# **PHIẾU HỌC TẬP ENTITY FRAMEWORK CORE**

**Họ và tên:** Đoàn Quốc Bảo

**Mã sinh viên:** 1871020071

**Ngày làm:** 21/01/2026

## **PHIẾU HỌC TẬP EF CORE [01] — ORM LÀ GÌ & CODE FIRST WORKFLOW**

**Video/Tài liệu đã xem (Link): https://www.youtube.com/watch?v=0e27MjAY0B4**

### **1. Kiến thức cốt lõi (Tóm tắt bằng lời của bạn)**

*Sau khi xem video/đọc tài liệu, hãy tự diễn giải 3 khái niệm quan trọng nhất:*

#### **1. ORM là gì? (Viết bằng lời riêng của bạn)**

ORM là viết tắt của Object-Relational Mapping

Nó giải quyết vấn đề gì? (Tại sao cần ORM thay vì viết SQL thuần?)

Trước đây, khi dùng SQL thuần, bạn phải viết những đoạn code dài dằng dặc, nối chuỗi loằng ngoằng để lấy dữ liệu. ORM ra đời để giải quyết các "nỗi đau" sau:

* Giảm bớt code lặp lại (Boilerplate code): Thay vì phải mở kết nối, viết câu lệnh SELECT \* FROM..., rồi đọc từng dòng dữ liệu gán vào đối tượng, ORM làm giúp bạn chỉ với 1 dòng code.
* Lập trình hướng đối tượng (OOP) đồng nhất: Bạn thao tác với dữ liệu hoàn toàn bằng ngôn ngữ lập trình (C#, Java, Python...) thay vì phải "nhảy" qua lại giữa code ứng dụng và cú pháp SQL.
* An toàn và bảo mật: ORM thường mặc định chống lại các lỗi bảo mật nguy hiểm như SQL Injection.
* Dễ bảo trì và đổi DB: Hôm nay bạn dùng SQL Server, mai sếp muốn đổi sang PostgreSQL? Nếu dùng SQL thuần, bạn phải sửa lại hàng nghìn câu lệnh. Với ORM, bạn chỉ cần thay đổi cấu hình (Config) là xong.

#### **2. Entity Framework Core (EF Core) là gì?**

EF Core là một Object-Relational Mapper (ORM) mã nguồn mở, nhẹ và đa nền tảng trong .NET để làm việc với Cơ sở dữ liệu (Database)

Ưu điểm chính của nó so với ADO.NET thuần:

* Tốc độ phát triển (Productivity): Bạn viết code cực nhanh. Thay vì mất 30 phút viết code ADO.NET để lấy dữ liệu từ 3 bảng liên kết, EF Core chỉ mất vài dòng.
* LINQ (Language Integrated Query): Đây là "vũ khí" tối thượng. Bạn có thể truy vấn dữ liệu trực tiếp bằng cú pháp C# (LINQ), giúp kiểm tra lỗi ngay khi đang viết (Compile-time check), tránh việc viết sai tên cột như trong SQL thuần.
* Quản lý Migration: EF Core giúp bạn quản lý việc thay đổi cấu trúc database (thêm cột, sửa bảng) đồng bộ với code C#. Bạn không cần phải vào SQL Management Studio để tạo bảng bằng tay nữa.

#### **3. Code First Workflow là gì? (Vẽ sơ đồ workflow)**

**Quy trình 5 bước:**

1. Viết các lớp POCO (Plain Old CLR Objects) (Product, Category)

↓

2. Tạo lớp DbContext

↓

3. Add Migration (Tạo bản ghi lịch sử thay đổi)

↓

4. Update Database

↓

5. Database và các Bảng được tạo tự động!

### **2. Ví dụ thực hành (Bắt buộc)**

*Yêu cầu: Tạo một project ASP.NET Core Web API đơn giản với EF Core*

#### **Mục tiêu:**

* Tạo 2 Entity: Category và Product
* Tạo DbContext
* Tạo migration đầu tiên
* Apply migration để tạo Database

#### **Bước 1: Tạo Entity Classes**

Viết code cho **Models/Category.cs**:

public class Category

{

public int Id { get; set; }

// VIẾT CÁC PROPERTIES KHÁC CỦA BẠN VÀO ĐÂY

// (Gợi ý: Name, Description)

// Navigation Property (1 Category → Many Products)

public ICollection<Product> Products { get; set; } = new List<Product>();

}

Viết code cho **Models/Product.cs**:

public class Product

{

public int Id { get; set; }

// VIẾT CÁC PROPERTIES KHÁC CỦA BẠN VÀO ĐÂY

// (Gợi ý: Name, Price, Stock, CategoryId)

// Foreign Key

public int CategoryId { get; set; }

// Navigation Property

public Category Category { get; set; } = null!;

}

**Câu hỏi:** Tại sao Property Category trong class Product lại có = null!;? Nó có nghĩa là gì?

Nó là sự kết hợp của hai tính năng: Nullable Reference Types và Null-Forgiving Operator.

* Nếu bạn khai báo một Property kiểu chuỗi hoặc một Class (như Category) mà không khởi tạo giá trị cho nó, trình biên dịch sẽ cảnh báo (Warning): "Biến này không được null, bạn phải gán giá trị cho nó trong hàm tạo (Constructor)".
* Tuy nhiên, với EF Core, các Property như Category (Navigation Property) thường sẽ được EF Core tự động đổ dữ liệu vào (Lazy loading hoặc Eager loading) khi bạn truy vấn từ Database. Bạn không muốn khởi tạo nó theo cách thủ công.

#### **Bước 2: Tạo DbContext**

Viết code cho **Data/AppDbContext.cs**:

using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)

{

}

// VIẾT CÁC DBSET CỦA BẠN VÀO ĐÂY

// (Gợi ý: DbSet<Category>, DbSet<Product>)

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

// Fluent API configuration (nếu cần)

}

}

#### **Bước 3: Đăng ký DbContext trong Program.cs**

Viết đoạn code để đăng ký DbContext:

// Thêm vào file Program.cs sau dòng: var builder = WebApplication.CreateBuilder(args);

// Bước 1: Đọc Connection String từ appsettings.json

var connectionString = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// Bước 2: Đăng ký DbContext với DI Container

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### **Bước 4: Cấu hình appsettings.json**

Thêm Connection String vào file **appsettings.json**:

{

"ConnectionStrings": {

"DefaultConnection": "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"

}

}

**Giải thích:**

* Server=. có nghĩa là: SQL Server trên máy local
* Database=EShopDb có nghĩa là: Tên database sẽ được tạo
* Trusted\_Connection=True có nghĩa là: Windows Authentication

#### **Bước 5: Tạo Migration & Apply vào Database**

Mở **Package Manager Console** và chạy lệnh:

Add-Migration InitialCreate

**Kết quả:** Một folder Migrations/ sẽ được tạo. Hãy mô tả file migration này là gì?

File migration này là lịch sử thay đổi Database dưới dạng code C#. Nó tự động được tạo bởi EF Core khi bạn chạy Add-Migration InitialCreate.

Một file Migration thường chứa một Class kế thừa từ Migration với hai phương thức quan trọng nhất:

1. Phương thức Up() (Lên đời)

Đây là nơi chứa các lệnh để thay đổi Database. Khi bạn chạy lệnh Update-Database, EF Core sẽ thực thi code trong hàm này.

Nội dung: Chứa các lệnh như CreateTable, AddColumn, CreateIndex.

Mục đích: Đưa Database từ trạng thái cũ lên trạng thái mới nhất theo Model bạn vừa viết.

2. Phương thức Down() (Xuống đời)

Đây là nơi chứa các lệnh để hoàn tác (Undo) những gì hàm Up đã làm.

Nội dung: Chứa các lệnh ngược lại như DropTable, DropColumn.

Mục đích: Giúp bạn quay lại phiên bản Database cũ nếu chẳng may bản Migration mới bị lỗi hoặc bạn đổi ý.

Tiếp theo, chạy lệnh:

Update-Database

**Kết quả:** Database EShopDb sẽ được tạo. Hãy mở **SQL Server Management Studio (SSMS)** và:

1. Kiểm tra Database EShopDb đã tồn tại
2. Chụp ảnh màn hình cấu trúc bảng Products (trong Object Explorer → Databases → EShopDb → Tables)

**Ảnh kết quả (Dán ảnh SSMS vào đây):**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

### **3. Câu hỏi/Thắc mắc của tôi**

**Ba điều tôi chưa hiểu rõ hoặc muốn hỏi thêm:**

1. Phân biệt IEnumerable và IQueryable trong EF Core?
2. Eager Loading, Lazy Loading và Explicit Loading là gì?
3. Tại sao không nên dùng Migration trên môi trường Production (Real Server)?

## **PHIẾU HỌC TẬP EF CORE [02] — CRUD & RELATIONSHIPS**

**Video/Tài liệu đã xem (Link):** https://www.youtube.com/watch?v=FNJpXWPka30&list=PLdHN14J7CHtaNHnAPOk\_yEq0GidtbTizl

### **1. Kiến thức cốt lõi (So sánh 4 thao tác CRUD)**

*Sau khi xem video/đọc tài liệu, hãy hoàn thành bảng so sánh dưới đây:*

| **CRUD** | **Ý nghĩa** | **Method EF Core** | **Ví dụ code (1-2 dòng)** |
| --- | --- | --- | --- |
| **C** | Thêm mới dữ liệu vào Database. | \_context.Products.Add() | *\_context.Add(new Product { Name = "Laptop" });*  await \_context.SaveChangesAsync(); |
| **R** | Đọc/Truy vấn dữ liệu từ Database. | \_context.Products.Find() hoặc FirstOrDefaultAsync() | *var p = await \_context.Products.FindAsync(id);*    *var list = await \_context.Products.ToListAsync();* |
| **U** | Cập nhật/Sửa dữ liệu đã tồn tại. | SaveChangesAsync() (sau khi modify property) | *product.Price = 1000;*    *await \_context.SaveChangesAsync();* |
| **D** | \_context.Products.Remove() | \_context.Products.Remove() | *\_context.Products.Remove(product);*    *await \_context.SaveChangesAsync();* |

### **2. Kiến thức về Relationships (Quan hệ giữa các bảng)**

#### **2.1 One-to-Many (1-N): Category → Products**

Quan hệ này có ý nghĩa gì?

1 Category chỉ có quan hệ với 1 product nhưng 1 product có thể quan hệ với nhiều categories

Trong code C#, chúng ta cấu hình bằng:

// Trong Category.cs

public ICollection<Product> Products { get; set; } = new List<Product>();

// Trong Product.cs

public int CategoryId { get; set; }

public Category Category { get; set; } = null!;

**Câu hỏi:**

* Tại sao Category cần ICollection<Product>?
* ICollection<Product> đóng vai trò là một Navigation Property (Thuộc tính điều hướng).
* Nó cho phép bạn đứng từ thực thể "Một" (Category) để truy cập danh sách các thực thể "Nhiều" (Products) thuộc về nó.
* Tại sao Product cần CategoryId?
  + Ở tầng Database, để bảng Products biết nó thuộc về dòng nào trong bảng Categories, nó cần một cột lưu trữ ID của dòng đó.
  + Nếu không có CategoryId, Database sẽ không biết sản phẩm "iPhone 15" thuộc về danh mục "Điện thoại" hay "Tủ lạnh".
  + Nó giúp đảm bảo toàn vẹn dữ liệu: Bạn không thể gán một sản phẩm vào một danh mục không tồn tại.

#### **2.2 N+1 Query Problem**

Đây là một **vấn đề hiệu suất** rất phổ biến. Hãy giải thích:

**❌ KHÔNG TỐT (Gây N+1 Problem):**

var products = await \_context.Products.ToListAsync();

foreach (var product in products)

{

var categoryName = product.Category.Name; // ← Mỗi lần này = 1 query!

}

// Nếu có 100 sản phẩm → 1 query lấy products + 100 queries lấy category = 101 queries total!

**Tại sao điều này xảy ra?**

Đây là một trong những "hố đen" về hiệu suất lớn nhất khi làm việc với ORM. Vấn đề này xảy ra do cơ chế Lazy Loading (Tải chậm) và cách EF Core tương tác với Database.

Dưới đây là lý do tại sao nó xảy ra và cách nó vận hành:

1. Bản chất của "Sự trì hoãn"

Khi bạn thực hiện dòng code đầu tiên: var products = await \_context.Products.ToListAsync(); EF Core chỉ chạy đúng một câu lệnh SELECT \* FROM Products. Lúc này, vì lý do tối ưu bộ nhớ, EF Core chưa hề lấy dữ liệu của bảng Categories. Các thuộc tính Category trong từng đối tượng product hiện đang ở trạng thái null hoặc là một "Proxy" (đối tượng giả lập) đang chờ được nạp.

2. Kích hoạt truy vấn ngoài ý muốn

Khi vào vòng lặp foreach, bạn chạm vào: product.Category.Name

Tại thời điểm này, EF Core nhận thấy: "À, lập trình viên đang cần dữ liệu của Category cho sản phẩm này, nhưng tôi chưa có nó trong bộ nhớ". Ngay lập tức, nó phải tạm dừng code của bạn lại để "phóng" một câu lệnh SQL xuống Database để tìm danh mục của đúng sản phẩm đó.

**✅ TỐT (Dùng Eager Loading):**

var products = await \_context.Products

.Include(p => p.Category) // ← Chỉ 1 query, load cả Category!

.ToListAsync();

**Tại sao cách này tốt hơn?**

1. Giảm thiểu số lượng "Round-trip" (Chuyến đi khứ hồi)

Vấn đề: Mỗi lần một câu lệnh SQL được gửi từ ứng dụng đến Database, nó mất một khoảng thời gian chờ (Latency) để kết nối, xác thực và truyền dữ liệu.

Giải pháp: Thay vì thực hiện 101 chuyến đi (như trong bài toán N+1), .Include() gộp tất cả yêu cầu vào 1 chuyến đi duy nhất. Điều này giúp ứng dụng phản hồi nhanh hơn đáng kể, đặc biệt khi Database đặt ở xa (Cloud).

2. Tận dụng sức mạnh của SQL JOIN

Khi bạn dùng .Include(), EF Core sẽ thông minh dịch mã C# của bạn thành một câu lệnh SQL có sử dụng các phép JOIN (thường là LEFT JOIN).

Các hệ quản trị cơ sở dữ liệu (như SQL Server, PostgreSQL) được thiết kế để xử lý các phép JOIN cực kỳ tối ưu. Việc tìm kiếm dữ liệu liên quan ở tầng Database luôn nhanh hơn hàng chục lần so với việc để ứng dụng C# đi tìm từng mảnh một.

3. Tiết kiệm tài nguyên Server (CPU & RAM)

Phía Database: Việc xử lý 1 câu lệnh lớn thường ít tốn tài nguyên hơn là phải quản lý, mở/đóng và thực thi hàng trăm câu lệnh nhỏ liên tục.

Phía Ứng dụng: Bạn tránh được việc tạo ra hàng trăm đối tượng tạm thời trong bộ nhớ chỉ để phục vụ cho các câu truy vấn lẻ tẻ, giúp giảm tải cho bộ dọn rác (Garbage Collector).

### **3. Ví dụ thực hành (Bắt buộc)**

*Yêu cầu: Triển khai một ProductService với các thao tác CRUD cơ bản*

#### **Mục tiêu:**

* Tạo ProductService class với các method: Create, GetById, GetAll, Update, Delete
* Sử dụng async/await
* Áp dụng Include để tránh N+1 problem

#### **Viết code cho Services/ProductService.cs:**

public class ProductService

{

private readonly AppDbContext \_context;

public ProductService(AppDbContext context)

{

\_context = context;

}

// CREATE — Thêm sản phẩm mới

public async Task<Product> CreateAsync(Product product)

{

// TODO: Viết code để:

// 1. Set CreatedAt = DateTime.UtcNow

// 2. Thêm product vào DbSet

// 3. Lưu vào Database

// 4. Return product

}

// READ — Lấy tất cả sản phẩm (có Include Category)

public async Task<List<Product>> GetAllAsync()

{

// TODO: Viết code để:

// 1. Lấy tất cả sản phẩm từ Database

// 2. Include thông tin Category

// 3. Sắp xếp theo tên (OrderBy)

// 4. Return list

}

// READ — Lấy 1 sản phẩm theo ID (có Include Category)

public async Task<Product?> GetByIdAsync(int id)

{

// TODO: Viết code để:

// 1. Tìm sản phẩm với id đã cho

// 2. Include thông tin Category

// 3. Return product hoặc null nếu không tìm thấy

}

// UPDATE — Cập nhật sản phẩm

public async Task<Product> UpdateAsync(int id, Product updateData)

{

// TODO: Viết code để:

// 1. Tìm sản phẩm với id đã cho

// 2. Throw exception nếu không tìm thấy

// 3. Cập nhật các properties (Name, Price, Stock, CategoryId)

// 4. Set UpdatedAt = DateTime.UtcNow

// 5. Lưu vào Database

// 6. Return product đã cập nhật

}

// DELETE — Xóa sản phẩm (Soft Delete)

public async Task DeleteAsync(int id)

{

// TODO: Viết code để:

// 1. Tìm sản phẩm với id đã cho

// 2. Throw exception nếu không tìm thấy

// 3. Set IsDeleted = true

// 4. Set DeletedAt = DateTime.UtcNow

// 5. Lưu vào Database

// Lưu ý: Phải thêm properties IsDeleted và DeletedAt vào Entity trước!

}

// SEARCH & FILTER

public async Task<List<Product>> SearchAsync(string? searchTerm, int? categoryId)

{

// TODO: Viết code để:

// 1. Bắt đầu query từ \_context.Products

// 2. Nếu searchTerm không null → Where(p => p.Name.Contains(searchTerm))

// 3. Nếu categoryId có giá trị → Where(p => p.CategoryId == categoryId)

// 4. Include Category

// 5. Return list kết quả

}

}

#### **Đăng ký ProductService trong Program.cs:**

Thêm dòng code sau (trong file Program.cs):

// Đăng ký ProductService vào DI Container

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### **4. Kiểm tra kết quả (Optional - Dùng Postman/Thunder Client)**

Nếu bạn đã tạo API Controller cho Products, hãy:

1. **Chụp ảnh kết quả** gọi API GET /api/products (nên có ít nhất 2-3 sản phẩm)
2. **Chụp ảnh kết quả** gọi API POST /api/products (thêm sản phẩm mới)
3. **Chụp ảnh kết quả** gọi API PUT /api/products/{id} (cập nhật sản phẩm)
4. **Chụp ảnh kết quả** gọi API DELETE /api/products/{id} (xóa mềm sản phẩm)

**Ảnh kết quả Postman (Dán ảnh vào đây):**

*(Dán ảnh 1)* *(Dán ảnh 2)* *(Dán ảnh 3)* *(Dán ảnh 4)*

### **5. Câu hỏi/Thắc mắc của tôi**

**Ba điều tôi chưa hiểu rõ hoặc muốn hỏi thêm:**

1. Sự khác biệt giữa Add() và AddAsync() là gì?
2. Làm sao để Update dữ liệu mà không cần tải nó lên trước? (Bulk Update)

## **PHIẾU HỌC TẬP EF CORE [03] — DATA ANNOTATIONS & FLUENT API**

**Video/Tài liệu đã xem (Link):** **https://www.youtube.com/watch?v=Yt36ofc5p94&t=153s**

### **1. Kiến thức cốt lõi (So sánh Annotations vs Fluent API)**

| **Tính năng** | **Data Annotations** | **Fluent API** |
| --- | --- | --- |
| **Vị trí code** | *Viết trực tiếp trong Model Class (trên các property).* | *Viết trong phương thức OnModelCreating của DbContext.* |
| **Dễ dùng** | *Dễ và nhanh: Chỉ cần thêm các thẻ như [Key], [Required].* | *Phức tạp hơn: Đòi hỏi viết code chuỗi (Chaining method).* |
| **Khả năng mạnh mẽ** | *Hạn chế: Chỉ cấu hình được những tính năng cơ bản.* | *Rất mạnh: Cấu hình được mọi thứ, kể cả các quan hệ phức tạp.* |
| **Ưu tiên nếu xung đột** | *Thấp hơn.* | *Cao nhất: Ghi đè (Override) mọi cấu hình từ Annotations.* |

### **2. Ví dụ thực hành (Cải tiến Entity Classes)**

*Yêu cầu: Sử dụng Data Annotations để cấu hình Entity Classes hoàn chỉnh*

#### **Cải tiến Category.cs:**

public class Category

{

[Key] // ← Chỉ định Primary Key

public int Id { get; set; }

[Required] // ← TODO: Điền annotation thích hợp để bắt buộc Name không null

[MaxLength(100)] // ← Giới hạn độ dài 100 ký tự

public string Name { get; set; } = string.Empty;

[MaxLength(500)]

public string? Description { get; set; } // ← Optional (nullable)

// TODO: Thêm CreatedAt property với annotation thích hợp

public ICollection<Product> Products { get; set; } = new List<Product>();

}

#### **Cải tiến Product.cs:**

public class Product

{

public int Id { get; set; }

[Required]

[MaxLength(200)]

public string Name { get; set; } = string.Empty;

[MaxLength(2000)]

public string? Description { get; set; }

// TODO: Thêm annotation để chỉ định kiểu dữ liệu là decimal(18,2)

[Column(TypeName = "decimal(18,2)")]

public decimal Price { get; set; }

[Range(0, 10000)] // ← TODO: Giải thích ý nghĩa của Range

public int Stock { get; set; }

// TODO: Thêm các properties khác (CreatedAt, UpdatedAt, IsDeleted, DeletedAt)

[ForeignKey("CategoryId")] // ← Chỉ định Foreign Key

public int CategoryId { get; set; }

public Category Category { get; set; } = null!;

}

### **3. Sử dụng Fluent API (Tùy chọn - Dành cho bạn muốn hiểu sâu)**

Viết Fluent API cấu hình trong **AppDbContext.cs**:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

// TODO: Viết Fluent API để cấu hình Product table

// (Gợi ý: Set table name, Primary Key, Properties)

modelBuilder.Entity<Product>(entity =>

{

entity.ToTable("Products");

entity.HasKey(p => p.Id);

// Cấu hình Name

entity.Property(p => p.Name)

.IsRequired()

.HasMaxLength(200);

// TODO: Cấu hình Price, Stock, và các properties khác

// Cấu hình Relationship

entity.HasOne(p => p.Category)

.WithMany(c => c.Products)

.HasForeignKey(p => p.CategoryId)

.OnDelete(DeleteBehavior.Cascade); // ← Nếu xóa Category thì Products bị xóa theo

});

}

### **4. Tạo Migration mới & Apply**

Sau khi cập nhật Entity Classes, chạy lệnh:

Add-Migration UpdateProductAndCategoryModels

Update-Database

**Chụp ảnh cấu trúc bảng Products trong SSMS:**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Nội dung do AI tạo ra có thể không chính xác.

### **5. Câu hỏi/Thắc mắc của tôi**

1. Khi nào BẮT BUỘC phải dùng Fluent API thay vì Data Annotations?

## **MINI TEST**

*Trả lời các câu hỏi sau để tự kiểm tra hiểu biết của mình:*

### **Phần 1: ORM & Code First**

**1. Code First có nghĩa là?**

* A. Viết SQL trước, sau đó tạo C# classes
* B. Viết C# classes trước, Database tự tạo từ code
* C. Đặt code trước SQL trong project

**2. DbContext là gì?**

* A. Một database server
* B. Cổng giao tiếp giữa C# code và Database
* C. Một bảng trong database

**3. Migration dùng để làm gì?**

* A. Backup database
* B. Tạo và cập nhật database schema
* C. Tối ưu performance

### **Phần 2: CRUD & Relationships**

**4. Eager Loading dùng method nào?**

* A. .Load()
* B. .Include()
* C. .Join()

**5. N+1 Query Problem xảy ra khi?**

* A. Không dùng Include → mỗi navigation property access = 1 query
* B. Dùng quá nhiều Include
* C. Query quá phức tạp

**6. Soft Delete khác Hard Delete?**

* A. Không khác
* B. Soft Delete đánh dấu (IsDeleted=true), Hard Delete xóa hẳn
* C. Soft Delete nhanh hơn

### **Phần 3: Annotations & Fluent API**

**7. [Required] annotation có ý nghĩa gì?**

* A. Property là optional
* B. Property không được phép null
* C. Property là Primary Key

**8. Data Annotations vs Fluent API, cái nào có quyền ưu tiên cao hơn khi xung đột?**

* A. Data Annotations
* B. Fluent API
* C. Bằng nhau

## **Đáp án Mini Test**

### 👉 Nhấn vào đây để xem đáp án **Phần 1:**

1. **B** - Viết C# classes trước, Database tự tạo
2. **B** - Cổng giao tiếp giữa C# code và Database
3. **B** - Tạo và cập nhật database schema

### **Phần 2:**

1. **B** - .Include()
2. **A** - Không dùng Include → mỗi access = 1 query
3. **B** - Soft Delete đánh dấu, Hard Delete xóa hẳn

### **Phần 3:**

1. **B** - Property không được phép null
2. **B** - Fluent API có ưu tiên cao hơn

## **📌 TÓM TẮT CHƯƠNG (Self-Reflection)**

**Sau khi hoàn thành các phiếu học tập, hãy tự mô tả:**

### **Những gì tôi đã hiểu:**

Tư duy về ORM và EF Core

ORM là cầu nối: Bạn đã hiểu ORM (Object-Relational Mapping) giúp xóa bỏ rào cản giữa lập trình hướng đối tượng (C#) và dữ liệu quan hệ (SQL).

EF Core vs ADO.NET: Bạn nhận ra EF Core giúp tăng năng suất cực nhanh nhờ giảm bớt code lặp lại, dù ADO.NET vẫn mạnh về hiệu năng thuần túy.

2. Quy trình "Code First" Workflow

Bạn đã nắm vững 5 bước để biến ý tưởng thành Database thực tế:

Model: Tạo class (C#).

DbContext: Thiết lập cấu hình kết nối.

Migration: Chụp ảnh sự thay đổi (Add-Migration).

Update: Áp dụng vào SQL (Update-Database).

Result: Database được sinh ra tự động.

3. Thiết kế Quan hệ (Relationships)

Navigation Property: Biết dùng ICollection<Product> trong Category để lấy danh sách sản phẩm liên quan.

Foreign Key: Biết dùng CategoryId trong Product để tạo sợi dây liên kết cứng ở tầng Database.

Null-safety: Hiểu ý nghĩa của = null!; để làm hài lòng trình biên dịch C# mà vẫn đảm bảo dữ liệu không được rỗng.

4. Hiệu năng và Truy vấn (Performance)

N+1 Problem: Bạn đã nhận diện được "sát thủ" làm chậm hệ thống khi dùng vòng lặp để truy vấn dữ liệu liên quan mà không nạp trước.

Eager Loading: Biết cách dùng .Include() để gom tất cả dữ liệu vào 1 câu lệnh SQL duy nhất nhằm tối ưu tốc độ.

CRUD: Thành thạo các phương thức Add, Find, Remove và hiểu rằng mọi thứ chỉ thành hiện thực khi gọi SaveChangesAsync().

5. Cấu hình Model (Mapping)

Data Annotations: Dùng Attribute nhanh, gọn ngay tại Class Model.

Fluent API: Cách cấu hình mạnh mẽ, chuyên nghiệp trong OnModelCreating, giúp code sạch và xử lý được các quan hệ phức tạp nhất.

### **Những gì tôi vẫn chưa rõ:**

### **Mục tiêu tiếp theo:**

**Ghi chú của giáo viên:**

**Ngày hoàn thành:** 03/01/2026

**Chữ ký sinh viên:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_