Student Guide: CRUD Operations with SQLite & Entity Framework Core

From Zero to Functional Application with Complete Conceptual Understanding

Table of Contents

1. Introduction to Entity Framework Core

1.1 What is Entity Framework (EF) Core?

- **Definition**: A lightweight, cross-platform **Object-Relational Mapper (ORM)** that:
  - Maps C# classes (**models**) to database tables
  - Translates LINQ queries to SQL commands
  - Automatically tracks changes to objects
  - Manages database connections and transactions
- **Key Benefits**:
  - Write database code using C# instead of raw SQL
  - Database-agnostic (supports SQLite, SQL Server, PostgreSQL, etc.)
  - Reduces boilerplate code for common operations

1.2 Key Components

Component	Role
DbContext	Central class for database interaction (connection, queries, transactions)
DbSet<T>	Represents a database table (e.g., `DbSet<Product>` = Products table)
LINQ	Language-Integrated Querying (write queries in C# syntax)
Migrations	Version control system for database schema changes

2. Project Setup & Configuration

2.1 Create a New Console Application

Steps:

1. Open Visual Studio 2022 → New Project → "Console App"
2. Name: `ProductManager` → .NET 8.0

Why Console App?: Focus on EF Core logic without UI distractions.

2.2 Install Required NuGet Packages

Packages:

- `Microsoft.EntityFrameworkCore.Sqlite`
- `Microsoft.EntityFrameworkCore.Tools`

Installation:

1. Right-click project → "Manage NuGet Packages"
2. Search and install packages

Why Required?:

- `Sqlite` package adds SQLite database support
- `Tools` (`Design`) enables EF Core tools like `EnsureCreated()`

---

3. Data Modeling & Database Context

3.1 Create the Model Class

File: `Models/Product.cs`

```
public class Product
{
    public int Id { get; set; }          // Primary key (auto-detected)
    public string? Name { get; set; }    // Nullable string
    public decimal Price { get; set; }
    public int Stock { get; set; }
}
```

Key Points:

- **Id Property**: EF Core automatically detects `Id` as the primary key.
- **Nullable Reference (`string?`)**: Allows `NULL` values in the database.
- **Data Types**: Maps to SQLite types (e.g., `decimal` stored as `TEXT`).

3.2 Configure Database Context

File: `Data/AppDbContext.cs`

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; } = null!; // Products table

    protected override void OnConfiguring(DbContextOptionsBuilder options)
    {
        // Configure database path in project root
        var path = Path.Combine(

Directory.GetParent(Directory.GetCurrentDirectory())!.Parent!.Parent!.FullName,
            "products.db"
        );
        options.UseSqlite($"Data Source={path}");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Optional: Explicit table name configuration
        modelBuilder.Entity<Product>().ToTable("Products");
    }
}
```

Line-by-Line Explanation:

- `DbSet<Product>`: Represents the `Products` table in the database.
- `OnConfiguring`:
  - `Directory.GetCurrentDirectory()`: Returns `bin/Debug/net7.0` at runtime.
  - `.Parent.Parent.Parent`: Navigates to the **project root folder**.
  - **Why?**: Ensures the app uses the same database file during development and runtime.
- `OnModelCreating`: Used for advanced configuration (e.g., custom table names).

---

4. Database Initialization

4.1 Initialize Database on Startup

File: `Program.cs`

```
using Microsoft.EntityFrameworkCore;

// Initialize database
using (var context = new AppDbContext())
{
    context.Database.EnsureCreated();
    Console.WriteLine("Database initialized!");
}
```

Key Concepts:

- `EnsureCreated()`:
  - Checks if the database exists.
  - If not: Creates database **and** tables based on `DbSet<T>` properties.
  - **No Migrations**: Simple but unsuitable for schema updates later.
- `using` **Statement**: Ensures proper disposal of `DbContext` resources.

---

5. CRUD Operations Implementation (code these in `Program.cs`)

5.1 Create (Add Product)

```
static void AddProduct()
{
    using var context = new AppDbContext(); // Short-lived context

    Console.Write("Enter product name: ");
    var name = Console.ReadLine()!;

    Console.Write("Enter price: ");
    var price = decimal.Parse(Console.ReadLine()!);

    Console.Write("Enter stock quantity: ");
    var stock = int.Parse(Console.ReadLine()!);

    // Create and add product
    context.Products.Add(new Product { Name = name, Price = price, Stock = stock
});
    context.SaveChanges(); // Execute INSERT
    Console.WriteLine("Product added!");
}
```

Workflow:

1. `Add()`: Marks the object for insertion.
2. `SaveChanges()`: Generates and executes `INSERT` SQL.

5.2 Read (List Products)

```
static void ListProducts()
{
    using var context = new AppDbContext();

    var products = context.Products.ToList(); // Execute SELECT *

    Console.WriteLine("\n{0,-5} {1,-20} {2,-10} {3}", "ID", "Name", "Price",
"Stock");
```

```
    foreach (var p in products)
    {
        Console.WriteLine($"{p.Id,-5} {p.Name,-20} {p.Price,-10:C} {p.Stock}");
    }
}
```

Key Points:

- `ToList()`: Forces immediate query execution.
- Without it: Returns `IQueryable` (deferred execution).

5.3 Update (Modify Product)

```
static void UpdateProduct()
{
    using var context = new AppDbContext();

    Console.Write("Enter product ID to update: ");
    var id = int.Parse(Console.ReadLine()!);

    // Find product (SELECT + tracking)
    var product = context.Products.Find(id);
    if (product == null)
    {
        Console.WriteLine("Product not found!");
        return;
    }

    // Update properties
    Console.Write($"New name [{product.Name}]: ");
    var name = Console.ReadLine();
    if (!string.IsNullOrWhiteSpace(name)) product.Name = name;

    Console.Write($"New price [{product.Price}]: ");
    var priceInput = Console.ReadLine();
    if (!string.IsNullOrWhiteSpace(priceInput))
        product.Price = decimal.Parse(priceInput);

    context.SaveChanges(); // Execute UPDATE
    Console.WriteLine("Updated successfully!");
}
```

EF Core Change Tracking:

1. `Find()` retrieves and tracks the entity.
2. Property changes mark the entity as modified.
3. `SaveChanges()` generates `UPDATE` SQL.

5.4 Delete (Remove Product)

```
static void DeleteProduct()
{
    using var context = new AppDbContext();

    Console.Write("Enter product ID to delete: ");
    var id = int.Parse(Console.ReadLine()!);

    var product = context.Products.Find(id);
    if (product == null)
    {
        Console.WriteLine("Product not found!");
        return;
    }

    context.Products.Remove(product); // Mark for deletion
    context.SaveChanges(); // Execute DELETE
    Console.WriteLine("Product deleted!");
}
```

Transaction Flow:

- `Remove()` flags the entity for deletion.
- Changes are batched and executed on `SaveChanges()`.

5.5 Main Menu

```
while (true)
{
    Console.WriteLine("\n1. Add Product");
    Console.WriteLine("2. List Products");
    Console.WriteLine("3. Update Product");
    Console.WriteLine("4. Delete Product");
    Console.WriteLine("5. Exit");
    Console.Write("Choose an option: ");

    switch (Console.ReadLine())
    {
        case "1": AddProduct(); break;
        case "2": ListProducts(); break;
        case "3": UpdateProduct(); break;
        case "4": DeleteProduct(); break;
        case "5": return;
        default: Console.WriteLine("Invalid option!"); break;
    }
}
```

---

6. Core Concepts Deep Dive

6.1 DbContext Lifecycle

- **Short-Lived**: Created per operation (recommended).
- **Disposal**: `using` ensures connections are closed promptly.
- **Change Tracking**: Tracks entity states (Added/Modified/Deleted).

6.2 SQLite Specifics

- **File-Based**: Single `.db` file – no server required.
- **Type Affinity**: Flexible typing (e.g., `TEXT` for decimals).
- **Case Sensitivity**: Table/column names are case-sensitive.

6.3 EnsureCreated() vs Migrations

Feature	EnsureCreated	Migrations
Schema Updates	Drops entire database	Incremental updates
Version History	None	Tracks changes via `__EFMigrationsHistory`
Use Case	Prototyping, simple apps	Production applications

7. Testing & Troubleshooting

7.1 Test Plan

1. **First Run**: Verify `products.db` is created in project root.
2. **Add Test Data**: Add 2-3 products with different prices.
3. **List Verification**: Confirm data appears correctly.
4. **Update Test**: Modify a product's price/name.
5. **Delete Test**: Remove an item and confirm deletion.

7.2 Common Errors & Solutions

Error	Solution
`Microsoft.Data.Sqlite.SqliteException: SQLite Error 1: 'no such table: Products'`	Check database path and run `EnsureCreated()`
`System.InvalidOperationException: No database provider has been configured.`	Verify `UseSqlite()` in `OnConfiguring`
`FormatException: Input string was not in a correct format.`	Add input validation (e.g., `TryParse`)

8. Best Practices & Next Steps

8.1 Best Practices

1. **Input Validation**: Use `TryParse` instead of direct parsing.

```
if (!int.TryParse(Console.ReadLine(), out int stock))
{
    Console.WriteLine("Invalid input!");
    return;
}
```

2. **Error Handling**: Wrap database operations in `try-catch` blocks.
3. **Async Operations**: Use `SaveChangesAsync()` for scalability.

8.2 Next Steps

1. **Add Relationships**: Create `Category` class with one-to-many relationship.
2. **Implement Migrations**: Use `Add-Migration` for schema versioning.
3. **Separate Layers**: Divide into Data Access Layer (DAL) and UI Layer.

This guide provides both practical implementation steps and deep conceptual understanding, ensuring you can confidently build and extend EF Core applications. Experiment with the code, modify the model, and explore relationships to reinforce your learning!