซึ่งครอบคลุมทั้ง One-to-Many และ Many-to-Many Relationships การสร้างความสัมพันธ์ที่ถูกต้องและเหมาะสมช่วยให้การจัดการข้อมูลมีความยืดหยุ่นและถูกต้องตามโครงสร้างธุรกิจ นอกจากนี้ยังช่วยให้การเขียน Query ทำได้ง่ายขึ้นและลดความซับซ้อนของการประมวลผลข้อมูล

อีกหนึ่งหัวข้อสำคัญคือ **Stored Procedures** การใช้ Stored Procedures ร่วมกับ EF Core ช่วยให้การทำงานบางอย่างมีประสิทธิภาพมากขึ้น และสามารถจัดการกับการประมวลผลที่ซับซ้อนภายในฐานข้อมูลได้โดยตรง การเรียนรู้วิธีการเรียกใช้งาน Stored Procedures และการแมปผลลัพธ์กับ Entity จะช่วยให้การพัฒนาแอปพลิเคชันมีความยืดหยุ่นมากขึ้น

นอกจากนี้ บทนี้ยังครอบคลุม **Query Optimization** เทคนิคต่าง ๆ ในการปรับปรุงประสิทธิภาพของ Query เช่น การใช้ Index, การเลือกใช้ Projection และการลดจำนวน Query ที่ถูกส่งไปยังฐานข้อมูล การทำความเข้าใจเรื่อง Query Optimization เป็นสิ่งจำเป็นสำหรับการพัฒนาแอปพลิเคชันที่ต้องจัดการข้อมูลจำนวนมาก

การเรียนรู้เนื้อหาในบทนี้จะช่วยให้นักพัฒนาสามารถออกแบบระบบข้อมูลที่มีประสิทธิภาพ ปลอดภัย และสามารถปรับขยายได้ตามความต้องการทางธุรกิจ การเข้าใจแนวคิดและเทคนิคขั้นสูงเหล่านี้ช่วยให้การพัฒนาแอปพลิเคชันมีมาตรฐานและง่ายต่อการบำรุงรักษาในระยะยาว
ท้ายที่สุด บทนี้ไม่เพียงแค่ให้ความรู้ทางเทคนิค แต่ยังสนับสนุนแนวคิดเชิงปฏิบัติที่นักพัฒนาสามารถนำไปใช้กับโปรเจกต์จริงได้ ความเข้าใจเชิงลึกเกี่ยวกับ EF Core จะทำให้ผู้เรียนสามารถสร้างโค้ดที่มีคุณภาพสูงและสามารถรับมือกับปัญหาที่ซับซ้อนของฐานข้อมูลได้อย่างมั่นใจ

ด้วยความรู้และแนวทางปฏิบัติที่ครอบคลุมในบทนี้ ผู้เรียนจะมีพื้นฐานที่แข็งแรงสำหรับการก้าวไปสู่การพัฒนาแอปพลิเคชัน .NET ที่เต็มไปด้วยประสิทธิภาพและความยืดหยุ่น พร้อมทั้งสามารถจัดการข้อมูลได้อย่างชาญฉลาดและมีประสิทธิผล

Advanced EF Core

Advanced EF Core

- Eager vs Lazy Loading
- Relationships (One-to-Many, Many-to-Many)
- Stored Procedures
- Query Optimization

Advanced EF Core แบบเชิงลึก ครอบคลุมทั้งแนวคิด + การใช้งานจริง + Best Practices

1. Eager vs Lazy Loading

การโหลดข้อมูลสัมพันธ์ (related data) มี 3 วิธีหลักใน EF Core:

☐ **Lazy Loading**

- คือการโหลดข้อมูลเฉพาะ entity หลักก่อน
- ความสัมพันธ์ (navigation properties) จะถูกโหลด **ก็ต่อเมื่อถูกเข้าถึงจริง**
- ต้องเปิดใช้งานด้วย Microsoft.EntityFrameworkCore.Proxies และ UseLazyLoadingProxies()

ตัวอย่าง

```
var blog = context.Blogs.First();
var posts = blog.Posts; // EF Core จะ query Posts ตอนนี้
```

☐ **ข้อดี**: query แรกเล็ก ไม่โหลดข้อมูลที่ไม่จำเป็น

☐ **ข้อเสีย**: อาจเกิดปัญหา **N+1 query problem** ถ้ามีการเข้าถึง collection หลายครั้งใน loop

☐ **Eager Loading**

- คือการโหลดข้อมูลสัมพันธ์ไปพร้อมกันใน query แรก
- ใช้ Include() และ ThenInclude()

ตัวอย่าง

```
var blogs = context.Blogs
                .Include(b => b.Posts)
                .ToList();
```

☐ **ข้อดี**: ลดจำนวน query, เหมาะเมื่อเรารู้ว่าต้องใช้ข้อมูลแน่นอน

☐ **ข้อเสีย**: query ใหญ่ขึ้น, อาจโหลดข้อมูลเกินความจำเป็น

☐ **Explicit Loading**

- ใช้ Entry(...).Collection(...).Load() หรือ Entry(...).Reference(...).Load() เพื่อบังคับโหลดข้อมูล

ตัวอย่าง

```
var blog = context.Blogs.First();
context.Entry(blog).Collection(b => b.Posts).Load();
```

☐ ใช้เมื่อ query แรกต้องการควบคุมว่าจะโหลดความสัมพันธ์ใดเป็นพิเศษ

---

2. Relationships (One-to-Many, Many-to-Many)

☐ **One-to-Many**

ตัวอย่าง: Blog ⟶ Posts

```
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Post> Posts { get; set; } = new();
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int BlogId { get; set; }   // Foreign Key
    public Blog Blog { get; set; }
```

```
}
```

Fluent API

```
modelBuilder.Entity<Post>()
    .HasOne(p => p.Blog)
    .WithMany(b => b.Posts)
    .HasForeignKey(p => p.BlogId);
```

☐ **Many-to-Many**

ตั้งแต่ EF Core 5+ รองรับ **Skip Navigation** ไม่ต้องสร้าง entity join table เอง

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Course> Courses { get; set; } = new();
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }
    public List<Student> Students { get; set; } = new();
}
```

Fluent API

```
modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
    .WithMany(c => c.Students);
```

☐ EF Core จะสร้างตารางกลาง StudentCourse ให้อัตโนมัติ

---

3. Stored Procedures

EF Core สามารถใช้ **Stored Procedure** ได้ทั้งสำหรับ **อ่านข้อมูล (FromSqlRaw)** และ **เขียนข้อมูล (ExecuteSqlRaw)**

☐ **อ่านข้อมูลด้วย Stored Procedure**

```
var blogs = context.Blogs
    .FromSqlRaw("EXEC GetAllBlogs")
```

```
    .ToList();
```

☐ **เขียนข้อมูลด้วย Stored Procedure**

```
context.Database.ExecuteSqlRaw(
    "EXEC InsertBlog @p0, @p1",
    parameters: new[] { "Tech Blog", "ASP.NET Core content" });
```

☐ **Mapping กับ SaveChanges() (EF Core 7+)**

สามารถ map Insert/Update/Delete ให้เรียก SP แทน SQL ปกติได้

```
modelBuilder.Entity<Blog>()
    .InsertUsingStoredProcedure("InsertBlog")
    .UpdateUsingStoredProcedure("UpdateBlog")
    .DeleteUsingStoredProcedure("DeleteBlog");
```

---

4. Query Optimization

การปรับปรุงประสิทธิภาพ query ใน EF Core สำคัญมาก โดยมีเทคนิคดังนี้:

☐ **ใช้ AsNoTracking()**

- เหมาะเมื่อ query ข้อมูลมาเพื่อ **อ่านอย่างเดียว**
- ลด overhead จาก Change Tracker

```
var blogs = context.Blogs.AsNoTracking().ToList();
```

☐ **Split Queries**

เมื่อ eager loading collection ใหญ่ ๆ ควรใช้ .AsSplitQuery() เพื่อลด Cartesian Explosion

```
var blogs = context.Blogs
                .Include(b => b.Posts)
                .AsSplitQuery()
                .ToList();
```

☐ **Filtered Include**

โหลดเฉพาะข้อมูลที่ต้องการ

```
var blogs = context.Blogs
                .Include(b => b.Posts.Where(p => p.Title.Contains("EF Core")))
                .ToList();
```

☐ **Compiled Queries**

สำหรับ query ที่ถูกเรียกซ้ำบ่อย ๆ สามารถ compile ล่วงหน้าได้

```
static readonly Func<MyDbContext, string, Blog?> GetBlogByName =
    EF.CompileQuery((MyDbContext ctx, string name) =>
        ctx.Blogs.FirstOrDefault(b => b.Name == name));

var blog = GetBlogByName(context, "Tech Blog");
```

☐ **Projection (Select) แทน Include**

ถ้าไม่ต้องการ entity เต็ม ๆ ให้เลือกเฉพาะ field ที่ใช้

```
var blogDtos = context.Blogs
    .Select(b => new BlogDto
    {
        Id = b.Id,
        Name = b.Name,
        PostCount = b.Posts.Count
    })
    .ToList();
```

☐ **สรุป**

1. **Eager vs Lazy Loading** ⟶ เลือกตาม use case, Eager ลดจำนวน query, Lazy เหมาะกับโหลดเฉพาะที่จำเป็น
2. **Relationships** ⟶ EF Core รองรับ One-to-Many และ Many-to-Many โดยไม่ต้องสร้าง join table เอง (ตั้งแต่ EF Core 5+)
3. **Stored Procedures** ⟶ ใช้ทั้งอ่าน/เขียน หรือ map ให้ EF Core ใช้งานแทน SQL ปกติได้
4. **Query Optimization** ⟶ ใช้ AsNoTracking(), SplitQuery, Filtered Include, CompiledQuery, Projection เพื่อให้ query เร็วและเบา

Advanced EF Core (เชิงลึก)

---

1. Eager vs Lazy Loading (Loading Strategies)

ใน EF Core มี **3 กลยุทธ์หลัก**: Lazy, Eager, Explicit

แต่การเลือกใช้งาน **ส่งผลโดยตรงต่อ performance และ behavior ของระบบ**

☐ **Lazy Loading (Deferred Execution)**

กลไกภายใน

- EF Core ใช้ *dynamic proxy* (ผ่าน Microsoft.EntityFrameworkCore.Proxies)
- เมื่อ navigation property ถูกเข้าถึง ⟶ proxy จะ trigger SQL ใหม่ **ต่อ object เดียว**

ข้อดี

- Query แรกเล็ก ⟶ memory footprint ต่ำ
- เหมาะกับ use case ที่ไม่รู้ล่วงหน้าว่าจะใช้ข้อมูลสัมพันธ์หรือไม่

ข้อเสีย

- เสี่ยงเจอ **N+1 problem** ⟶ loop 100 items = 101 queries
- ยากต่อการ trace performance เพราะ query ถูก execute ใน runtime

ตัวอย่างปัญหา

```
var blogs = context.Blogs.ToList();
foreach (var blog in blogs)
{
    Console.WriteLine(blog.Posts.Count); // Query ซ้ำ ๆ ⟶ N+1 Problem
}
```

☐ **Eager Loading (Data Pre-fetching)**

กลไกภายใน

- EF Core แปลง .Include() เป็น SQL JOIN หรือ **SplitQuery**
- โหลดข้อมูลที่ต้องใช้ *ล่วงหน้า*

ข้อดี

- เหมาะกับ UI แบบ "list with details" ⟶ ลด roundtrip
- ป้องกัน N+1 Problem

ข้อเสีย

- JOIN ใหญ่ ⟶ Cartesian Explosion ⟶ memory สูง
- ถ้ามีหลาย collection ⟶ query อาจไม่ efficient

ตัวอย่าง

```
var blogs = context.Blogs
    .Include(b => b.Posts)
    .Include(b => b.Author)
    .ToList();
```

☐ **Explicit Loading (On-demand)**

กลไกภายใน

- ใช้ context.Entry(entity).Collection(...).Load()
- คล้าย lazy loading แต่ควบคุมได้ตรงจุด

ข้อดี

- ให้ control มากกว่า lazy
- เหมาะกับการโหลดแบบ *conditional*

ตัวอย่าง

```
var blog = context.Blogs.First();
if (needPosts)
{
    context.Entry(blog).Collection(b => b.Posts).Load();
}
```

☐ **Best Practices**

- ใช้ **Eager** ⟶ ถ้า UI ต้องใช้ข้อมูลสัมพันธ์แน่นอน
- ใช้ **Lazy** ⟶ ถ้าต้องการลด query แรกและไม่แน่ใจว่าจะใช้ข้อมูลเสริม
- ใช้ **Explicit** ⟶ ถ้า logic กำหนดชัดว่าบางกรณีเท่านั้นถึงต้องโหลด

---

2. Relationships (One-to-Many, Many-to-Many)

☐ **One-to-Many**

กลไก

- EF Core สร้าง Foreign Key ใน DB + Navigation Properties
- สามารถกำหนด Behavior: Cascade Delete, Restrict, SetNull

```
modelBuilder.Entity<Post>()
    .HasOne(p => p.Blog)
    .WithMany(b => b.Posts)
    .OnDelete(DeleteBehavior.Cascade);
```

☐ Cascade Delete สำคัญ: ถ้า delete Blog ⟶ Post ถูก delete อัตโนมัติ

☐ **Many-to-Many (EF Core 5+)**

กลไก

- EF Core auto-create join table (shadow entity)
- สามารถกำหนดชื่อ table เอง

```
modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
    .WithMany(c => c.Students)
    .UsingEntity(j => j.ToTable("Enrollments"));
```

☐ ถ้า relationship มี attribute เพิ่ม (เช่น Grade, EnrollmentDate) ⟶ ต้องสร้าง **explicit join entity**

```
public class Enrollment
{
    public int StudentId { get; set; }
    public Student Student { get; set; }

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

    public DateTime EnrolledOn { get; set; }
}
```

☐ **Best Practices**

- One-to-Many ⟶ ใช้ ICollection<> หรือ List<> สำหรับ navigation
- Many-to-Many ⟶ ใช้ implicit join ถ้าเป็น pure relation, explicit join ถ้ามี attribute เพิ่ม

---

3. Stored Procedures

แม้ EF Core จะออกแบบมาเพื่อ LINQ + LINQ-to-SQL แต่ยังรองรับ **Stored Procedure (SP)**

☐ **ใช้งาน SP สำหรับ Query**

```
var blogs = context.Blogs
    .FromSqlRaw("EXEC GetBlogsByAuthor @p0", "Alice")
    .ToList();
```

☐ กลไก: EF Core แปลงผลลัพธ์ SP ⟶ Entity Tracking (เหมือน query ปกติ)

☐ **ใช้งาน SP สำหรับ Command**

```
context.Database.ExecuteSqlRaw(
    "EXEC InsertBlog @p0, @p1",
    "My Blog", "EF Core Content");
```

☐ **Mapping SaveChanges() ⟶ Stored Procedure (EF Core 7+)**

```
modelBuilder.Entity<Blog>()
    .InsertUsingStoredProcedure("sp_InsertBlog")
    .UpdateUsingStoredProcedure("sp_UpdateBlog")
    .DeleteUsingStoredProcedure("sp_DeleteBlog");
```

☐ ใช้ได้ถ้าต้องการ enforce logic ใน DB layer

☐ ระวัง: SP ทำให้ lose portability (DB-specific)

☐ **Best Practices**

- ใช้ SP ⟶ ถ้ามี **business logic ใน DB** หรือ **performance-critical query**
- ใช้ LINQ ⟶ ถ้าต้องการ maintainability และ portability

---

4. Query Optimization

EF Core มี abstraction แต่ก็มี cost ถ้าไม่ optimize

☐ **AsNoTracking()**

- ปิด Change Tracker ⟶ ลด memory และ CPU
- เหมาะกับ Read-Only queries

```
var blogs = context.Blogs.AsNoTracking().ToList();
```

☐ **Split vs Single Query**

EF Core Default (Single Query): JOIN ทุกอย่างใน query เดียว

ปัญหา: Cartesian Explosion

```
var blogs = context.Blogs
    .Include(b => b.Posts)
    .AsSplitQuery() // แยก query เพื่อลด explosion
    .ToList();
```

☐ **Projection (Select DTO)**

โหลดเฉพาะ field ที่จำเป็น แทนที่จะโหลดทั้ง entity

```
var blogDtos = context.Blogs
    .Select(b => new BlogDto
    {
        Id = b.Id,
```

```
        Name = b.Name,
        PostCount = b.Posts.Count
    })
    .ToList();
```

☐ **Compiled Queries**

EF Core ปกติจะ compile query ทุกครั้ง

ถ้า query ถูกเรียกซ้ำ ⟶ ควรใช้ EF.CompileQuery

```
private static readonly Func<MyDbContext, string, Blog?> _getBlogByName =
    EF.CompileQuery((MyDbContext ctx, string name) =>
        ctx.Blogs.FirstOrDefault(b => b.Name == name));

var blog = _getBlogByName(context, "Tech Blog");
```

☐ **Indexing & Query Hints**

- ใช้ Fluent API เพิ่ม Index:

```
modelBuilder.Entity<Blog>()
    .HasIndex(b => b.Name);
```

- ใช้ Raw SQL + Hint (ถ้า DB engine support)

☐ **Best Practices**

1. ใช้ AsNoTracking() สำหรับ read-only
2. ใช้ .AsSplitQuery() สำหรับ multiple collection include
3. ใช้ DTO Projection เพื่อลด memory
4. ใช้ CompiledQuery สำหรับ hot-path queries
5. Monitor ด้วย **SQL Profiler / EF Core Logging**

☐ สรุปเชิงกลยุทธ์

- **Loading**:
    - Lazy ⟶ flexibility, but N+1 risk
    - Eager ⟶ predictable, but heavy join
    - Explicit ⟶ balance, full control
- **Relationships**:
    - One-to-Many = straightforward

- Many-to-Many = implicit (simple) vs explicit (with attributes)

- **Stored Procedures**:
  - ใช้เมื่อมี **legacy DB**, **complex logic**, หรือ **performance-critical**
- **Optimization**:
  - Monitor queries
  - เลือก loading strategy ให้เหมาะ
  - ใช้ DTO, Index, CompiledQuery
  - ตรวจสอบ execution plan

Eager vs Lazy Loading ใน Entity Framework Core (EF Core)

Eager vs Lazy Loading ใน Entity Framework Core (EF Core) แบบละเอียดที่สุด ตั้งแต่ **กลไกภายใน, ตัวอย่างโค้ด, การใช้งานจริง, ปัญหา, Best Practices**

□ Eager vs Lazy Loading (EF Core)

1. ความหมายและแนวคิด

ใน EF Core เวลาที่เรามี **ความสัมพันธ์ (Relationship)** เช่น Blog มี Posts เราจะต้องตัดสินใจว่าจะโหลด Posts มาพร้อมกันหรือไม่

- **Eager Loading** ⟶ โหลดข้อมูลความสัมพันธ์ *ล่วงหน้า* ใน query แรก
- **Lazy Loading** ⟶ โหลดข้อมูลความสัมพันธ์ *เมื่อถูกเข้าถึงจริง*

---

2. กลไกของแต่ละแบบ

□ **Eager Loading**

- ใช้ Include() และ ThenInclude()
- EF Core จะสร้าง **SQL JOIN (หรือหลาย query ถ้าใช้ AsSplitQuery)**
- Navigation properties ถูกเติมข้อมูลตั้งแต่แรก

ตัวอย่าง

```
var blogs = context.Blogs
    .Include(b => b.Posts)  // โหลด Posts มาพร้อมกับ Blogs
    .ToList();

foreach (var blog in blogs)
{
    Console.WriteLine($"{blog.Name} has {blog.Posts.Count} posts");
```

```
}
```

☐ ผลลัพธ์ SQL (สมมุติ)

```
SELECT b.Id, b.Name, p.Id, p.Title, p.BlogId
FROM Blogs b
LEFT JOIN Posts p ON b.Id = p.BlogId;
```

☐ **Lazy Loading**

- ต้องติดตั้ง NuGet: Microsoft.EntityFrameworkCore.Proxies
- เปิดใช้งานใน DbContext

```
optionsBuilder.UseLazyLoadingProxies()
              .UseSqlServer(connectionString);
```

- ต้องกำหนด navigation property เป็น virtual

ตัวอย่าง

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

    public virtual ICollection<Post> Posts { get; set; } = new List<Post>();
}
```

การใช้งาน

```
var blog = context.Blogs.First();
Console.WriteLine(blog.Posts.Count); // EF Core จะยิง query ตอนนี้
```

☐ ผลลัพธ์ SQL สองรอบ

```
-- Query แรก
SELECT TOP(1) b.Id, b.Name FROM Blogs b;

-- Query ตอนเข้าถึง Posts
SELECT p.Id, p.Title, p.BlogId FROM Posts p WHERE p.BlogId = 1;
```

---

3. เปรียบเทียบ

ประเด็น	Eager Loading	Lazy Loading
การโหลดข้อมูล	โหลดข้อมูลสัมพันธ์ทันที	โหลดเมื่อ navigation ถูกเข้าถึง

ประเด็น	Eager Loading	Lazy Loading
จำนวน Query	มัก query เดียว (JOIN)	หลาย query (อาจเกิด N+1)
Performance	ดีสำหรับ data ชุดเล็ก, UI ที่ต้องใช้ทั้งหมด	ดีถ้าข้อมูลสัมพันธ์ไม่ถูกใช้งานบ่อย
Memory	ใช้มากขึ้น (โหลดข้อมูลเยอะ)	ใช้น้อยในตอนแรก
ควบคุม	ชัดเจน, predictable	ยืดหยุ่นแต่ควบคุมยาก
เหมาะกับ	API/หน้า UI ที่ต้องการข้อมูลครบ	ระบบที่ข้อมูลสัมพันธ์ใช้บางกรณี

---

4. ปัญหาที่เจอบ่อย

☐ **Eager Loading**

- **Cartesian Explosion**:

  ถ้ามี Blog 100 และ Post 1000 ⟶ JOIN อาจ return row = 1000
- **Over-fetching**: โหลดข้อมูลเยอะเกินที่ต้องใช้

☐ **Lazy Loading**

- **N+1 Problem**:
- var blogs = context.Blogs.ToList();
- foreach (var blog in blogs)
- {
- Console.WriteLine(blog.Posts.Count); // Query แยกต่อ Blog ⟶ N+1
- }

  ถ้า Blog = 100 ⟶ 101 queries

---

5. Best Practices

☐ ใช้ **Eager Loading**

- เมื่อ UI/API ต้องการข้อมูลทั้งหมดอยู่แล้ว เช่น **List + Detail**
- ใช้ .AsSplitQuery() เมื่อ Include หลาย collection

☐ ใช้ **Lazy Loading**

- เมื่อ navigation ไม่ได้ถูกใช้บ่อย
- เมื่อไม่แน่ใจว่า user ต้องการข้อมูลนั้น

☐ ใช้ **Explicit Loading**

- ควบคุมโหลดเฉพาะบางกรณี (สมดุลระหว่าง Eager/Lazy)

---

6. ตัวอย่างรวม (Eager vs Lazy)

```
// Eager Loading
var eagerBlogs = context.Blogs
    .Include(b => b.Posts)
    .ToList();
Console.WriteLine($"Eager loaded {eagerBlogs.Count} blogs with posts");

// Lazy Loading
var lazyBlog = context.Blogs.First();
Console.WriteLine($"Lazy blog: {lazyBlog.Name}, Posts = {lazyBlog.Posts.Count}");
```

---

7. สรุป

- **Eager Loading** = โหลดข้อมูลสัมพันธ์ทันที ⟶ ดีต่อ performance ถ้ารู้ว่าใช้แน่นอน
- **Lazy Loading** = โหลดเมื่อถูกเรียกใช้ ⟶ ยืดหยุ่น แต่เสี่ยงเจอ N+1
- ต้องเลือกตาม **Use Case + ปริมาณข้อมูล + Query Pattern**

ตัวอย่าง **Eager vs Lazy Loading** บน **EF Core (ASP.NET Core Web API)** แบบ **เต็มไฟล์ + โครงสร้าง + คำอธิบายโค้ด + ผลการรัน**

ผมจะจัดให้เป็น 2 ชุด:

- **พื้นฐาน 3 โปรแกรม** ⟶ เข้าใจความแตกต่างระหว่าง Eager, Lazy, Explicit
- **แนวประยุกต์ 3 โปรแกรม** ⟶ ใช้ใน API จริง, กรองข้อมูล, Optimize Query

---

☐ **ชุดที่ 1: พื้นฐาน (Basic Examples)**

☐ **โปรแกรมที่ 1: Eager Loading**

☐ **โครงสร้างโปรเจกต์**

```
EagerLoadingDemo/
├── Program.cs
├── Models/
│   ├── Blog.cs
│   └── Post.cs
├── Data/
│   └── AppDbContext.cs
└── Controllers/
```

└── BlogsController.cs

☐ **Models/Blog.cs**

```
namespace EagerLoadingDemo.Models;

public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public List<Post> Posts { get; set; } = new();
}
```

☐ **Models/Post.cs**

```
namespace EagerLoadingDemo.Models;

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; } = "";
    public int BlogId { get; set; }
    public Blog Blog { get; set; } = null!;
}
```

☐ **Data/AppDbContext.cs**

```
using Microsoft.EntityFrameworkCore;
using EagerLoadingDemo.Models;

namespace EagerLoadingDemo.Data;

public class AppDbContext : DbContext
{
    public DbSet<Blog> Blogs => Set<Blog>();
    public DbSet<Post> Posts => Set<Post>();

    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
}
```

☐ **Controllers/BlogsController.cs**

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

namespace EagerLoadingDemo.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BlogsController : ControllerBase
{
    private readonly AppDbContext _context;
    public BlogsController(AppDbContext context) => _context = context;

    [HttpGet("eager")]
    public IActionResult GetBlogsEager()
    {
        var blogs = _context.Blogs
            .Include(b => b.Posts) // Eager Loading
            .ToList();
        return Ok(blogs);
    }
}
```

☐ **Program.cs**

```
using EagerLoadingDemo.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<AppDbContext>(opt =>
    opt.UseInMemoryDatabase("EagerDemoDB"));
builder.Services.AddControllers();
var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
```

```
    var db = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    db.Blogs.Add(new Models.Blog
    {
        Name = "Tech Blog",
        Posts = new List<Models.Post>
        {
            new() { Title = "EF Core Basics" },
            new() { Title = "Eager vs Lazy" }
        }
    });
    db.SaveChanges();
}

app.MapControllers();
app.Run();
```

▶□ **ผลการรัน**

เรียก API ⟶ GET /api/blogs/eager

```
[
  {
    "id": 1,
    "name": "Tech Blog",
    "posts": [
      { "id": 1, "title": "EF Core Basics", "blogId": 1 },
      { "id": 2, "title": "Eager vs Lazy", "blogId": 1 }
    ]
  }
]
```

□ **โหลด Blog + Posts ใน query เดียว**

□ **โปรแกรมที่ 2: Lazy Loading**

□ **โครงสร้าง**

```
LazyLoadingDemo/
  ├── Program.cs
  ├── Models/
```

```
│   ├── Blog.cs
│   └── Post.cs
├── Data/AppDbContext.cs
└── Controllers/BlogsController.cs
```

☐ **Models/Blog.cs**

```
namespace LazyLoadingDemo.Models;

public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; } = "";

    public virtual ICollection<Post> Posts { get; set; } = new List<Post>(); // virtual
}
```

☐ **Models/Post.cs**

```
namespace LazyLoadingDemo.Models;

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; } = "";
    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; } = null!;
}
```

☐ **Data/AppDbContext.cs**

```
using Microsoft.EntityFrameworkCore;
using LazyLoadingDemo.Models;

namespace LazyLoadingDemo.Data;

public class AppDbContext : DbContext
{
    public DbSet<Blog> Blogs => Set<Blog>();
    public DbSet<Post> Posts => Set<Post>();
```

```
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseLazyLoadingProxies();
    }
}
```

☐ **Controllers/BlogsController.cs**

```
using Microsoft.AspNetCore.Mvc;
using LazyLoadingDemo.Data;

namespace LazyLoadingDemo.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BlogsController : ControllerBase
{
    private readonly AppDbContext _context;
    public BlogsController(AppDbContext context) => _context = context;

    [HttpGet("lazy")]
    public IActionResult GetBlogsLazy()
    {
        var blogs = _context.Blogs.ToList(); // ยังไม่โหลด Posts
        var result = blogs.Select(b => new
        {
            b.Id,
            b.Name,
            PostCount = b.Posts.Count // Query ตอนนี้
        });
        return Ok(result);
    }
}
```

Program.cs

(เหมือนโปรแกรมที่ 1 แต่เปลี่ยน namespace)

▶ ผลการรัน

เรียก API ⟶ GET /api/blogs/lazy

```
[
  {
    "id": 1,
    "name": "Tech Blog",
    "postCount": 2
  }
]
```

Posts ถูกโหลดเฉพาะตอนที่เข้าถึง b.Posts

โปรแกรมที่ 3: Explicit Loading

Controllers/BlogsController.cs

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

namespace ExplicitLoadingDemo.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BlogsController : ControllerBase
{
    private readonly AppDbContext _context;
    public BlogsController(AppDbContext context) => _context = context;

    [HttpGet("explicit")]
    public IActionResult GetBlogExplicit()
    {
        var blog = _context.Blogs.First();

        // Explicit Loading
```

```
        _context.Entry(blog)
                .Collection(b => b.Posts)
                .Load();

        return Ok(new
        {
            blog.Id,
            blog.Name,
            PostCount = blog.Posts.Count
        });
    }
}
```

▶□ **ผลการรัน**

เรียก API ⟶ GET /api/blogs/explicit

```
{
  "id": 1,
  "name": "Tech Blog",
  "postCount": 2
}
```

□ **โหลด Posts แบบควบคุมด้วยตนเอง**

□ **ชุดที่ 2: แนวประยุกต์ (Applied Examples)**

□ **โปรแกรมที่ 4: API เปรียบเทียบ Eager vs Lazy**

- Endpoint /api/blogs/eager ⟶ ใช้ Include
- Endpoint /api/blogs/lazy ⟶ ใช้ lazy proxies

ผลการรัน

- Eager: ยิง query เดียว ⟶ ได้ Blog + Posts พร้อม
- Lazy: Blog query 1 ครั้ง + Posts query ตามจำนวน Blog

□ **โปรแกรมที่ 5: Filtered Eager Loading**

```
var blogs = _context.Blogs
    .Include(b => b.Posts.Where(p => p.Title.Contains("EF")))
    .ToList();
```

ผลการรัน

```
[
  {
    "id": 1,
    "name": "Tech Blog",
    "posts": [
      { "id": 1, "title": "EF Core Basics", "blogId": 1 }
    ]
  }
]
```

□ โหลด Posts เฉพาะที่เกี่ยวข้อง ⟶ Query เบาลง

□ **โปรแกรมที่ 6: Performance Optimization (Split Query)**

```
var blogs = _context.Blogs
    .Include(b => b.Posts)
    .AsSplitQuery() // ป้องกัน Cartesian Explosion
    .ToList();
```

ผลการรัน

- EF Core ยิง 2 query:
    1. Blog
    2. Posts by BlogId
- ลดปัญหา duplicate row เวลา join ข้อมูลจำนวนมาก

□ **สรุป**

- **พื้นฐาน 3 โปรแกรม** ⟶ แสดงความแตกต่างระหว่าง Eager, Lazy, Explicit
- **แนวประยุกต์ 3 โปรแกรม** ⟶ API เปรียบเทียบ, Filtered Include, Split Query

---

Relationships (One-to-Many, Many-to-Many)

Relationships (One-to-Many, Many-to-Many) ใน **Entity Framework Core (EF Core)** ให้ครบถ้วนที่สุด โดยแบ่งเป็น

1. □ แนวคิดพื้นฐาน (Concepts)
2. □ การ Mapping ความสัมพันธ์
3. □ ตัวอย่างโค้ดเต็ม (One-to-Many, Many-to-Many)

4. □ ปัญหาที่พบบ่อย (Pitfalls)
5. □ Best Practices

---

1. □ แนวคิดพื้นฐาน

Relationship ใน EF Core คือการกำหนดความสัมพันธ์ระหว่าง Entity (ตาราง) เช่น Blog กับ Post

- **One-to-Many**:
  หนึ่ง Blog มีหลาย Post
  (Blog ⟶ Parent, Post ⟶ Child)
  เช่น Blog.Id ⟶ Post.BlogId
- **Many-to-Many**:
  หลาย Student เรียนหลาย Course
  (ใช้ **Join Table** เช่น StudentCourses)

---

2. □ การ Mapping ความสัมพันธ์

One-to-Many

- Parent มี **Collection Navigation Property**
- Child มี **Reference Navigation Property** + **Foreign Key**

```
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public List<Post> Posts { get; set; } = new(); // One-to-Many
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; } = "";
    public int BlogId { get; set; } // Foreign Key
    public Blog Blog { get; set; } = null!;
}
```

Many-to-Many

ตั้งแต่ **EF Core 5+** ⟶ รองรับ Many-to-Many แบบ **Implicit Join Table** โดยอัตโนมัติ

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public List<Course> Courses { get; set; } = new();
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; } = "";
    public List<Student> Students { get; set; } = new();
}
```

EF Core จะสร้างตารางกลาง CourseStudent ให้อัตโนมัติ

ถ้าต้องการ **Custom Join Entity** (เช่นมี Field เพิ่ม เช่น วันที่ลงทะเบียน) ⟶ ต้องประกาศ Entity เอง

```
public class Enrollment
{
    public int StudentId { get; set; }
    public Student Student { get; set; } = null!;
    public int CourseId { get; set; }
    public Course Course { get; set; } = null!;
    public DateTime EnrolledDate { get; set; }
}
```

---

3. □ ตัวอย่างโค้ดเต็ม

□ โปรแกรมตัวอย่างที่ 1: One-to-Many

```
using Microsoft.EntityFrameworkCore;

public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public List<Post> Posts { get; set; } = new();
}
```

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; } = "";
    public int BlogId { get; set; }
    public Blog Blog { get; set; } = null!;
}

public class AppDbContext : DbContext
{
    public DbSet<Blog> Blogs => Set<Blog>();
    public DbSet<Post> Posts => Set<Post>();

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseInMemoryDatabase("DemoDB");
}

class Program
{
    static void Main()
    {
        using var db = new AppDbContext();

        var blog = new Blog { Name = "Tech Blog" };
        blog.Posts.Add(new Post { Title = "EF Core Basics" });
        blog.Posts.Add(new Post { Title = "Relationships" });
        db.Blogs.Add(blog);
        db.SaveChanges();

        var blogs = db.Blogs.Include(b => b.Posts).ToList();
        foreach (var b in blogs)
            Console.WriteLine($"{b.Name} has {b.Posts.Count} posts");
    }
```

```
}
```

▶ ผลลัพธ์

```
Tech Blog has 2 posts
```

โปรแกรมตัวอย่างที่ 2: Many-to-Many (Implicit Join Table)

```
using Microsoft.EntityFrameworkCore;

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public List<Course> Courses { get; set; } = new();
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; } = "";
    public List<Student> Students { get; set; } = new();
}

public class SchoolContext : DbContext
{
    public DbSet<Student> Students => Set<Student>();
    public DbSet<Course> Courses => Set<Course>();

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseInMemoryDatabase("SchoolDB");
}

class Program
{
    static void Main()
    {
```

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

        var student = new Student { Name = "Alice" };
        var course1 = new Course { Title = "Math" };
        var course2 = new Course { Title = "Science" };

        student.Courses.Add(course1);
        student.Courses.Add(course2);

        db.Students.Add(student);
        db.SaveChanges();

        var students = db.Students.Include(s => s.Courses).ToList();
        foreach (var s in students)
            Console.WriteLine($"{s.Name} enrolled: {string.Join(", ", s.Courses.Select(c =>
c.Title))}");
    }
}
```

▶□ **ผลลัพธ์**

```
Alice enrolled: Math, Science
```

□ **โปรแกรมตัวอย่างที่ 3: Many-to-Many (Custom Join Entity)**

```
using Microsoft.EntityFrameworkCore;

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public List<Enrollment> Enrollments { get; set; } = new();
}

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

```
    public string Title { get; set; } = "";
    public List<Enrollment> Enrollments { get; set; } = new();
}

public class Enrollment
{
    public int StudentId { get; set; }
    public Student Student { get; set; } = null!;
    public int CourseId { get; set; }
    public Course Course { get; set; } = null!;
    public DateTime EnrolledDate { get; set; }
}

public class SchoolContext : DbContext
{
    public DbSet<Student> Students => Set<Student>();
    public DbSet<Course> Courses => Set<Course>();
    public DbSet<Enrollment> Enrollments => Set<Enrollment>();

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseInMemoryDatabase("SchoolDB");
}

class Program
{
    static void Main()
    {
        using var db = new SchoolContext();

        var student = new Student { Name = "Bob" };
        var course = new Course { Title = "Physics" };

        db.Enrollments.Add(new Enrollment
        {
```

```
            Student = student,
            Course = course,
            EnrolledDate = DateTime.Now
        });

        db.SaveChanges();

        var enrollments = db.Enrollments
            .Include(e => e.Student)
            .Include(e => e.Course)
            .ToList();

        foreach (var e in enrollments)
            Console.WriteLine($"{e.Student.Name} enrolled {e.Course.Title} on {e.EnrolledDate}");
    }
}
```

▶ **ผลลัพธ์**

```
Bob enrolled Physics on 8/20/2025 10:15:32 AM
```

---

4. ปัญหาที่พบบ่อย

- **One-to-Many** ⟶ ลืมใส่ Foreign Key ⟶ EF Core จะสร้าง shadow property เอง
- **Many-to-Many** ⟶ ถ้าใช้ Custom Join Entity ⟶ ต้อง map ด้วย modelBuilder.Entity<Enrollment>().HasKey(...)
- การโหลดข้อมูลมาก ๆ โดยใช้ Include ⟶ อาจเจอ **Cartesian Explosion**

---

5. Best Practices

 กำหนด Navigation Property ทั้งสองฝั่งเสมอ

 ใช้ Include() / ThenInclude() สำหรับโหลดข้อมูลสัมพันธ์

 ถ้าเป็น Many-to-Many แบบมีข้อมูลเสริม (เช่น วันที่, เกรด) ⟶ ใช้ Custom Join Entity

 ใช้ .AsSplitQuery() ป้องกัน query บวมเวลา Include หลาย collection

---

ตัวอย่าง **Relationships (One-to-Many, Many-to-Many)** แบบ **เต็มไฟล์ + โครงสร้าง + คำอธิบายโค้ด + ผลการรัน**

ผมจะแบ่งเป็น 2 ชุด: