# **PHIẾU HỌC TẬP ENTITY FRAMEWORK CORE**

**Họ và tên:** Phạm Anh Tài

**Mã sinh viên:** 1871020512

**Ngày làm:**

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

**Video/Tài liệu đã xem (Link):**

https://youtu.be/prumncXXtew?si=8ba1854BySRqjsoi

https://youtu.be/V5yP5P1gIlE?si=tc2HePX-vkK-ww4D

### **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?)

+ Giúp lập trình viên **không phải viết SQL thủ công quá nhiều**

+ Làm việc với database bằng **object quen thuộc của C#**

+ Giảm lỗi SQL, dễ bảo trì, dễ mở rộng hệ thống

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

EF Core là một **ORM framework** trong .NET để làm việc với **cơ sở dữ liệu**

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

* Tự động mapping giữa class và table
* Hỗ trợ LINQ → code dễ đọc, dễ viết hơn
* Quản lý migration, schema database rất tiện

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

**Quy trình 5 bước:**

1. Viết **Entity** **Classes** (Product, Category)

↓

2. Tạo **DbContext**

↓

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

↓

4. Update **Database**

↓

5. **Database** **&** **Tables** 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)

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

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

**public string? Description { get; set; }**

// 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)

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

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

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

// 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?

+ Báo cho compiler rằng: *lập trình viên đảm bảo biến này sẽ không null khi chạy thực tế*

+ Tránh cảnh báo nullable reference khi bật Nullable Reference Types

+ EF Core sẽ tự động gán giá trị cho Category khi truy vấn dữ liệu từ database

#### **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>)

public DbSet<Category> Categories { get; set; }

public DbSet<Product> Products { get; set; }

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 = **builder.Configuration.GetConnectionString("DefaultConnection");**

// Bước 2: Đăng ký DbContext với DI Container

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(connectionString)

);

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

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

{

"ConnectionStrings": {

"DefaultConnection": "Server=.;Database=EShopDb;Trusted\_Connection=True;TrustServerCertificate=True"

}

}

**Giải thích:**

* Server=. có nghĩa là: Kết nối tới SQL Server cài trên **máy local**
* Database=EShopDb có nghĩa là: Tên database sẽ được tạo
* Trusted\_Connection=True có nghĩa là: Dùng Windows Authentication (không cần user/password)

#### **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?

Là file mô tả **sự thay đổi cấu trúc database**

Chứa code tạo bảng Categories, Products, khóa chính, khóa ngoại

EF Core dùng migration để **quản lý phiên bản database**

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):**

*(Dán ảnh chụp từ SSMS)*

### **3. Câu hỏi/Thắc mắc của tôi**

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

1. Khi nào nên dùng Code First và khi nào nên dùng Database First?
2. EF Core có phù hợp với hệ thống lớn không?
3. Khi nào cần viết SQL thuần thay vì dùng EF Core?

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

**Video/Tài liệu đã xem (Link):**

https://youtu.be/\_N\_5zxlfrgw?si=\_klD4M1N9KG6Jlk5

### **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** | **Create** - Tạo mới bản ghi | \_context.Products.Add() | *await \_context.Products.AddAsync(product);* |
| **R** | **Read** - Đọc dữ liệu | \_context.Products.Find() hoặc FirstOrDefaultAsync() | *await \_context.Products.FirstOrDefaultAsync(p => p.Id == id);* |
| **U** | **Update** - Cập nhật bản ghi | SaveChangesAsync() (sau khi modify property) | *product.Name = "Updated Name"; await \_context.SaveChangesAsync();* |
| **D** | **Delete** - Xóa bản ghi | \_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?

* Một Category có thể có nhiều Products
* Một Product chỉ thuộc về một Category duy nhất
* Quan hệ cha-con, phân cấp dữ liệu

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>?

Để EF Core biết đây là quan hệ 1-n, giúp truy vấn navigation property và thực hiện eager loading

* Tại sao Product cần CategoryId?

Là foreign key để thiết lập quan hệ với bảng Categories, đảm bảo tính toàn vẹn dữ liệu

#### **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?**

* EF Core lazy loading: Khi truy cập navigation property product.Category, nó sẽ thực hiện query riêng để lấy dữ liệu Category
* Với N products → thực hiện N queries riêng biệt + 1 query ban đầu = N+1 queries

**✅ 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?**

* Eager loading với Include() join 2 bảng trong 1 query duy nhất
* Giảm số lượng round-trip đến database
* Tăng hiệu suất đáng kể, đặc biệt với dữ liệu lớn

### **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

product.CreatedAt = DateTime.UtcNow;

// 2. Thêm product vào DbSet

await \_context.Products.AddAsync(product);

// 3. Lưu vào Database

await \_context.SaveChangesAsync();

// 4. Return product

await \_context.Entry(product)

.Reference(p => p.Category)

.LoadAsync();

return product;

}

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

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

{

// TODO: Viết code để:

return await \_context.Products

.Include(p => p.Category)

.Where(p => !p.IsDeleted)

.OrderBy(p => p.Name)

.ToListAsync();

}

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

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

{

// TODO: Viết code để:

return await \_context.Products

.Include(p => p.Category)

.FirstOrDefaultAsync(p => p.Id == id && !p.IsDeleted);

}

// UPDATE — Cập nhật sản phẩm

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

{

// TODO: Viết code để:

var product = await \_context.Products.FindAsync(id);

if (product == null || product.IsDeleted)

throw new Exception("Product not found");

product.Name = updateData.Name;

product.Price = updateData.Price;

product.Stock = updateData.Stock;

product.CategoryId = updateData.CategoryId;

product.UpdatedAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

return product;

}

// DELETE — Xóa sản phẩm (Soft Delete)

public async Task DeleteAsync(int id)

{

// TODO: Viết code để:

var product = await \_context.Products.FindAsync(id);

if (product == null)

throw new Exception("Product not found");

product.IsDeleted = true;

product.DeletedAt = DateTime.UtcNow;

await \_context.SaveChangesAsync();

}

// SEARCH & FILTER

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

{

// TODO: Viết code để:

// 1. Bắt đầu query

IQueryable<Product> query = \_context.Products;

// 2. Search theo tên sản phẩm

if (!string.IsNullOrWhiteSpace(searchTerm))

{

query = query.Where(p => p.Name.Contains(searchTerm));

}

// 3. Filter theo Category

if (categoryId.HasValue)

{

query = query.Where(p => p.CategoryId == categoryId.Value);

}

// 4 + 5. Include Category và return kết quả

return await query

.Include(p => p.Category)

.ToListAsync();

}

}

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

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

// Đăng ký ProductService vào DI Container

builder.Services.AddScoped<ProductService>();

### **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. Khi nào nên dùng **Include** và khi nào nên dùng **Lazy Loading**?
2. Sự khác nhau giữa **Soft Delete** và **Hard Delete** là gì?
3. Khi hệ thống lớn, nên đặt **Service Layer** hay **Repository Pattern**?

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

**Video/Tài liệu đã xem (Link):**

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

| **Tính năng** | **Data Annotations** | **Fluent API** |
| --- | --- | --- |
| **Vị trí code** | *Trực tiếp trên Entity class (attribute)* | *Trong DbContext → OnModelCreating()* |
| **Dễ dùng** | *Dễ, nhanh, dễ nhớ* | *Khó hơn, nhiều code* |
| **Khả năng mạnh mẽ** | *Hạn chế* | *Rất mạnh, cấu hình được mọi thứ* |
| **Ưu tiên nếu xung đột** | *Thấp hơn* | *Cao hơn (Fluent API override)* |

### **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

[Required]

public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

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)

public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

public DateTime? UpdatedAt { get; set; }

public bool IsDeleted { get; set; } = false;

public DateTime? DeletedAt { get; set; }

[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 Price

entity.Property(p => p.Price)

.HasColumnType("decimal(18,2)")

.IsRequired();

// Cấu hình Stock

entity.Property(p => p.Stock)

.IsRequired();

// Cấu hình các cột thời gian

entity.Property(p => p.CreatedAt)

.IsRequired();

entity.Property(p => p.UpdatedAt)

.IsRequired(false);

entity.Property(p => p.IsDeleted)

.HasDefaultValue(false);

// 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:**

*(Dán ảnh)*

### **5. Câu hỏi/Thắc mắc của tôi**

1. Khi nào nên ưu tiên dùng Fluent API thay vì Data Annotations?
2. Fluent API có ảnh hưởng đến performance khi chạy ứng dụng không?

## **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:**

1. ORM và EF Core giúp làm việc với cơ sở dữ liệu bằng C# object thay vì viết SQL thuần
2. Code First cho phép thiết kế database bắt đầu từ Entity Classes và quản lý bằng Migration
3. CRUD, Include và các mối quan hệ (1-N) giúp truy vấn dữ liệu hiệu quả và đúng thiết kế

### **Những gì tôi vẫn chưa rõ:**

1. Cách tối ưu performance EF Core trong hệ thống lớn
2. Khi nào nên dùng Fluent API thay vì Data Annotations trong thực tế

### **Mục tiêu tiếp theo:**

Xây dựng API hoàn chỉnh với **Pagination, Filtering, Authentication** và tối ưu truy vấn EF Core

**Ghi chú của giáo viên:**

**Ngày hoàn thành:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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