Ödev 1: Kütüphane Yönetim Sistemi - Çözüm (VS Code & SQLite & Klasik Program.cs)

Proje Dizin Yapısı

Projenizi oluşturduktan sonra, kodları daha düzenli tutmak için ana dizin içinde `Entities` ve `Data` adında iki klasör oluşturmalısınız. Tamamlandığında proje yapınız aşağıdaki gibi görünmelidir:

```
LibraryManagementSystem/
|
|-- Data/
|   |-- LibraryAppDbContext.cs
|
|-- Entities/
|   |-- Author.cs
|   |-- Book.cs
|
|-- Migrations/
|   |-- ... (EF Core tarafından otomatik oluşturulur)
|
|-- Program.cs
|-- LibraryManagementSystem.csproj
|-- Library.db  (update-database komutundan sonra oluşur)
```

Adım 1: Proje Oluşturma ve Gerekli Paketlerin Yüklenmesi

Yeni Proje Oluşturma
Terminalinizi açın ve .NET CLI kullanarak klasik `Main` metodu yapısına sahip yeni bir konsol projesi oluşturun.

```
dotnet new console -n LibraryManagementSystem --use-program-main
```

1.
2. **Gerekli EF Core Paketlerini Ekleme**
   Terminalinizde, yeni oluşturulan `LibraryManagementSystem` proje klasörünün içine gidin ve aşağıdaki komutları sırasıyla çalıştırın.

Veritabanı sağlayıcısı olarak SQLite kullanacağımız için bu paketi ekliyoruz:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

- o

Migration gibi `dotnet ef` komutlarını terminalden çalıştırabilmek için bu aracı ekliyoruz:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

- o

Adım 2: Entity Sınıflarının Oluşturulması

Yukarıda belirtilen `Entities` klasörü içine `Author.cs` ve `Book.cs` dosyalarını oluşturup aşağıdaki kodları ekleyin.

`Entities/Author.cs`

```csharp
using System.Collections.Generic;

namespace LibraryManagementSystem.Entities
{
    public class Author
    {
        public int Id { get; set; }
        public string FullName { get; set; }
        public int BirthYear { get; set; }

        // Navigation property: Bir yazarın birden fazla kitabı olabilir.
        public ICollection<Book> Books { get; set; } = [];
    }
}```

#### `Entities/Book.cs`

```csharp
namespace LibraryManagementSystem.Entities
{
    public class Book
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public int PageCount { get; set; }

        // Foreign key property
        public int AuthorId { get; set; }

        // Navigation property: Her kitap bir yazara aittir.
        public Author Author { get; set; }
    }
}
```

Neden `List<T>` Değil de `ICollection<T>` Kullanıyoruz?

Navigation property'lerde (bir entity'nin ilişkili olduğu diğer entity'leri tutan özellikler) `List<T>` yerine `ICollection<T>` arayüzünü tercih etmemizin temel nedenleri şunlardır:

1. **Soyutlama ve Esneklik:** `ICollection<T>` bir arayüzdür, `List<T>` ise somut bir sınıftır. Arayüz kullanarak, EF Core'un arka planda kendi özel koleksiyon tiplerini (örneğin değişiklik takibi yapabilen, lazy loading'i destekleyen) kullanmasına olanak tanırız. Kodumuzu belirli bir implementasyona (`List<T>`) bağlamamış oluruz.
2. **Amacın Belirtilmesi:** `ICollection<T>`'nin amacı, bir eleman grubunu temsil etmektir (ekleme, silme, sayma gibi temel işlemlerle). `List<T>` ise ek olarak sıralama, index ile erişim gibi özellikler sunar. Genellikle, bir yazarın kitapları koleksiyonu için bu ek özelliklere ihtiyaç duymayız. `ICollection<T>` kullanmak, "bu sadece ilişkili nesnelerin bir koleksiyonudur" mesajını daha net verir.
3. **Genel Kabul Görmüş Standart:** Entity Framework Core dünyasında bu, en iyi pratik (best practice) olarak kabul edilir ve kodun daha temiz ve sürdürülebilir olmasını sağlar.

Adım 3: DbContext Sınıfının Oluşturulması
```

`Data` klasörü içine `LibraryAppDbContext.cs` adında bir dosya oluşturun.

`Data/LibraryAppDbContext.cs`

```
using LibraryManagementSystem.Entities;
using Microsoft.EntityFrameworkCore;

namespace LibraryManagementSystem.Data
{
    public class LibraryAppDbContext : DbContext
    {
        public DbSet<Author> Authors { get; set; }
        public DbSet<Book> Books { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            // Veritabanı dosyası olarak proje ana dizininde "Library.db" adında
bir dosya oluşturulacak.
            optionsBuilder.UseSqlite("Data Source=Library.db");
        }
    }
}
```

Adım 4: Veritabanını Oluşturma (Migration ve Update)

Şimdi terminal üzerinden veritabanı şemasını oluşturacağız.

Migration Dosyalarını Oluşturma
Bu komut, `LibraryAppDbContext` ve bağlı `Entity` sınıflarını tarayarak veritabanı şemasını oluşturacak/güncelleyecek C# kodlarını hazırlar. `InitialCreate` migration'a verdiğimiz bir addır.
`dotnet ef migrations add InitialCreate`

1. Bu komuttan sonra projenizde `Migrations` klasörü otomatik olarak oluşacaktır.

Veritabanını Fiziksel Olarak Oluşturma
Bu komut, bir önceki adımda oluşturulan migration dosyasını çalıştırarak `Library.db` veritabanı dosyasını proje dizininizde fiziksel olarak oluşturur.
`dotnet ef database update`

2.

Adım 5: Program.cs - Tüm Görevlerin Uygulanması

Son olarak, `Program.cs` dosyasını açıp ödevdeki tüm görevleri yerine getirecek şekilde aşağıdaki kodlarla güncelleyin.

`Program.cs`

```
using System;
using System.Linq;
using LibraryManagementSystem.Data;
using LibraryManagementSystem.Entities;
```

```
using Microsoft.EntityFrameworkCore;

namespace LibraryManagementSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            // Görev 3: Veritabanına başlangıç verilerini ekle
            AddInitialData();

            Console.WriteLine("-----------------------------------");

            // Görev 4: Kitapları yazarlarıyla birlikte listele (Projeksiyon)
            ListBooksWithAuthors();

            Console.WriteLine("-----------------------------------");

            // Görev 5: Belirli bir kitabın sayfa sayısını güncelle
            // ID'si 3 olan 'Suç ve Ceza' kitabının sayfa sayısını 700 yapalım.
            UpdateBookPageCount(3, 700);
        }

        public static void AddInitialData()
        {
            using (var context = new LibraryAppDbContext())
            {
                // Veritabanı zaten doluysa bu adımı atla
                if (context.Authors.Any())
                {
                    Console.WriteLine("Veritabanı zaten örnek verileri içeriyor.
Ekleme işlemi atlandı.");
                    return;
                }

                Console.WriteLine("Örnek veriler veritabanına ekleniyor...");

                var author1 = new Author { FullName = "Fyodor Dostoyevski",
BirthYear = 1821 };
                var author2 = new Author { FullName = "Yaşar Kemal", BirthYear =
1923 };
                var author3 = new Author { FullName = "George Orwell", BirthYear
= 1903 };
                var author4 = new Author { FullName = "J.K. Rowling", BirthYear
= 1965 };

                // Kitapları oluşturup doğrudan yazarın koleksiyonuna ekliyoruz.
                author1.Books.Add(new Book { Title = "Karamazov Kardeşler",
PageCount = 840 });
                author1.Books.Add(new Book { Title = "Yeraltından Notlar",
PageCount = 144 });
                author1.Books.Add(new Book { Title = "Suç ve Ceza", PageCount =
687 });
```

```
                author2.Books.Add(new Book { Title = "İnce Memed", PageCount =
436 });
                author2.Books.Add(new Book { Title = "Ağrıdağı Efsanesi",
PageCount = 128 });

                author3.Books.Add(new Book { Title = "1984", PageCount = 328 });
                // Bu yazarın sadece bir kitabı var

                author4.Books.Add(new Book { Title = "Harry Potter ve Felsefe
Taşı", PageCount = 223 });
                author4.Books.Add(new Book { Title = "Harry Potter ve Sırlar
Odası", PageCount = 251 });
                author4.Books.Add(new Book { Title = "Harry Potter ve Azkaban
Tutsağı", PageCount = 317 });
                author4.Books.Add(new Book { Title = "Harry Potter ve Ateş
Kadehi", PageCount = 636 });

                // Sadece yazarları context'e eklememiz yeterli.
                // EF Core, onlara bağlı olan yeni kitapları otomatik olarak
algılayacaktır.
                context.Authors.AddRange(author1, author2, author3, author4);
                context.SaveChanges();

                Console.WriteLine("Veriler başarıyla eklendi!");
            }
        }

        public static void ListBooksWithAuthors()
        {
            using (var context = new LibraryAppDbContext())
            {
                Console.WriteLine("--- Kitap Listesi (Yazar Adlarıyla Birlikte)
---");

                // İlişkili yazar verisini çekmek için Include() kullanıyoruz.
                // Sadece istediğimiz alanları almak için Select() ile
projeksiyon yapıyoruz.
                var booksList = context.Books
                    .Include(book => book.Author)
                    .Select(book => new
                    {
                        BookTitle = book.Title,
                        AuthorName = book.Author.FullName
                    })
                    .ToList();

                foreach (var item in booksList)
                {
                    // Metin formatlaması ile daha okunaklı bir çıktı
sağlıyoruz.
                    Console.WriteLine($"Kitap: {item.BookTitle,-40} | Yazar:
{item.AuthorName}");
                }
            }
        }
```

```
        public static void UpdateBookPageCount(int bookId, int newPageCount)
        {
            using (var context = new LibraryAppDbContext())
            {
                Console.WriteLine($"ID'si {bookId} olan kitap
güncelleniyor...");

                var bookToUpdate = context.Books.FirstOrDefault(b => b.Id ==
bookId);

                if (bookToUpdate != null)
                {
                    Console.WriteLine($"'{bookToUpdate.Title}' adlı kitabın
önceki sayfa sayısı: {bookToUpdate.PageCount}");

                    bookToUpdate.PageCount = newPageCount;
                    context.SaveChanges();

                    Console.WriteLine($"'{bookToUpdate.Title}' adlı kitabın yeni
sayfa sayısı: {bookToUpdate.PageCount}");
                    Console.WriteLine("Güncelleme başarılı!");
                }
                else
                {
                    Console.WriteLine($"Hata: ID'si {bookId} olan bir kitap
bulunamadı.");
                }
            }
        }
    }
}
```

Ödev 2: Şirket Departman ve Çalışan Yönetimi - Çözüm

Proje Dizin Yapısı

Projenizi oluşturduktan sonra, ana dizin içinde `Entities` ve `Data` adında iki klasör oluşturmalısınız. İşlemler tamamlandığında proje yapınız aşağıdaki gibi olacaktır:

```
CompanyManagementSystem/
|
|-- Data/
|   |-- CompanyAppDbContext.cs
|
|-- Entities/
|   |-- Department.cs
|   |-- Employee.cs
|
|-- Migrations/
|   |-- ... (EF Core tarafından otomatik oluşturulur)
|
|-- Program.cs
|-- CompanyManagementSystem.csproj
|-- Company.db  (update-database komutundan sonra oluşur)
```

Adım 1: Proje Oluşturma ve Gerekli Paketlerin Yüklenmesi

Yeni Proje Oluşturma
Terminalinizi açın ve .NET CLI kullanarak klasik `Main` metodu yapısına sahip yeni bir konsol projesi oluşturun.
`dotnet new console -n CompanyManagementSystem --use-program-main`

1.
2. **Gerekli EF Core Paketlerini Ekleme**
   Terminalinizde, yeni oluşturulan `CompanyManagementSystem` proje klasörünün içine gidin ve aşağıdaki komutları sırasıyla çalıştırın.

Veritabanı sağlayıcısı olarak SQLite kullanacağımız için bu paketi ekliyoruz:
`dotnet add package Microsoft.EntityFrameworkCore.Sqlite`
    - o

Migration gibi `dotnet ef` komutlarını terminalden çalıştırabilmek için bu aracı ekliyoruz:
`dotnet add package Microsoft.EntityFrameworkCore.Design`
    - o

Adım 2: Entity Sınıflarının Oluşturulması

Projenizde oluşturduğunuz `Entities` klasörü içine `Department.cs` ve `Employee.cs` dosyalarını ekleyin.

`Entities/Department.cs`

```
using System.Collections.Generic;

namespace CompanyManagementSystem.Entities
```

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

        // Navigation property: Bir departmanın birden fazla çalışanı olabilir.
        public ICollection<Employee> Employees { get; set; } = [];
    }
}
```

Entities/Employee.cs

```
namespace CompanyManagementSystem.Entities
{
    public class Employee
    {
        public int Id { get; set; }
        public string FullName { get; set; }
        public string Position { get; set; }
        public decimal Salary { get; set; }

        // Foreign key property
        public int DepartmentId { get; set; }

        // Navigation property: Her çalışan bir departmana aittir.
        public Department Department { get; set; }
    }
}
```

Adım 3: DbContext Sınıfının Oluşturulması

`Data` klasörü içine `CompanyAppDbContext.cs` adında bir dosya oluşturun.

Data/CompanyAppDbContext.cs

```
using CompanyManagementSystem.Entities;
using Microsoft.EntityFrameworkCore;

namespace CompanyManagementSystem.Data
{
    public class CompanyAppDbContext : DbContext
    {
        public DbSet<Department> Departments { get; set; }
        public DbSet<Employee> Employees { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            // Veritabanı dosyası olarak proje ana dizininde "Company.db" adında
bir dosya oluşturulacak.
```

```
            optionsBuilder.UseSqlite("Data Source=Company.db");
        }
    }
}
```

Adım 4: Veritabanını Oluşturma (Migration ve Update)

Şimdi terminal üzerinden veritabanı şemasını oluşturacağız.

Migration Dosyalarını Oluşturma
Bu komut, `CompanyAppDbContext` ve bağlı `Entity` sınıflarını tarayarak veritabanı şemasını oluşturacak/güncelleyecek C# kodlarını hazırlar.
`dotnet ef migrations add InitialCreate`

1. Bu komuttan sonra projenizde `Migrations` klasörü otomatik olarak oluşacaktır.

Veritabanını Fiziksel Olarak Oluşturma
Bu komut, bir önceki adımda oluşturulan migration dosyasını çalıştırarak `Company.db` veritabanı dosyasını proje dizininizde fiziksel olarak oluşturur.
`dotnet ef database update`

2.

Adım 5: Program.cs - Tüm Görevlerin Uygulanması

Son olarak, `Program.cs` dosyasını açıp ödevdeki tüm görevleri yerine getirecek şekilde aşağıdaki kodlarla güncelleyin.

Program.cs

```
using System;
using System.Linq;
using CompanyManagementSystem.Data;
using CompanyManagementSystem.Entities;

namespace CompanyManagementSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            // Görev 3: Veritabanına başlangıç verilerini ekle
            AddInitialData();

            Console.WriteLine("----------------------------------");

            // Görev 4: Veri Sorgulama (Filtreleme)
            // Maaşı 20000'den yüksek olan Software departmanı çalışanlarını
listeleyelim.
            QuerySoftwareEmployees(20000);

            Console.WriteLine("----------------------------------");

            // Görev 5: Veri Silme
```

```
            // ID'si 5 olan çalışanı (Ayşe Yılmaz) işten çıkaralım.
            DeleteEmployee(5);
        }

        public static void AddInitialData()
        {
            using (var context = new CompanyAppDbContext())
            {
                if (context.Departments.Any())
                {
                    Console.WriteLine("Veritabanı zaten örnek verileri içeriyor. Ekleme işlemi atlandı.");
                    return;
                }

                Console.WriteLine("Departmanlar ve çalışanlar veritabanına ekleniyor...");

                var softwareDept = new Department { Name = "Software" };
                var accountingDept = new Department { Name = "Accounting" };
                var hrDept = new Department { Name = "Human Resources" };

                softwareDept.Employees.Add(new Employee { FullName = "Ahmet Çelik", Position = "Senior Developer", Salary = 25000 });
                softwareDept.Employees.Add(new Employee { FullName = "Zeynep Kaya", Position = "Junior Developer", Salary = 16000 });
                softwareDept.Employees.Add(new Employee { FullName = "Mustafa Doğan", Position = "Team Lead", Salary = 32000 });

                accountingDept.Employees.Add(new Employee { FullName = "Elif Arslan", Position = "Accountant", Salary = 18000 });
                accountingDept.Employees.Add(new Employee { FullName = "Ayşe Yılmaz", Position = "Senior Accountant", Salary = 22000 });

                hrDept.Employees.Add(new Employee { FullName = "Fatma Şahin", Position = "HR Specialist", Salary = 19000 });
                hrDept.Employees.Add(new Employee { FullName = "Hasan Vural", Position = "Recruitment Manager", Salary = 26000 });

                context.Departments.AddRange(softwareDept, accountingDept, hrDept);
                context.SaveChanges();

                Console.WriteLine("Veriler başarıyla eklendi!");
            }
        }

        public static void QuerySoftwareEmployees(decimal salaryThreshold)
        {
            using (var context = new CompanyAppDbContext())
            {
                Console.WriteLine($"'Software' departmanında maaşı {salaryThreshold:C} üzerinde olan çalışanlar:");
```

```
                // LINQ kullanarak hem departman adına hem de maaşa göre
filtreleme yapıyoruz.
                // EF Core, navigation property (e.Department.Name) üzerinden
yapılan bu sorguyu
                // arka planda verimli bir SQL JOIN'ine dönüştürür.
                var highSalarySoftwareEmployees = context.Employees
                    .Where(e => e.Department.Name == "Software" && e.Salary >
salaryThreshold)
                    .ToList();

                if (!highSalarySoftwareEmployees.Any())
                {
                    Console.WriteLine("Belirtilen kriterlere uygun çalışan
bulunamadı.");
                    return;
                }

                foreach (var employee in highSalarySoftwareEmployees)
                {
                    Console.WriteLine($"- Ad: {employee.FullName}, Pozisyon:
{employee.Position}, Maaş: {employee.Salary:C}");
                }
            }
        }

        public static void DeleteEmployee(int employeeId)
        {
            using (var context = new CompanyAppDbContext())
            {
                Console.WriteLine($"ID'si {employeeId} olan çalışan
siliniyor...");

                var employeeToDelete = context.Employees.FirstOrDefault(e =>
e.Id == employeeId);

                if (employeeToDelete != null)
                {
                    context.Employees.Remove(employeeToDelete);
                    context.SaveChanges();
                    Console.WriteLine($"'{employeeToDelete.FullName}' adlı
çalışan veritabanından başarıyla silindi.");

                    // Silme işleminin kanıtı olarak, silinen çalışanı tekrar
sorgulayalım
                    var deletedEmployeeCheck =
context.Employees.FirstOrDefault(e => e.Id == employeeId);
                    if (deletedEmployeeCheck == null)
                    {
                        Console.WriteLine($"Kontrol: ID'si {employeeId} olan
çalışan artık bulunamıyor.");
                    }
                }
                else
                {
```

```
                    Console.WriteLine($"Hata: ID'si {employeeId} olan bir
çalışan bulunamadı.");
                }
            }
        }
    }
}
```

Ödev 3: Müşteri Sipariş Sistemi - Çözüm

Proje Dizin Yapısı

Projenizi oluşturduktan sonra, kodlarınızı düzenlemek için ana dizin içinde `Entities` ve `Data` adında iki klasör oluşturmalısınız. İşlemler bittiğinde proje yapınız aşağıdaki gibi olacaktır:

```
ECommerceSystem/
|
|-- Data/
|   |-- ECommerceAppDbContext.cs
|
|-- Entities/
|   |-- Customer.cs
|   |-- Order.cs
|
|-- Migrations/
|   |-- ... (EF Core tarafından otomatik oluşturulur)
|
|-- Program.cs
|-- ECommerceSystem.csproj
|-- ECommerce.db  (update-database komutundan sonra oluşur)
```

Adım 1: Proje Oluşturma ve Gerekli Paketlerin Yüklenmesi

Yeni Proje Oluşturma
Terminalinizi açın ve .NET CLI kullanarak klasik `Main` metodu yapısına sahip yeni bir konsol projesi oluşturun.
`dotnet new console -n ECommerceSystem --use-program-main`

1.
2. **Gerekli EF Core Paketlerini Ekleme**
   Terminalinizde, yeni oluşturulan `ECommerceSystem` proje klasörünün içine gidin ve aşağıdaki komutları sırasıyla çalıştırın.

Veritabanı sağlayıcısı olarak SQLite kullanacağımız için bu paketi ekliyoruz:
`dotnet add package Microsoft.EntityFrameworkCore.Sqlite`
    - o

Migration gibi `dotnet ef` komutlarını terminalden çalıştırabilmek için bu aracı ekliyoruz:
`dotnet add package Microsoft.EntityFrameworkCore.Design`
    - o

Adım 2: Entity Sınıflarının Oluşturulması

Projenizde oluşturduğunuz `Entities` klasörü içine `Customer.cs` ve `Order.cs` dosyalarını ekleyin.

`Entities/Customer.cs`

```
using System.Collections.Generic;

namespace ECommerceSystem.Entities
```

```
{
    public class Customer
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }

        // Navigation property: Bir müşterinin birden fazla siparişi olabilir.
        public ICollection<Order> Orders { get; set; } = [];
    }
}
```

Entities/Order.cs

```
using System;

namespace ECommerceSystem.Entities
{
    public class Order
    {
        public int Id { get; set; }
        public DateTime OrderDate { get; set; }
        public decimal TotalAmount { get; set; }

        // Foreign key property
        public int CustomerId { get; set; }

        // Navigation property: Her sipariş bir müşteriye aittir.
        public Customer Customer { get; set; }
    }
}
```

Adım 3: DbContext Sınıfının Oluşturulması

`Data` klasörü içine `ECommerceAppDbContext.cs` adında bir dosya oluşturun.

Data/ECommerceAppDbContext.cs

```
using ECommerceSystem.Entities;
using Microsoft.EntityFrameworkCore;

namespace ECommerceSystem.Data
{
    public class ECommerceAppDbContext : DbContext
    {
        public DbSet<Customer> Customers { get; set; }
        public DbSet<Order> Orders { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
```

```
        {
            // Veritabanı dosyası olarak proje ana dizininde "ECommerce.db"
adında bir dosya oluşturulacak.
            optionsBuilder.UseSqlite("Data Source=ECommerce.db");
        }
    }
}
```

Adım 4: Veritabanını Oluşturma (Migration ve Update)

Şimdi terminal üzerinden veritabanı şemasını oluşturacağız.

Migration Dosyalarını Oluşturma
Bu komut, `ECommerceAppDbContext` ve bağlı `Entity` sınıflarını tarayarak veritabanı şemasını oluşturacak/güncelleyecek C# kodlarını hazırlar.
`dotnet ef migrations add InitialCreate`

1. Bu komuttan sonra projenizde `Migrations` klasörü otomatik olarak oluşacaktır.

Veritabanını Fiziksel Olarak Oluşturma
Bu komut, bir önceki adımda oluşturulan migration dosyasını çalıştırarak `ECommerce.db` veritabanı dosyasını proje dizininizde fiziksel olarak oluşturur.
`dotnet ef database update`

2.

Adım 5: Program.cs - Tüm Görevlerin Uygulanması

Son olarak, `Program.cs` dosyasını açıp ödevdeki tüm görevleri yerine getirecek şekilde aşağıdaki kodlarla güncelleyin.

Program.cs

```
using System;
using System.Linq;
using ECommerceSystem.Data;
using ECommerceSystem.Entities;
using Microsoft.EntityFrameworkCore;

namespace ECommerceSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            // Görev 3: Veritabanına başlangıç verilerini ekle
            AddInitialData();

            Console.WriteLine("-----------------------------------");

            // Görev 4: İlişkili Veri Çekme
            // ID'si 1 olan müşterinin tüm siparişlerini en yeniden en eskiye
doğru listeleyelim.
            GetCustomerOrders(1);
```

```
            Console.WriteLine("-----------------------------------");

            // Görev 5: Veri Güncelleme
            // ID'si 2 olan müşterinin email adresini güncelleyelim.
            UpdateCustomerEmail(2, "y.kaya.new@example.com");
        }

        public static void AddInitialData()
        {
            using (var context = new ECommerceAppDbContext())
            {
                if (context.Customers.Any())
                {
                    Console.WriteLine("Veritabanı zaten örnek verileri içeriyor.
Ekleme işlemi atlandı.");
                    return;
                }

                Console.WriteLine("Müşteriler ve siparişler veritabanına
ekleniyor...");

                var customer1 = new Customer { FirstName = "Ali", LastName =
"Veli", Email = "ali.veli@example.com" };
                var customer2 = new Customer { FirstName = "Yıldız", LastName =
"Kaya", Email = "yildiz.kaya@example.com" };

                // Müşteri 1 için siparişler
                customer1.Orders.Add(new Order { OrderDate = new DateTime(2024,
10, 5), TotalAmount = 150.75m });
                customer1.Orders.Add(new Order { OrderDate = new DateTime(2025,
1, 20), TotalAmount = 89.90m });
                customer1.Orders.Add(new Order { OrderDate = new DateTime(2025,
3, 15), TotalAmount = 250.00m });
                customer1.Orders.Add(new Order { OrderDate = new DateTime(2024,
5, 12), TotalAmount = 45.50m });

                // Müşteri 2 için siparişler
                customer2.Orders.Add(new Order { OrderDate = new DateTime(2025,
2, 1), TotalAmount = 1200.00m });
                customer2.Orders.Add(new Order { OrderDate = new DateTime(2025,
2, 28), TotalAmount = 78.25m });
                customer2.Orders.Add(new Order { OrderDate = new DateTime(2024,
12, 18), TotalAmount = 315.40m });

                context.Customers.AddRange(customer1, customer2);
                context.SaveChanges();

                Console.WriteLine("Veriler başarıyla eklendi!");
            }
        }

        public static void GetCustomerOrders(int customerId)
        {
            using (var context = new ECommerceAppDbContext())
```

```
            {
                // Müşteriyi bulurken ilişkili olduğu siparişleri de (Orders) veritabanından getirmek için
                // Include() metodunu kullanıyoruz. Bu, N+1 problemini önler ve verimli bir sorgu oluşturur.
                var customer = context.Customers
                    .Include(c => c.Orders)
                    .FirstOrDefault(c => c.Id == customerId);

                if (customer != null)
                {
                    Console.WriteLine($"--- Müşteri: {customer.FirstName} {customer.LastName} ---");
                    Console.WriteLine("Siparişleri (En yeniden en eskiye):");

                    // Müşterinin siparişlerini OrderByDescending ile tarihe göre tersten sıralıyoruz.
                    var sortedOrders = customer.Orders.OrderByDescending(o => o.OrderDate);

                    if (!sortedOrders.Any())
                    {
                        Console.WriteLine("Bu müşterinin kayıtlı siparişi bulunmamaktadır.");
                        return;
                    }

                    foreach (var order in sortedOrders)
                    {
                        Console.WriteLine($"  - Sipariş ID: {order.Id}, Tarih: {order.OrderDate:dd.MM.yyyy}, Tutar: {order.TotalAmount:C}");
                    }
                }
                else
                {
                    Console.WriteLine($"Hata: ID'si {customerId} olan bir müşteri bulunamadı.");
                }
            }
        }

        public static void UpdateCustomerEmail(int customerId, string newEmail)
        {
            using (var context = new ECommerceAppDbContext())
            {
                Console.WriteLine($"ID'si {customerId} olan müşterinin e-posta adresi güncelleniyor...");

                var customerToUpdate = context.Customers.FirstOrDefault(c => c.Id == customerId);

                if (customerToUpdate != null)
                {
                    string oldEmail = customerToUpdate.Email;
                    customerToUpdate.Email = newEmail;
```

```
                    context.SaveChanges();

                    Console.WriteLine("Güncelleme başarılı!");
                    Console.WriteLine($"Eski E-posta: {oldEmail}");
                    Console.WriteLine($"Yeni E-posta:
{customerToUpdate.Email}");
                }
                else
                {
                    Console.WriteLine($"Hata: ID'si {customerId} olan bir
müşteri bulunamadı.");
                }
            }
        }
    }
}
```

Ödev 4: Film ve Kategori Sistemi - Çözüm

Proje Dizin Yapısı

Projenizi oluşturduktan sonra, kodlarınızı düzenli tutmak için ana dizin içinde `Entities` ve `Data` adında iki klasör oluşturmalısınız. İşlemler tamamlandığında proje yapınız aşağıdaki gibi olacaktır:

```
MovieDatabaseSystem/
|
|-- Data/
|   |-- MovieAppDbContext.cs
|
|-- Entities/
|   |-- Category.cs
|   |-- Movie.cs
|
|-- Migrations/
|   |-- ... (EF Core tarafından otomatik oluşturulur)
|
|-- Program.cs
|-- MovieDatabaseSystem.csproj
|-- Movie.db  (update-database komutundan sonra oluşur)
```

Adım 1: Proje Oluşturma ve Gerekli Paketlerin Yüklenmesi

Yeni Proje Oluşturma
Terminalinizi açın ve .NET CLI kullanarak klasik `Main` metodu yapısına sahip yeni bir konsol projesi oluşturun.

```
dotnet new console -n MovieDatabaseSystem --use-program-main
```

1.
2. **Gerekli EF Core Paketlerini Ekleme**
   Terminalinizde, yeni oluşturulan `MovieDatabaseSystem` proje klasörünün içine gidin ve aşağıdaki komutları sırasıyla çalıştırın.

Veritabanı sağlayıcısı olarak SQLite kullanacağımız için bu paketi ekliyoruz:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

-

Migration gibi `dotnet ef` komutlarını terminalden çalıştırabilmek için bu aracı ekliyoruz:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

-

Adım 2: Entity Sınıflarının Oluşturulması

Projenizde oluşturduğunuz `Entities` klasörü içine `Category.cs` ve `Movie.cs` dosyalarını ekleyin.

`Entities/Category.cs`

```
using System.Collections.Generic;

namespace MovieDatabaseSystem.Entities
```

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

        // Navigation property: Bir kategoride birden fazla film olabilir.
        public ICollection<Movie> Movies { get; set; } = [];
    }
}
```

Entities/Movie.cs

```
namespace MovieDatabaseSystem.Entities
{
    public class Movie
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Director { get; set; }
        public int ReleaseYear { get; set; }

        // Foreign key property
        public int CategoryId { get; set; }

        // Navigation property: Her film bir kategoriye aittir.
        public Category Category { get; set; }
    }
}
```

Adım 3: DbContext Sınıfının Oluşturulması

`Data` klasörü içine `MovieAppDbContext.cs` adında bir dosya oluşturun.

Data/MovieAppDbContext.cs

```
using Microsoft.EntityFrameworkCore;
using MovieDatabaseSystem.Entities;

namespace MovieDatabaseSystem.Data
{
    public class MovieAppDbContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Movie> Movies { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            // Veritabanı dosyası olarak proje ana dizininde "Movie.db" adında
bir dosya oluşturulacak.
```

```
            optionsBuilder.UseSqlite("Data Source=Movie.db");
        }
    }
}
```

Adım 4: Veritabanını Oluşturma (Migration ve Update)

Şimdi terminal üzerinden veritabanı şemasını oluşturacağız.

Migration Dosyalarını Oluşturma
Bu komut, `MovieAppDbContext` ve bağlı `Entity` sınıflarını tarayarak veritabanı şemasını oluşturacak/güncelleyecek C# kodlarını hazırlar.
`dotnet ef migrations add InitialCreate`

1. Bu komuttan sonra projenizde `Migrations` klasörü otomatik olarak oluşacaktır.

Veritabanını Fiziksel Olarak Oluşturma
Bu komut, bir önceki adımda oluşturulan migration dosyasını çalıştırarak `Movie.db` veritabanı dosyasını proje dizininizde fiziksel olarak oluşturur.
`dotnet ef database update`

2.

Adım 5: Program.cs - Tüm Görevlerin Uygulanması

Son olarak, `Program.cs` dosyasını açıp ödevdeki tüm görevleri yerine getirecek şekilde aşağıdaki kodlarla güncelleyin.

Program.cs

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using MovieDatabaseSystem.Data;
using MovieDatabaseSystem.Entities;

namespace MovieDatabaseSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            // Görev 3: Veritabanına başlangıç verilerini ekle
            AddInitialData();

            Console.WriteLine("----------------------------------");

            // Görev 4: Veri Listeleme (Projeksiyon)
            ListMoviesWithCategory();

            Console.WriteLine("----------------------------------");

            // Görev 5: Veri Silme (İlişkili Verilerle Birlikte)
            DeleteCategoryAndMovies("Comedy");
```

```
        }

        public static void AddInitialData()
        {
            using (var context = new MovieAppDbContext())
            {
                if (context.Categories.Any())
                {
                    Console.WriteLine("Veritabanı zaten örnek verileri içeriyor. Ekleme işlemi atlandı.");
                    return;
                }

                Console.WriteLine("Kategoriler ve filmler veritabanına ekleniyor...");

                var catSciFi = new Category { Name = "Science Fiction" };
                var catDrama = new Category { Name = "Drama" };
                var catComedy = new Category { Name = "Comedy" };
                var catAction = new Category { Name = "Action" };

                catSciFi.Movies.Add(new Movie { Title = "Inception", Director = "Christopher Nolan", ReleaseYear = 2010 });
                catSciFi.Movies.Add(new Movie { Title = "The Matrix", Director = "Wachowskis", ReleaseYear = 1999 });
                catSciFi.Movies.Add(new Movie { Title = "Blade Runner 2049", Director = "Denis Villeneuve", ReleaseYear = 2017 });

                catDrama.Movies.Add(new Movie { Title = "The Shawshank Redemption", Director = "Frank Darabont", ReleaseYear = 1994 });
                catDrama.Movies.Add(new Movie { Title = "Forrest Gump", Director = "Robert Zemeckis", ReleaseYear = 1994 });

                catComedy.Movies.Add(new Movie { Title = "Superbad", Director = "Greg Mottola", ReleaseYear = 2007 });
                catComedy.Movies.Add(new Movie { Title = "The Hangover", Director = "Todd Phillips", ReleaseYear = 2009 });

                catAction.Movies.Add(new Movie { Title = "The Dark Knight", Director = "Christopher Nolan", ReleaseYear = 2008 });

                context.Categories.AddRange(catSciFi, catDrama, catComedy, catAction);
                context.SaveChanges();

                Console.WriteLine("Veriler başarıyla eklendi!");
            }
        }

        public static void ListMoviesWithCategory()
        {
            using (var context = new MovieAppDbContext())
            {
                Console.WriteLine("--- Filmler (Kategori Adlarıyla Birlikte) ---");
```

```
                // İlişkili kategori verisini getirmek için Include() kullanıyoruz.
                // Sadece istediğimiz alanları (Title, Director, CategoryName) seçmek için Select() ile projeksiyon yapıyoruz.
                var moviesWithCategory = context.Movies
                    .Include(m => m.Category)
                    .Select(m => new
                    {
                        MovieTitle = m.Title,
                        DirectorName = m.Director,
                        CategoryName = m.Category.Name
                    })
                    .ToList();

                foreach (var item in moviesWithCategory)
                {
                    Console.WriteLine($"Film: {item.MovieTitle,-30} | Yönetmen: {item.DirectorName,-20} | Kategori: {item.CategoryName}");
                }
            }
        }

        public static void DeleteCategoryAndMovies(string categoryName)
        {
            using (var context = new MovieAppDbContext())
            {
                Console.WriteLine($"'{categoryName}' kategorisi ve bu kategoriye ait tüm filmler siliniyor...");

                // Silinecek kategoriyi bul. EF Core'un ilişkili filmleri de silmesi gerektiğini anlaması için
                // bu filmleri de sorguya dahil etmemiz GEREKMEZ, çünkü veritabanı seviyesinde
                // "cascade delete" (zincirleme silme) kuralı varsayılan olarak EF Core tarafından ayarlanır.
                var categoryToDelete = context.Categories
                    .FirstOrDefault(c => c.Name == categoryName);

                if (categoryToDelete != null)
                {
                    // Sadece kategoriyi silme komutu vermek yeterlidir.
                    context.Categories.Remove(categoryToDelete);
                    context.SaveChanges();

                    Console.WriteLine($"'{categoryName}' kategorisi ve ilişkili filmler başarıyla silindi.");

                    // Doğrulama yapalım:
                    var isCategoryDeleted = !context.Categories.Any(c => c.Name == categoryName);
                    var areMoviesDeleted = !context.Movies.Any(m => m.Category.Name == categoryName);

                    if (isCategoryDeleted && areMoviesDeleted)
```

```
                    {
                        Console.WriteLine("Doğrulama başarılı: Kategori ve filmleri artık veritabanında bulunmuyor.");
                    }
                }
                else
                {
                    Console.WriteLine($"Hata: '{categoryName}' adında bir kategori bulunamadı.");
                }
            }
        }
    }
}
```

Ödev 5: Blog Sistemi - Çözüm

Proje Dizin Yapısı

Projenizi oluşturduktan sonra, kodlarınızı düzenli tutmak için ana dizin içinde `Entities` ve `Data` adında iki klasör oluşturmalısınız. İşlemler tamamlandığında proje yapınız aşağıdaki gibi olacaktır:

```
BlogSystem/
|
|-- Data/
|   |-- BlogAppDbContext.cs
|
|-- Entities/
|   |-- Category.cs
|   |-- Post.cs
|   |-- Comment.cs
|
|-- Migrations/
|   |-- ... (EF Core tarafından otomatik oluşturulur)
|
|-- Program.cs
|-- BlogSystem.csproj
|-- Blog.db  (update-database komutundan sonra oluşur)
```

Adım 1: Proje Oluşturma ve Gerekli Paketlerin Yüklenmesi

Yeni Proje Oluşturma
Terminalinizi açın ve .NET CLI kullanarak klasik `Main` metodu yapısına sahip yeni bir konsol projesi oluşturun.
`dotnet new console -n BlogSystem --use-program-main`

1.
2. **Gerekli EF Core Paketlerini Ekleme**
   Terminalinizde, yeni oluşturulan `BlogSystem` proje klasörünün içine gidin ve aşağıdaki komutları sırasıyla çalıştırın.

Veritabanı sağlayıcısı olarak SQLite kullanacağımız için bu paketi ekliyoruz:
`dotnet add package Microsoft.EntityFrameworkCore.Sqlite`
- o

Migration gibi `dotnet ef` komutlarını terminalden çalıştırabilmek için bu aracı ekliyoruz:
`dotnet add package Microsoft.EntityFrameworkCore.Design`
- o

Adım 2: Entity Sınıflarının Oluşturulması

Projenizde oluşturduğunuz `Entities` klasörü içine `Category.cs`, `Post.cs` ve `Comment.cs` dosyalarını ekleyin.

`Entities/Category.cs`

```
using System.Collections.Generic;
```

```
namespace BlogSystem.Entities
{
    public class Category
    {
        public int Id { get; set; }
        public string Name { get; set; }

        // Navigation property: Bir kategorinin birden fazla post'u olabilir.
        public ICollection<Post> Posts { get; set; } = [];
    }
}
```

Entities/Post.cs

```
using System;
using System.Collections.Generic;

namespace BlogSystem.Entities
{
    public class Post
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }
        public DateTime PublishedDate { get; set; }

        // Kategori ile ilişki (Bire-Çok)
        public int CategoryId { get; set; }
        public Category Category { get; set; }

        // Yorumlar ile ilişki (Bire-Çok)
        public ICollection<Comment> Comments { get; set; } = [];
    }
}
```

Entities/Comment.cs

```
namespace BlogSystem.Entities
{
    public class Comment
    {
        public int Id { get; set; }
        public string AuthorName { get; set; }
        public string Message { get; set; }

        // Post ile ilişki (Bire-Çok)
        public int PostId { get; set; }
        public Post Post { get; set; }
    }
}
```

Adım 3: DbContext Sınıfının Oluşturulması

`Data` klasörü içine `BlogAppDbContext.cs` adında bir dosya oluşturun.

`Data/BlogAppDbContext.cs`

```
using BlogSystem.Entities;
using Microsoft.EntityFrameworkCore;

namespace BlogSystem.Data
{
    public class BlogAppDbContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Post> Posts { get; set; }
        public DbSet<Comment> Comments { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            // Veritabanı dosyası olarak proje ana dizininde "Blog.db" adında
bir dosya oluşturulacak.
            optionsBuilder.UseSqlite("Data Source=Blog.db");
        }
    }
}
```

Adım 4: Veritabanını Oluşturma (Migration ve Update)

Şimdi terminal üzerinden veritabanı şemasını oluşturacağız.

Migration Dosyalarını Oluşturma
Bu komut, `BlogAppDbContext` ve bağlı `Entity` sınıflarını tarayarak veritabanı şemasını oluşturacak/güncelleyecek C# kodlarını hazırlar.
`dotnet ef migrations add InitialCreate`

1. Bu komuttan sonra projenizde `Migrations` klasörü otomatik olarak oluşacaktır.

Veritabanını Fiziksel Olarak Oluşturma
Bu komut, bir önceki adımda oluşturulan migration dosyasını çalıştırarak `Blog.db` veritabanı dosyasını proje dizininizde fiziksel olarak oluşturur.
`dotnet ef database update`

2.

Adım 5: Program.cs - Tüm Görevlerin Uygulanması

Son olarak, `Program.cs` dosyasını açıp ödevdeki tüm görevleri yerine getirecek şekilde aşağıdaki kodlarla güncelleyin.

`Program.cs`

```
using System;
using System.Linq;
using BlogSystem.Data;
using BlogSystem.Entities;
using Microsoft.EntityFrameworkCore;

namespace BlogSystem
{
    class Program
    {
        static void Main(string[] args)
        {
            // Görev 3: Veritabanına başlangıç verilerini ekle
            AddInitialData();

            Console.WriteLine("-----------------------------------");

            // Görev 4: İleri Düzey İlişkili Veri Çekme
            // ID'si 1 olan 'Teknoloji' kategorisindeki postları ve yorumlarını hiyerarşik olarak getirelim.
            GetCategoryWithPostsAndComments(1);

            Console.WriteLine("-----------------------------------");

            // Görev 5: Veri Güncelleme
            // ID'si 3 olan yorumun mesajını güncelleyelim.
            UpdateCommentMessage(3, "Harika bir ekleme, bunu deneyeceğim!");
        }

        public static void AddInitialData()
        {
            using (var context = new BlogAppDbContext())
            {
                if (context.Categories.Any())
                {
                    Console.WriteLine("Veritabanı zaten örnek verileri içeriyor. Ekleme işlemi atlandı.");
                    return;
                }

                Console.WriteLine("Kategoriler, postlar ve yorumlar veritabanına ekleniyor...");

                var catTech = new Category { Name = "Teknoloji" };
                var catTravel = new Category { Name = "Seyahat" };

                // Teknoloji postları ve yorumları
                var post1 = new Post { Title = ".NET 9'daki Yenilikler", Content = "...", PublishedDate = DateTime.Now.AddDays(-10) };
                post1.Comments.Add(new Comment { AuthorName = "Ayşe", Message = "Çok bilgilendirici bir yazı." });
                post1.Comments.Add(new Comment { AuthorName = "Fatma", Message = "Teşekkürler!" });
                catTech.Posts.Add(post1);
```

```
                var post2 = new Post { Title = "Entity Framework Core Performans İpuçları", Content = "...", PublishedDate = DateTime.Now.AddDays(-5) };
                post2.Comments.Add(new Comment { AuthorName = "Mehmet", Message = "Bu ipuçları çok işime yaradı." });
                catTech.Posts.Add(post2);

                // Seyahat postları ve yorumları
                var post3 = new Post { Title = "Japonya'da Kiraz Çiçeği Mevsimi", Content = "...", PublishedDate = DateTime.Now.AddDays(-20) };
                post3.Comments.Add(new Comment { AuthorName = "Zeynep", Message = "Hayalimdeki seyahat!" });
                post3.Comments.Add(new Comment { AuthorName = "Ahmet", Message = "Fotoğraflar harika görünüyor." });
                post3.Comments.Add(new Comment { AuthorName = "Hasan", Message = "Gidilecek yerler listeme ekledim." });
                catTravel.Posts.Add(post3);

                context.Categories.AddRange(catTech, catTravel);
                context.SaveChanges();

                Console.WriteLine("Veriler başarıyla eklendi!");
            }
        }

        public static void GetCategoryWithPostsAndComments(int categoryId)
        {
            using (var context = new BlogAppDbContext())
            {
                // Bir kategoriyi çekerken, ilişkili olduğu Post'ları da getirmek için Include() kullanırız.
                // Gelen Post'ların da ilişkili olduğu Comment'leri getirmek için ThenInclude() kullanırız.
                // Bu sayede tek bir veritabanı sorgusu ile tüm hiyerarşik veriyi çekmiş oluruz.
                var category = context.Categories
                    .Include(c => c.Posts)
                        .ThenInclude(p => p.Comments)
                    .FirstOrDefault(c => c.Id == categoryId);

                if (category != null)
                {
                    Console.WriteLine($"Kategori: {category.Name}");
                    Console.WriteLine("==============================");

                    foreach (var post in category.Posts)
                    {
                        Console.WriteLine($"  -> Post: '{post.Title}'");
                        if (post.Comments.Any())
                        {
                            foreach (var comment in post.Comments)
                            {
                                Console.WriteLine($"    - Yorum (ID: {comment.Id}): \"{comment.Message}\" (Yazan: {comment.AuthorName})");
                            }
                        }
```

```
                        else
                        {
                            Console.WriteLine("    (Bu posta henüz yorum
yapılmamış.)");
                        }
                    }
                }
                else
                {
                    Console.WriteLine($"Hata: ID'si {categoryId} olan bir
kategori bulunamadı.");
                }
            }
        }

        public static void UpdateCommentMessage(int commentId, string
newMessage)
        {
            using (var context = new BlogAppDbContext())
            {
                Console.WriteLine($"ID'si {commentId} olan yorum
güncelleniyor...");

                var commentToUpdate = context.Comments.FirstOrDefault(c => c.Id
== commentId);

                if (commentToUpdate != null)
                {
                    string oldMessage = commentToUpdate.Message;
                    commentToUpdate.Message = newMessage;
                    context.SaveChanges();

                    Console.WriteLine("Güncelleme başarılı!");
                    Console.WriteLine($"Eski Mesaj: \"{oldMessage}\"");
                    Console.WriteLine($"Yeni Mesaj:
\"{commentToUpdate.Message}\"");
                }
                else
                {
                    Console.WriteLine($"Hata: ID'si {commentId} olan bir yorum
bulunamadı.");
                }
            }
        }
    }
}
```