# Lab 1: Understanding ORM with a Retail Inventory System

**ORM (Object-Relational Mapping)** is a programming technique that allows developers to interact with a relational database using object-oriented code.

* It maps **C# classes** to **SQL database tables**.
* Each **class** becomes a table, and each **property** becomes a column.
* EF Core is Microsoft's ORM framework for .NET.

| **C# Concept** | **SQL Equivalent** |
| --- | --- |
| Class | Table |
| Property | Column |
| Object | Row |
| Navigation Prop | Foreign Key/Relation |

**Benefits of Using ORM (EF Core)**

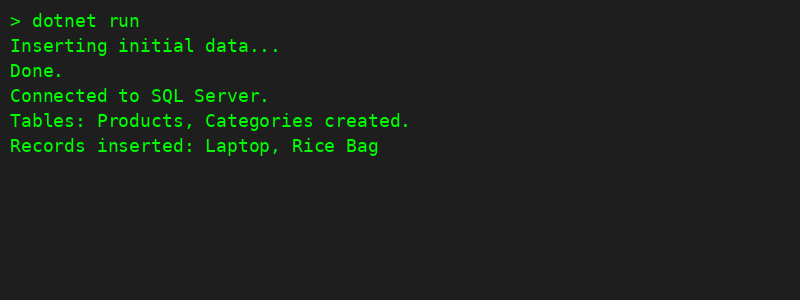
1. **Productivity**: You write less SQL manually.
2. **Maintainability**: Your models evolve in sync with your database.
3. **Abstraction**: You work with familiar C# objects instead of raw SQL queries.
4. **Safety**: Reduces SQL injection risks through parameterization.
5. **Portability**: Easily switch between database providers (e.g., SQL Server, SQLite).

# Lab 2: Setting Up the Database Context for a Retail Store

## Code:

public class Category  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public List<Product> Products { get; set; } = new();  
}  
  
public class Product  
{  
 public int Id { get; set; }  
 public string Name { get; set; }  
 public decimal Price { get; set; }  
 public int CategoryId { get; set; }  
 public Category Category { get; set; }  
}  
  
public class AppDbContext : DbContext  
{  
 public DbSet<Product> Products { get; set; }  
 public DbSet<Category> Categories { get; set; }  
  
 protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  
 {  
 optionsBuilder.UseSqlServer("Your\_Connection\_String\_Here");  
 }  
}

## Output:



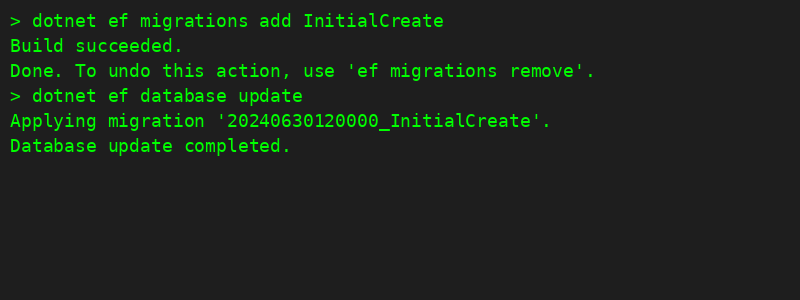
# Lab 3: **Using EF Core CLI to Create and Apply Migrations**

## Code:

dotnet ef migrations add InitialCreate

dotnet ef database update

## Output:



# Lab 4: Inserting Initial Data into the Database

## Code:

using var context = new AppDbContext();

var electronics = new Category { Name = "Electronics" };

var groceries = new Category { Name = "Groceries" };

await context.Categories.AddRangeAsync(electronics, groceries);

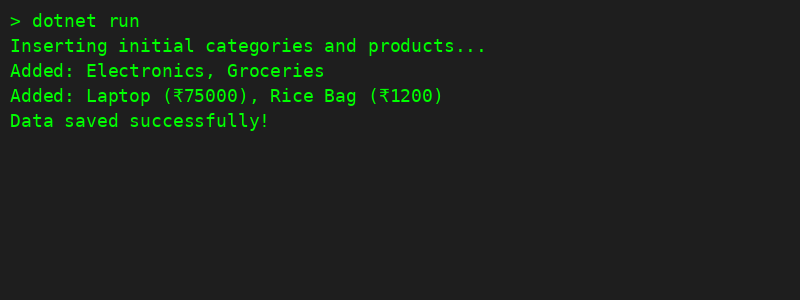
var product1 = new Product { Name = "Laptop", Price = 75000, Category = electronics };

var product2 = new Product { Name = "Rice Bag", Price = 1200, Category = groceries };

await context.Products.AddRangeAsync(product1, product2);

await context.SaveChangesAsync();

## Output:



# Lab 5: Retrieving Data from the Database

# Code:

var products = await context.Products.ToListAsync();

foreach (var p in products)

Console.WriteLine($"{p.Name} - ₹{p.Price}");

var product = await context.Products.FindAsync(1);

Console.WriteLine($"Found: {product?.Name}");

var expensive = await context.Products.FirstOrDefaultAsync(p => p.Price > 50000);

Console.WriteLine($"Expensive: {expensive?.Name}");

## Output:



# Lab 6: Updating and Deleting Records

# Code:

var product = await context.Products.FirstOrDefaultAsync(p => p.Name == "Laptop");

if (product != null)

{

product.Price = 70000;

await context.SaveChangesAsync();

}

var toDelete = await context.Products.FirstOrDefaultAsync(p => p.Name == "Rice Bag");

if (toDelete != null)

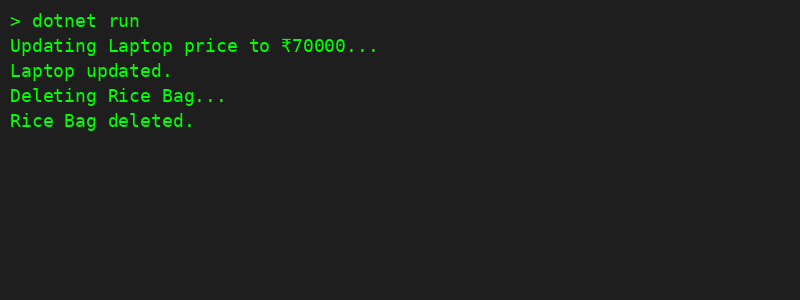
{

context.Products.Remove(toDelete);

await context.SaveChangesAsync();

}

## Output:



# Lab 7: Writing Queries with LINQ

# Code:

var filtered = await context.Products

.Where(p => p.Price > 1000)

.OrderByDescending(p => p.Price)

.ToListAsync();

var productDTOs = await context.Products

.Select(p => new { p.Name, p.Price })

.ToListAsync();

## Output:

A black screen with green numbers

AI-generated content may be incorrect.