Database Testing Lab Exercise

Overview

In this lab, you will apply the concepts from our database testing tutorial to develop and test a simple order management system using Entity Framework Core and .NET. You will write both database models and corresponding tests to ensure data integrity and proper functionality.

Prerequisites

- Visual Studio 2022 or VS Code
- .NET 7.0 or later
- SQL Server (LocalDB is sufficient)
- Basic knowledge of C#, EF Core, and SQL

Part 1: Setup (30 minutes)

1.1 Create the Solution Structure

1. Create a new solution named `OrderManager`
2. Add two projects:
    - `OrderManager.Core` (.NET Class Library)
    - `OrderManager.Tests` (xUnit Test Project)
3. Add the following NuGet packages:

    For OrderManager.Core:

    ```
    Microsoft.EntityFrameworkCore.SqlServer
    Microsoft.EntityFrameworkCore.Design
    ```

    For OrderManager.Tests:

    ```
    Microsoft.EntityFrameworkCore.InMemory
    Microsoft.NET.Test.Sdk
    xunit
    xunit.runner.visualstudio
    ```

4. Add a reference from `OrderManager.Tests` to `OrderManager.Core`

1.2 Create the Data Models

Create the following classes in the `OrderManager.Core/Models` folder:

```
// Customer.cs
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public bool IsActive { get; set; }
    public DateTime CreatedDate { get; set; }

    public ICollection<Order> Orders { get; set; }
}

// Order.cs
public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
    public DateTime OrderDate { get; set; }
    public string Status { get; set; }
    public decimal TotalAmount { get; set; }

    public Customer Customer { get; set; }
    public ICollection<OrderItem> Items { get; set; }
}

// OrderItem.cs
public class OrderItem
{
    public int Id { get; set; }
    public int OrderId { get; set; }
    public string ProductName { get; set; }
    public decimal UnitPrice { get; set; }
    public int Quantity { get; set; }

    public Order Order { get; set; }
}
```

1.3 Create the Database Context

Create `OrderContext.cs` in the `OrderManager.Core/Data` folder:

```
using Microsoft.EntityFrameworkCore;
using OrderManager.Core.Models;

namespace OrderManager.Core.Data
{
    public class OrderContext : DbContext
    {
        public OrderContext(DbContextOptions<OrderContext> options)
            : base(options)
        {
```

```
        }

        public DbSet<Customer> Customers { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // TODO: Configure the model relationships and constraints

            // Example:
            modelBuilder.Entity<Customer>()
                .HasIndex(c => c.Email)
                .IsUnique();

            modelBuilder.Entity<Order>()
                .HasOne(o => o.Customer)
                .WithMany(c => c.Orders)
                .HasForeignKey(o => o.CustomerId);

            modelBuilder.Entity<OrderItem>()
                .HasOne(oi => oi.Order)
                .WithMany(o => o.Items)
                .HasForeignKey(oi => oi.OrderId);
        }
    }
}
```

Part 2: Service Implementation (30 minutes)

2.1 Create Order Service Interface

Create `IOrderService.cs` in the `OrderManager.Core/Services` folder:

```
using OrderManager.Core.Models;

namespace OrderManager.Core.Services
{
    public interface IOrderService
    {
        Task<Customer> GetCustomerByIdAsync(int id);
        Task<Customer> CreateCustomerAsync(Customer customer);
        Task<Order> CreateOrderAsync(Order order, List<OrderItem> items);
        Task<decimal> CalculateOrderTotalAsync(int orderId);
        Task<IEnumerable<Order>> GetCustomerOrdersAsync(int customerId);
        Task<bool> UpdateOrderStatusAsync(int orderId, string status);
    }
}
```

2.2 Implement the Order Service

Create `OrderService.cs` in the `OrderManager.Core/Services` folder:

```
using Microsoft.EntityFrameworkCore;
using OrderManager.Core.Data;
using OrderManager.Core.Models;

namespace OrderManager.Core.Services
{
    public class OrderService : IOrderService
    {
        private readonly OrderContext _context;

        public OrderService(OrderContext context)
        {
            _context = context;
        }

        public async Task<Customer> GetCustomerByIdAsync(int id)
        {
            return await _context.Customers.FindAsync(id);
        }

        public async Task<Customer> CreateCustomerAsync(Customer customer)
        {
            customer.CreatedDate = DateTime.Now;
            customer.IsActive = true;

            _context.Customers.Add(customer);
            await _context.SaveChangesAsync();

            return customer;
        }

        public async Task<Order> CreateOrderAsync(Order order, List<OrderItem>
items)
        {
            // Start a transaction
            using var transaction = await
_context.Database.BeginTransactionAsync();

            try
            {
                // Add the order
                order.OrderDate = DateTime.Now;
                order.Status = "New";
                _context.Orders.Add(order);
                await _context.SaveChangesAsync();

                // Add order items and link to the order
                foreach (var item in items)
                {
                    item.OrderId = order.Id;
                    _context.OrderItems.Add(item);
```

```
                }

                await _context.SaveChangesAsync();

                // Calculate total amount
                order.TotalAmount = items.Sum(i => i.UnitPrice * i.Quantity);
                await _context.SaveChangesAsync();

                await transaction.CommitAsync();
                return order;
            }
            catch
            {
                await transaction.RollbackAsync();
                throw;
            }
        }

        public async Task<decimal> CalculateOrderTotalAsync(int orderId)
        {
            return await _context.OrderItems
                .Where(oi => oi.OrderId == orderId)
                .SumAsync(oi => oi.UnitPrice * oi.Quantity);
        }

        public async Task<IEnumerable<Order>> GetCustomerOrdersAsync(int
customerId)
        {
            return await _context.Orders
                .Where(o => o.CustomerId == customerId)
                .Include(o => o.Items)
                .ToListAsync();
        }

        public async Task<bool> UpdateOrderStatusAsync(int orderId, string status)
        {
            var order = await _context.Orders.FindAsync(orderId);
            if (order == null)
                return false;

            order.Status = status;
            await _context.SaveChangesAsync();
            return true;
        }
    }
}
```

Part 3: Testing (60 minutes)

3.1 Create Test Base Class

Create `TestBase.cs` in the `OrderManager.Tests` folder:

```
using Microsoft.EntityFrameworkCore;
using OrderManager.Core.Data;
using System;

namespace OrderManager.Tests
{
    public abstract class TestBase
    {
        protected OrderContext CreateInMemoryContext()
        {
            var options = new DbContextOptionsBuilder<OrderContext>()
                .UseInMemoryDatabase(databaseName: Guid.NewGuid().ToString())
                .Options;

            var context = new OrderContext(options);
            context.Database.EnsureCreated();
            return context;
        }

        protected OrderContext CreateSqlServerContext()
        {
            var options = new DbContextOptionsBuilder<OrderContext>()
                .UseSqlServer("Server=
(localdb)\\mssqllocaldb;Database=OrderManager_Tests;Trusted_Connection=True;")
                .Options;

            var context = new OrderContext(options);
            context.Database.EnsureCreated();
            return context;
        }
    }
}
```

3.2 Create Customer Service Tests

Create `CustomerServiceTests.cs` in the `OrderManager.Tests` folder:

```
using OrderManager.Core.Models;
using OrderManager.Core.Services;
using System;
using System.Threading.Tasks;
using Xunit;

namespace OrderManager.Tests
{
    public class CustomerServiceTests : TestBase
    {
        [Fact]
        public async Task CreateCustomer_ShouldSetCreatedDateAndActiveStatus()
        {
```

```
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new OrderService(context);

            var customer = new Customer
            {
                Name = "Vinod",
                Email = "vinod@vinod.co"
            };

            // Act
            var result = await service.CreateCustomerAsync(customer);

            // Assert
            Assert.NotEqual(0, result.Id);
            Assert.Equal(DateTime.Now.Date, result.CreatedDate.Date);
            Assert.True(result.IsActive);
        }

        [Fact]
        public async Task GetCustomerById_ShouldReturnCorrectCustomer()
        {
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new OrderService(context);

            var customer = new Customer
            {
                Name = "Vinod",
                Email = "vinod@vinod.co"
            };

            context.Customers.Add(customer);
            await context.SaveChangesAsync();

            // Act
            var result = await service.GetCustomerByIdAsync(customer.Id);

            // Assert
            Assert.NotNull(result);
            Assert.Equal(customer.Id, result.Id);
            Assert.Equal("Vinod", result.Name);
        }

        // TODO: Add more customer-related tests
    }
}
```

3.3 Implement Order Service Tests

Now it's your turn! Implement tests for the `OrderService` methods. Create `OrderServiceTests.cs` in the `OrderManager.Tests` folder. Your tests should cover:

1. Creating an order with items
2. Calculating order totals
3. Retrieving customer orders
4. Updating order status

Here's a sample test to get you started:

```
using OrderManager.Core.Models;
using OrderManager.Core.Services;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Xunit;

namespace OrderManager.Tests
{
    public class OrderServiceTests : TestBase
    {
        [Fact]
        public async Task CreateOrder_ShouldCalculateTotalCorrectly()
        {
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new OrderService(context);

            // Create a customer
            var customer = new Customer
            {
                Name = "Vinod",
                Email = "vinod@vinod.co"
            };
            context.Customers.Add(customer);
            await context.SaveChangesAsync();

            // Prepare order and items
            var order = new Order
            {
                CustomerId = customer.Id
            };

            var items = new List<OrderItem>
            {
                new OrderItem { ProductName = "Laptop", UnitPrice = 1200.00m,
Quantity = 1 },
                new OrderItem { ProductName = "Mouse", UnitPrice = 25.50m,
Quantity = 2 }
            };

            // Act
            var result = await service.CreateOrderAsync(order, items);

            // Assert
```

```
            Assert.Equal(1251.00m, result.TotalAmount);
            Assert.Equal("New", result.Status);
            Assert.Equal(DateTime.Now.Date, result.OrderDate.Date);
        }

        // TODO: Implement the remaining tests
    }
}
```

Part 4: SQL Database Testing (30 minutes)

4.1 Create Integration Test Class

Create `OrderServiceIntegrationTests.cs` in the `OrderManager.Tests` folder:

```
using Microsoft.EntityFrameworkCore;
using OrderManager.Core.Models;
using OrderManager.Core.Services;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Xunit;

namespace OrderManager.Tests
{
    public class OrderServiceIntegrationTests : TestBase, IDisposable
    {
        private readonly OrderContext _context;
        private readonly OrderService _service;

        public OrderServiceIntegrationTests()
        {
            _context = CreateSqlServerContext();
            _service = new OrderService(_context);
        }

        public void Dispose()
        {
            // Clean up test data
            _context.Database.ExecuteSqlRaw("DELETE FROM OrderItems");
            _context.Database.ExecuteSqlRaw("DELETE FROM Orders");
            _context.Database.ExecuteSqlRaw("DELETE FROM Customers");
            _context.Dispose();
        }

        [Fact]
        public async Task CreateAndVerifyOrderWithSql()
        {
            // Arrange
            var customer = new Customer
```

```
            {
                Name = "Vinod",
                Email = "vinod@vinod.co"
            };

            await _service.CreateCustomerAsync(customer);

            var order = new Order { CustomerId = customer.Id };
            var items = new List<OrderItem>
            {
                new OrderItem { ProductName = "Product A", UnitPrice = 10.00m,
Quantity = 2 },
                new OrderItem { ProductName = "Product B", UnitPrice = 15.00m,
Quantity = 1 }
            };

            // Act
            await _service.CreateOrderAsync(order, items);

            // Assert - using direct SQL
            var orderCount = await _context.Database
                .ExecuteSqlRawAsync("SELECT COUNT(*) FROM Orders WHERE CustomerId
= {0}", customer.Id);

            var itemCount = await _context.Database
                .ExecuteSqlRawAsync("SELECT COUNT(*) FROM OrderItems WHERE OrderId
= {0}", order.Id);

            var total = await _context.Database
                .ExecuteSqlInterpolatedAsync($"SELECT SUM(UnitPrice * Quantity)
FROM OrderItems WHERE OrderId = {order.Id}");

            Assert.Equal(1, orderCount);
            Assert.Equal(2, itemCount);
            Assert.Equal(35.00m, total);
        }

        // TODO: Implement tests for transaction rollback behavior
        // TODO: Implement tests for database constraints
    }
}
```

4.2 Complete the Integration Tests

Implement at least three integration tests in the class above. Your tests should:

1. Use direct SQL execution to set up or validate test data
2. Test transaction behavior (commit and rollback)
3. Verify that database constraints are enforced

Submission Requirements

Submit:

1. Your complete solution code
    - Remove the bin and obj folders
    - Create a zip file and submit it on LMS

Helpful Tips

- Use meaningful test names that describe what's being tested
- Follow the Arrange-Act-Assert pattern in your tests
- Consider edge cases (empty collections, invalid inputs, etc.)
- Use transactions to isolate tests when working with a real database
- Use test fixtures to share setup code between multiple tests