Database Testing Lab Exercise

Overview

In this lab, you will apply database testing concepts to develop and test a simple customer management system using Entity Framework Core and .NET. You will focus on creating and testing operations for a single Customers table.

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 (20 minutes)

1.1 Create the Solution Structure

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

   For CustomerManager.Core:

   ```
   Microsoft.EntityFrameworkCore.SqlServer
   Microsoft.EntityFrameworkCore.Design
   System.ComponentModel.Annotations
   ```

   For CustomerManager.Tests:

   ```
   Microsoft.EntityFrameworkCore.InMemory
   Microsoft.NET.Test.Sdk (installed by default for NUnit projects)
   NUnit (installed by default for NUnit projects)
   NUnit3TestAdapter (installed by default for NUnit projects)
   ```

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

1.2 Create the Customer Model

Create the following class in the `CustomerManager.Core/Models` folder:

```
// Customer.cs
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace CustomerManager.Core.Models
{
    // This class represents the Customer entity with validation attributes
    public class Customer
    {
        // Primary key identifier for the customer
        [Key]
        public int Id { get; set; }

        // Required field with maximum length of 100 characters
        [Required] // Will cause validation error if null or empty
        [MaxLength(100)] // Will enforce database column max length
        public string Name { get; set; }

        // Required email with maximum length and format validation
        [Required]
        [MaxLength(150)]
        [EmailAddress] // Validates that the string is in email format
        public string Email { get; set; }

        // Optional phone number with format validation
        [MaxLength(20)]
        [Phone] // Validates that the string is in phone number format
        public string PhoneNumber { get; set; }

        // Boolean flag to indicate if the customer account is active
        public bool IsActive { get; set; }

        // Required timestamp for when the customer was created
        [Required]
        public DateTime CreatedDate { get; set; }

        // Optional timestamp for when the customer was last modified
        // Nullable (DateTime?) means this field can be null
        public DateTime? LastModifiedDate { get; set; }

        // Optional address fields with maximum lengths
        [MaxLength(200)]
        public string Address { get; set; }

        [MaxLength(100)]
        public string City { get; set; }

        [MaxLength(100)]
        public string Country { get; set; }

        // Required postal code with maximum length
        [MaxLength(20)]
        [Required]
        public string PostalCode { get; set; }
    }
}
```

1.3 Create the Database Context

Create `CustomerContext.cs` in the `CustomerManager.Core/Data` folder:

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

namespace CustomerManager.Core.Data
{
    // DbContext is the primary class that coordinates Entity Framework
functionality
    public class CustomerContext : DbContext
    {
        // Constructor that accepts options for configuring the context
        public CustomerContext(DbContextOptions<CustomerContext> options)
            : base(options) // Pass options to the base DbContext constructor
        {
        }

        // DbSet represents the collection of Customer entities in the database
        // It can be used to query and save instances of the Customer entity
        public DbSet<Customer> Customers { get; set; }

        // Override this method to further configure the model that was discovered
by convention
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // Configure the Customer entity to have a unique index on the Email
property
            // This ensures that no two customers can have the same email address
            // This constraint cannot be specified using Data Annotations, so we
use Fluent API
            modelBuilder.Entity<Customer>()
                .HasIndex(c => c.Email)
                .IsUnique();
        }
    }
}
```

Part 2: Service Implementation (20 minutes)

2.1 Create Customer Service Interface

Create `ICustomerService.cs` in the `CustomerManager.Core/Services` folder:

```
using CustomerManager.Core.Models;
using System.Collections.Generic;

namespace CustomerManager.Core.Services
{
    public interface ICustomerService
    {
        Customer GetCustomerById(int id);
        Customer GetCustomerByEmail(string email);
        IEnumerable<Customer> GetAllCustomers();
        IEnumerable<Customer> GetActiveCustomers();
        Customer CreateCustomer(Customer customer);
        bool UpdateCustomer(Customer customer);
        bool DeactivateCustomer(int id);
        bool DeleteCustomer(int id);
        int GetCustomerCount();
        bool BulkCreateCustomers(List<Customer> customers);
    }
}
```

2.2 Implement the Customer Service

Create `CustomerService.cs` in the `CustomerManager.Core/Services` folder:

```
using CustomerManager.Core.Data;
using CustomerManager.Core.Models;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;

namespace CustomerManager.Core.Services
{
    // Service class that implements the ICustomerService interface
    // Contains business logic for customer operations
    public class CustomerService : ICustomerService
    {
        // Private field to store the database context
        private readonly CustomerContext _context;

        // Constructor that accepts a CustomerContext via dependency injection
        public CustomerService(CustomerContext context)
        {
            _context = context;
        }

        // Find a customer by their primary key ID
        public Customer GetCustomerById(int id)
        {
            // Find() is a DbSet method that retrieves an entity by primary key
            return _context.Customers.Find(id);
        }

        // Find a customer by their email address
        public Customer GetCustomerByEmail(string email)
        {
            // LINQ query to find a customer with the matching email
            return _context.Customers
                .Where(c => c.Email == email) // Filter criteria
                .FirstOrDefault(); // Returns the first matching customer or null
if none found
        }

        // Get a list of all customers in the database
        public IEnumerable<Customer> GetAllCustomers()
        {
            // ToList() executes the query and returns the results as a List
            return _context.Customers.ToList();
        }

        // Get only customers with IsActive set to true
        public IEnumerable<Customer> GetActiveCustomers()
        {
            // LINQ query with a filter for active customers
            return _context.Customers
                .Where(c => c.IsActive) // Filter for IsActive == true
                .ToList();
        }

        // Create a new customer in the database
        public Customer CreateCustomer(Customer customer)
        {
            // Set metadata before saving
            customer.CreatedDate = DateTime.Now; // Set creation timestamp
            customer.IsActive = true; // New customers are active by default

            // Add the new customer to the DbSet
            _context.Customers.Add(customer);
            // Persist changes to the database
            _context.SaveChanges();

            // Return the customer with the new ID assigned by the database
            return customer;
        }

        // Update an existing customer's information
        public bool UpdateCustomer(Customer customer)
        {
            // Find the existing customer in the database
            var existingCustomer = _context.Customers.Find(customer.Id);

            // If no customer with this ID exists, return false
            if (existingCustomer == null)
                return false;
            // Update individual properties rather than replacing the entire
entity
            // This approach allows more control over what gets updated
            existingCustomer.Name = customer.Name;
            existingCustomer.PhoneNumber = customer.PhoneNumber;
            existingCustomer.Address = customer.Address;
            existingCustomer.City = customer.City;
            existingCustomer.Country = customer.Country;
            existingCustomer.PostalCode = customer.PostalCode;
            existingCustomer.LastModifiedDate = DateTime.Now; // Update
modification timestamp

            // Note: Email cannot be changed after creation (business rule)

            // Save changes to the database
            _context.SaveChanges();
            return true;
        }

        // Mark a customer as inactive instead of deleting them
        public bool DeactivateCustomer(int id)
        {
            // Find the customer by ID
            var customer = _context.Customers.Find(id);

            // If no customer with this ID exists, return false
            if (customer == null)
                return false;

            // Update the IsActive flag and modification timestamp
            customer.IsActive = false;
            customer.LastModifiedDate = DateTime.Now;

            // Save changes to the database
            _context.SaveChanges();
            return true;
        }

        // Permanently remove a customer from the database
        public bool DeleteCustomer(int id)
        {
            // Find the customer by ID
            var customer = _context.Customers.Find(id);

            // If no customer with this ID exists, return false
            if (customer == null)
                return false;

            // Remove the customer from the DbSet
            _context.Customers.Remove(customer);
            // Persist the deletion to the database
            _context.SaveChanges();
            return true;
        }

        // Get the total number of customers in the database
        public int GetCustomerCount()
        {
            // Count() executes a COUNT query against the database
            return _context.Customers.Count();
        }

        // Create multiple customers in a single transaction
        public bool BulkCreateCustomers(List<Customer> customers)
        {
            // Begin a transaction to ensure atomicity (all or nothing)
            // If any part fails, the entire operation is rolled back
            using var transaction = _context.Database.BeginTransaction();

            try
            {
                // Process each customer in the list
                foreach (var customer in customers)
                {
                    // Set metadata before saving
                    customer.CreatedDate = DateTime.Now;
                    customer.IsActive = true;
                    // Add to the DbSet
                    _context.Customers.Add(customer);
                }

                // Save all changes in one batch
                _context.SaveChanges();
                // Commit the transaction if everything succeeded
                transaction.Commit();
                return true;
            }
            catch
            {
                // If any error occurs, roll back the transaction
                // This ensures no partial data is saved
                transaction.Rollback();
                return false;
            }
        }
    }
}
```

Part 3: Testing (40 minutes)

3.1 Create Test Base Class

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

```csharp
using CustomerManager.Core.Data;
using Microsoft.EntityFrameworkCore;
using NUnit.Framework;
using System;

namespace CustomerManager.Tests
{
    // Base class for all test classes, containing common setup functionality
    public abstract class TestBase
    {
        // Creates an in-memory database context for fast, isolated unit tests
        protected CustomerContext CreateInMemoryContext()
        {
            // Configure the context to use an in-memory database with a unique
name
            // Using Guid.NewGuid() ensures each test gets its own isolated
database
            var options = new DbContextOptionsBuilder<CustomerContext>()
                .UseInMemoryDatabase(databaseName: Guid.NewGuid().ToString())
                .Options;

            // Create a new context with the options
            var context = new CustomerContext(options);
            // Ensure the database is created
            context.Database.EnsureCreated();
            return context;
        }

        // Creates a SQL Server context for integration tests with a real database
        protected CustomerContext CreateSqlServerContext()
        {
            // Configure the context to use SQL Server
            var options = new DbContextOptionsBuilder<CustomerContext>()
                .UseSqlServer("Server=
(localdb)\\mssqllocaldb;Database=CustomerManager_Tests;Trusted_Connection=True;")
                .Options;

            // Create a new context with the options
            var context = new CustomerContext(options);

            // Make sure database is in a clean state for each test
            // This ensures test isolation - each test starts with a fresh
database
            context.Database.EnsureDeleted();
            context.Database.EnsureCreated();

            return context;
        }
    }
}
```

3.2 Create In-Memory Database Tests

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

```
using CustomerManager.Core.Models;
using CustomerManager.Core.Services;
using NUnit.Framework;
using System;
using System.Linq;

namespace CustomerManager.Tests
{
    [TestFixture]
    public class CustomerServiceTests : TestBase
    {
        [Test]
        public void CreateCustomer_ShouldSetMetadata()
        {
            // Arrange: Set up the test environment
            // Create an in-memory database context for this test
            using var context = CreateInMemoryContext();
            // Create the service with the test context
            var service = new CustomerService(context);

            // Create a test customer object with all required fields
            var customer = new Customer
            {
                Name = "Rajesh Sharma",
                Email = "rajesh.sharma@example.com",
                PhoneNumber = "9876543210",
                Address = "42 Mahatma Gandhi Road",
                City = "Bangalore",
                Country = "India",
                PostalCode = "560001"
            };

            // Act: Execute the method being tested
            var result = service.CreateCustomer(customer);

            // Assert: Verify the expected outcomes
            // Check that an ID was assigned (non-zero)
            Assert.That(result.Id, Is.Not.EqualTo(0));
            // Check that CreatedDate was set to today
            Assert.That(result.CreatedDate.Date, Is.EqualTo(DateTime.Now.Date));
            // Check that IsActive was set to true
            Assert.That(result.IsActive, Is.True);
        }

        [Test]
        public void GetCustomerById_ShouldReturnCorrectCustomer()
        {
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new CustomerService(context);
            var customer = new Customer
            {
                Name = "Priya Patel",
                Email = "priya.patel@example.com",
                PhoneNumber = "9988776655",
                Address = "15 Nehru Street",
                City = "Mumbai",
                Country = "India",
                PostalCode = "400001"
            };

            context.Customers.Add(customer);
            context.SaveChanges();

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

            // Assert
            Assert.That(result, Is.Not.Null);
            Assert.That(result.Id, Is.EqualTo(customer.Id));
            Assert.That(result.Name, Is.EqualTo("Priya Patel"));
        }

        [Test]
        public void GetCustomerByEmail_ShouldReturnCorrectCustomer()
        {
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new CustomerService(context);

            var customer = new Customer
            {
                Name = "Anand Verma",
                Email = "anand.verma@example.com",
                PhoneNumber = "8765432109",
                Address = "27 Tagore Lane",
                City = "Delhi",
                Country = "India",
                PostalCode = "110001"
            };

            context.Customers.Add(customer);
            context.SaveChanges();

            // Act
            var result = service.GetCustomerByEmail("anand.verma@example.com");

            // Assert
            Assert.That(result, Is.Not.Null);
            Assert.That(result.Name, Is.EqualTo("Anand Verma"));
        }

        [Test]
        public void GetActiveCustomers_ShouldReturnOnlyActiveCustomers()
        {
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new CustomerService(context);

            context.Customers.AddRange(
                new Customer {
                    Name = "Arun Kumar",
                    Email = "arun.kumar@example.com",
                    IsActive = true,
                    PhoneNumber = "7654321098",
                    Address = "8 Patel Nagar",
                    City = "Jaipur",
                    Country = "India",
                    PostalCode = "302001"
                },
                new Customer {
                    Name = "Meera Singh",
                    Email = "meera.singh@example.com",
                    IsActive = false,
                    PhoneNumber = "9012345678",
                    Address = "54 Lal Bahadur Colony",
                    City = "Chennai",
                    Country = "India",
                    PostalCode = "600001"
                },
                new Customer {
                    Name = "Vikram Joshi",
                    Email = "vikram.joshi@example.com",
                    IsActive = true,
                    PhoneNumber = "8901234567",
                    Address = "23 Rajaji Avenue",
                    City = "Hyderabad",
                    Country = "India",
                    PostalCode = "500001"
                }
            );
            context.SaveChanges();

            // Act
            var results = service.GetActiveCustomers();

            // Assert
            Assert.That(results.Count(), Is.EqualTo(2));
            Assert.That(results, Is.All.Matches<Customer>(c => c.IsActive));
        }

        [Test]
        public void DeactivateCustomer_ShouldSetIsActiveToFalse()
        {
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new CustomerService(context);
            var customer = new Customer
            {
                Name = "Deepa Gupta",
                Email = "deepa.gupta@example.com",
                PhoneNumber = "7890123456",
                Address = "11 Indira Nagar",
                City = "Lucknow",
                Country = "India",
                PostalCode = "226001",
                IsActive = true
            };

            context.Customers.Add(customer);
            context.SaveChanges();

            // Act
            var result = service.DeactivateCustomer(customer.Id);
            var updatedCustomer = service.GetCustomerById(customer.Id);

            // Assert
            Assert.That(result, Is.True);
            Assert.That(updatedCustomer.IsActive, Is.False);
            Assert.That(updatedCustomer.LastModifiedDate, Is.Not.Null);
        }

        [Test]
        public void UpdateCustomer_ShouldUpdatePropertiesAndSetLastModifiedDate()
        {
            // Arrange
            using var context = CreateInMemoryContext();
            var service = new CustomerService(context);

            var customer = new Customer
            {
                Name = "Sanjay Malhotra",
                Email = "sanjay.malhotra@example.com",
                PhoneNumber = "9876123450",
                Address = "39 Sardar Patel Road",
                City = "Ahmedabad",
                Country = "India",
                PostalCode = "380001"
            };

            context.Customers.Add(customer);
            context.SaveChanges();

            // Update customer information
            customer.Name = "Sanjay Kumar Malhotra";
            customer.PhoneNumber = "9876123451";
            customer.City = "Gandhinagar";

            // Act
            var result = service.UpdateCustomer(customer);
            var updatedCustomer = service.GetCustomerById(customer.Id);
            // Assert
            Assert.That(result, Is.True);
            Assert.That(updatedCustomer.Name, Is.EqualTo("Sanjay Kumar
Malhotra"));
            Assert.That(updatedCustomer.PhoneNumber, Is.EqualTo("9876123451"));
            Assert.That(updatedCustomer.City, Is.EqualTo("Gandhinagar"));
            Assert.That(updatedCustomer.LastModifiedDate, Is.Not.Null);
        }

        // TODO: Implement additional tests for remaining service methods
    }
}
```

3.3 Create SQL Database Integration Tests

Create `CustomerServiceIntegrationTests.cs` in the `CustomerManager.Tests` folder:

```
using CustomerManager.Core.Models;
using CustomerManager.Core.Services;
using Microsoft.EntityFrameworkCore;
using NUnit.Framework;
using System;
using System.Collections.Generic;

namespace CustomerManager.Tests
{
    [TestFixture]
    public class CustomerServiceIntegrationTests : TestBase
    {
        private CustomerContext _context;
        private CustomerService _service;

        [SetUp]
        public void Setup()
        {
            _context = CreateSqlServerContext();
            _service = new CustomerService(_context);
        }

        [TearDown]
        public void Cleanup()
        {
            // Clean up test data
            _context.Database.ExecuteSqlRaw("DELETE FROM Customers");
            _context.Dispose();
        }

        [Test]
        public void CreateAndRetrieveCustomer_WithSql()
        {
            // Arrange: Create a test customer
            var customer = new Customer
            {
                Name = "Kavita Reddy",
                Email = "kavita.reddy@example.com",
                PhoneNumber = "9988776600",
                Address = "7 Cubbon Road",
                City = "Bangalore",
                Country = "India",
                PostalCode = "560002"  // Required field
            };

            // Act: Save the customer using the service
            _service.CreateCustomer(customer);

            // Assert: Use SQL queries to verify the data was stored correctly

            // NOT RECOMMENDED: This doesn't work as expected because
ExecuteSqlRaw returns
            // the number of rows affected by the command, not the result of a
SELECT query
            var count = _context.Database
                .ExecuteSqlRaw("SELECT COUNT(1) FROM Customers WHERE Email = {0}",
"kavita.reddy@example.com");

            // BETTER APPROACH: Use FromSqlRaw to execute a query and then count
the results
            // This uses parameterized queries which prevents SQL injection
            var actualCount = _context.Customers
                .FromSqlRaw("SELECT * FROM Customers WHERE Email = {0}",
"kavita.reddy@example.com")
                .Count();

            // Retrieve the full customer record to verify all fields
            var retrievedCustomer = _context.Customers
                .FromSqlRaw("SELECT * FROM Customers WHERE Email = {0}",
"kavita.reddy@example.com")
                .FirstOrDefault();

            // Verify the count and customer data
            Assert.That(actualCount, Is.EqualTo(1));
            Assert.That(retrievedCustomer.Name, Is.EqualTo("Kavita Reddy"));
            Assert.That(retrievedCustomer.PostalCode, Is.EqualTo("560002"));
        }

        [Test]
        public void UniqueEmailConstraint_ShouldPreventDuplicates()
        {
            // Arrange: Create and save a customer with a specific email
            var customer1 = new Customer
            {
                Name = "Rahul Mehta",
                Email = "rahul.mehta@example.com", // This email will be used
twice
```

```
                PhoneNumber = "7788990011",
                Address = "18 Bandra West",
                City = "Mumbai",
                Country = "India",
                PostalCode = "400050"
            };

            // Create a second customer with the same email
            var customer2 = new Customer
            {
                Name = "Rohan Mehta",
                Email = "rahul.mehta@example.com", // Duplicate email - should
violate constraint
                PhoneNumber = "7788990022",
                Address = "22 Andheri East",
                City = "Mumbai",
                Country = "India",
                PostalCode = "400069"
            };

            // Save the first customer - should succeed
            _service.CreateCustomer(customer1);

            // Act & Assert: Try to save the second customer and expect an
exception
            // DbUpdateException is thrown when a database constraint is violated
            Assert.Throws<DbUpdateException>(() =>
                _service.CreateCustomer(customer2));
            // This test verifies that our unique index on Email is working
correctly
        }

        [Test]
        public void RequiredFields_ShouldBeEnforced()
        {
            // Arrange: Create customers with missing required fields

            // Customer missing Name (which is required)
            var customerNoName = new Customer
            {
                Email = "nameless@example.com",
                Name = null, // Missing required field
                PhoneNumber = "9876543211",
                Address = "5 Karol Bagh",
                City = "New Delhi",
                Country = "India",
                PostalCode = "110005"
            };

            // Customer missing Email (which is required)
            var customerNoEmail = new Customer
            {
                Name = "Suresh Iyer",
                Email = null, // Missing required field
                PhoneNumber = "9123456789",
                Address = "14 T Nagar",
                City = "Chennai",
                Country = "India",
                PostalCode = "600017"
            };

            // Customer missing PostalCode (which is required)
            var customerNoPostalCode = new Customer
            {
                Name = "Amit Patel",
                Email = "amit.patel@example.com",
                PhoneNumber = "9898765432",
                Address = "35 Satellite Road",
                City = "Ahmedabad",
                Country = "India",
                PostalCode = null // Missing required field
            };

            // Act & Assert: Try to save each customer and expect an exception
            // These tests verify that Entity Framework correctly enforces our
[Required] constraints
            Assert.Throws<DbUpdateException>(() =>
                _service.CreateCustomer(customerNoName));

            Assert.Throws<DbUpdateException>(() =>
                _service.CreateCustomer(customerNoEmail));

            Assert.Throws<DbUpdateException>(() =>
                _service.CreateCustomer(customerNoPostalCode));
        }

        [Test]
        public void MaxLengthConstraints_ShouldBeEnforced()
        {
            // Arrange
            var customer = new Customer
            {
                Name = new string('A', 101),
                Email = "toolong@example.com",
                PhoneNumber = "9898989898",
                Address = "9 MG Road",
                City = "Pune",
                Country = "India",
                PostalCode = "411001"
            };

            // Act & Assert
            Assert.Throws<DbUpdateException>(() =>
                _service.CreateCustomer(customer));
        }

        [Test]
        public void BulkCreate_ShouldRollbackOnFailure()
        {
            // Arrange: Create a list of customers with one invalid entry
            var customers = new List<Customer>
            {
                // First customer - valid
                new Customer {
                    Name = "Arjun Nair",
                    Email = "arjun.nair@example.com",
                    PhoneNumber = "9567891234",
                    Address = "28 Koramangala",
                    City = "Bangalore",
                    Country = "India",
                    PostalCode = "560034"
                },
                // Second customer - valid
                new Customer {
                    Name = "Lakshmi Menon",
                    Email = "lakshmi.menon@example.com",
                    PhoneNumber = "9446789123",
                    Address = "42 Jayanagar",
                    City = "Bangalore",
                    Country = "India",
                    PostalCode = "560041"
                },
                // Third customer - invalid (duplicate email)
                new Customer {
                    Name = "Pranav Nambiar",
                    Email = "arjun.nair@example.com", // Duplicate email - will
cause constraint violation
                    PhoneNumber = "9387651234",
                    Address = "15 Indiranagar",
                    City = "Bangalore",
                    Country = "India",
                    PostalCode = "560038"
                }
            };

            // Act: Attempt to bulk create the customers
            var result = _service.BulkCreateCustomers(customers);

            // Assert: Verify that the operation failed
            Assert.That(result, Is.False);

            // Verify that no records were inserted due to rollback
            var count = _context.Customers
                .FromSqlRaw("SELECT * FROM Customers WHERE Email LIKE {0}",
"arjun%@example.com")
                .Count();

            // Count should be 0 because the transaction was rolled back
            Assert.That(count, Is.EqualTo(0));
        }

        [Test]
        public void BulkCreate_ShouldCommitAllChanges()
        {
            // Arrange
            var customers = new List<Customer>
            {
                new Customer {
                    Name = "Karthik Iyer",
                    Email = "karthik.iyer@example.com",
                    PhoneNumber = "9848012345",
                    Address = "7 Ameerpet",
                    City = "Hyderabad",
                    Country = "India",
                    PostalCode = "500016"
                },
                new Customer {
                    Name = "Divya Sharma",
                    Email = "divya.sharma@example.com",
                    PhoneNumber = "9912345678",
                    Address = "22 Jubilee Hills",
                    City = "Hyderabad",
                    Country = "India",
                    PostalCode = "500033"
                },
                new Customer {
                    Name = "Mohan Krishna",
                    Email = "mohan.krishna@example.com",
                    PhoneNumber = "9010203040",
                    Address = "11 Gachibowli",
                    City = "Hyderabad",
                    Country = "India",
                    PostalCode = "500032"
                }
            };

            // Act
            var result = _service.BulkCreateCustomers(customers);

            // Assert
            Assert.That(result, Is.True);

            // Verify all records were inserted
            var count = _service.GetCustomerCount();
            Assert.That(count, Is.EqualTo(3));
        }
    }
}
```

Part 4: Testing Transactions (20 minutes)

4.1 Test Transaction Behavior

Add these tests to CustomerServiceIntegrationTests.cs:

```
[Test]
public void BulkCreate_ShouldRollbackOnFailure()
{
    // Arrange
    var customers = new List<Customer>
    {
        new Customer {
            Name = "Arjun Nair",
            Email = "arjun.nair@example.com",
            PhoneNumber = "9567891234",
            Address = "28 Koramangala",
            City = "Bangalore",
            Country = "India",
            PostalCode = "560034"
        },
        new Customer {
            Name = "Lakshmi Menon",
            Email = "lakshmi.menon@example.com",
            PhoneNumber = "9446789123",
            Address = "42 Jayanagar",
            City = "Bangalore",
            Country = "India",
            PostalCode = "560041"
        },
        new Customer {
            Name = "Pranav Nambiar",
            Email = "arjun.nair@example.com", // Duplicate email
            PhoneNumber = "9387651234",
            Address = "15 Indiranagar",
            City = "Bangalore",
            Country = "India",
            PostalCode = "560038"
        }
    };

    // Act
    var result = _service.BulkCreateCustomers(customers);

    // Assert
    Assert.That(result, Is.False);

    // Verify that no records were inserted due to rollback
    var count = _context.Customers
        .FromSqlRaw("SELECT * FROM Customers WHERE Email LIKE {0}",
"arjun%@example.com")
        .Count();

    Assert.That(count, Is.EqualTo(0));
}

[Test]
public void BulkCreate_ShouldCommitAllChanges()
```

```
{
    // Arrange
    var customers = new List<Customer>
    {
        new Customer {
            Name = "Karthik Iyer",
            Email = "karthik.iyer@example.com",
            PhoneNumber = "9848012345",
            Address = "7 Ameerpet",
            City = "Hyderabad",
            Country = "India",
            PostalCode = "500016"
        },
        new Customer {
            Name = "Divya Sharma",
            Email = "divya.sharma@example.com",
            PhoneNumber = "9912345678",
            Address = "22 Jubilee Hills",
            City = "Hyderabad",
            Country = "India",
            PostalCode = "500033"
        },
        new Customer {
            Name = "Mohan Krishna",
            Email = "mohan.krishna@example.com",
            PhoneNumber = "9010203040",
            Address = "11 Gachibowli",
            City = "Hyderabad",
            Country = "India",
            PostalCode = "500032"
        }
    };

    // Act
    var result = _service.BulkCreateCustomers(customers);

    // Assert
    Assert.That(result, Is.True);

    // Verify all records were inserted
    var count = _service.GetCustomerCount();
    Assert.That(count, Is.EqualTo(3));
}
```

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 (null values, empty strings, etc.)
- Use transactions to isolate tests when working with a real database
- Test both positive and negative scenarios
- Ensure your tests are deterministic and don't rely on specific data already in the database