Database Testing with SQL and .NET Core Tutorial

Introduction

Database testing is a critical aspect of ensuring application reliability. In this tutorial, we'll explore how to effectively test database interactions using SQL and .NET Core, combining the power of Entity Framework Core (EF Core) with proper testing practices.

Introduction to SQL for Validation and Test Data Management

Why SQL for Testing?

SQL provides direct access to your database, allowing you to:

- Verify data states without going through your application code
- Set up test environments with specific data
- Clean up test data efficiently
- Perform complex validation queries that might be cumbersome in application code

Essential SQL for Testing

1. Query-Based Validation

```sql
-- Validate record existence
SELECT COUNT(*) FROM Orders WHERE CustomerId = 101;

-- Validate data integrity
SELECT
    OrderId,
    CustomerId,
    OrderDate,
    CASE
        WHEN TotalAmount = (SELECT SUM(Price * Quantity) FROM OrderItems WHERE
OrderId = o.OrderId)
        THEN 'Valid'
        ELSE 'Invalid'
    END AS AmountStatus
FROM Orders o
WHERE OrderId = 500;
```

2. Test Data Setup

```sql
-- Insert test records
INSERT INTO Customers (CustomerId, Name, Email)
VALUES (999, 'Test Customer', 'test@example.com');
```

```
-- Insert related records
INSERT INTO Orders (OrderId, CustomerId, OrderDate, TotalAmount)
VALUES (9999, 999, GETDATE(), 150.00);
```

3. Test Data Cleanup

```
-- Clean up test data (reverse order due to foreign keys)
DELETE FROM OrderItems WHERE OrderId IN (SELECT OrderId FROM Orders WHERE
CustomerId = 999);
DELETE FROM Orders WHERE CustomerId = 999;
DELETE FROM Customers WHERE CustomerId = 999;

-- Alternative approach with cascading deletes
DELETE FROM Customers WHERE CustomerId = 999;
```

Testing Best Practices with SQL

1. **Use Transactions**: Wrap test operations in transactions to ensure cleanup
2. **Isolate Test Data**: Use unique identifiers or schemas for test data
3. **Verify Constraints**: Test foreign key constraints, unique constraints, etc.
4. **Test Edge Cases**: NULL values, boundary conditions, and error cases

Writing Automated Tests for Database Validation

Setting Up a Test Project

1. Create a new xUnit test project:

```
dotnet new xunit -n YourApp.Database.Tests
```

2. Add required packages:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.NET.Test.Sdk
dotnet add package xunit
dotnet add package xunit.runner.visualstudio
```

Database Connection for Tests

```
using Microsoft.EntityFrameworkCore;
using System;
using Xunit;

public class DatabaseTestBase : IDisposable
```

```
{
    protected readonly YourDbContext _context;

    public DatabaseTestBase()
    {
        var options = new DbContextOptionsBuilder<YourDbContext>()

.UseSqlServer("Server=localhost;Database=TestDb;Trusted_Connection=True;")
            .EnableSensitiveDataLogging()
            .Options;

        _context = new YourDbContext(options);

        // Ensure database is created
        _context.Database.EnsureCreated();
    }

    public void Dispose()
    {
        _context.Dispose();
    }
}
```

Writing Basic Database Tests

```
public class CustomerTests : DatabaseTestBase
{
    [Fact]
    public void InsertCustomer_ShouldPersistData()
    {
        // Arrange
        var customer = new Customer
        {
            Name = "Test Customer",
            Email = "test@example.com"
        };

        // Act
        _context.Customers.Add(customer);
        _context.SaveChanges();

        // Assert
        var savedCustomer = _context.Customers
            .FirstOrDefault(c => c.Email == "test@example.com");

        Assert.NotNull(savedCustomer);
        Assert.Equal("Test Customer", savedCustomer.Name);
    }

    [Fact]
    public void UpdateCustomer_ShouldUpdateDatabase()
```

```
        {
            // Arrange
            var customer = new Customer
            {
                Name = "Original Name",
                Email = "update@example.com"
            };

            _context.Customers.Update(customer);
            _context.SaveChanges();

            // Act
            var savedCustomer = _context.Customers
                .First(c => c.Email == "update@example.com");
            savedCustomer.Name = "Updated Name";
            _context.SaveChanges();

            // Assert
            var updatedCustomer = _context.Customers
                .AsNoTracking() // Important to get fresh data from DB
                .First(c => c.Email == "update@example.com");

            Assert.Equal("Updated Name", updatedCustomer.Name);
        }
    }
```

Direct SQL Execution in Tests

```
    [Fact]
    public async Task DirectSqlTest_ShouldReturnExpectedData()
    {
        // Arrange
        await _context.Database.ExecuteSqlRawAsync(@"
            INSERT INTO Customers (Name, Email)
            VALUES ('SQL Test', 'sql@example.com')
        ");

        // Act
        var result = await _context.Customers
            .FromSqlRaw("SELECT * FROM Customers WHERE Email = 'sql@example.com'")
            .ToListAsync();

        // Assert
        Assert.Single(result);
        Assert.Equal("SQL Test", result[0].Name);

        // Cleanup
        await _context.Database.ExecuteSqlRawAsync(@"
            DELETE FROM Customers WHERE Email = 'sql@example.com'
        ");
    }
```

Transaction-Based Tests

```
[Fact]
public async Task TransactionTest_ShouldRollback()
{
    // Start a transaction
    using var transaction = await _context.Database.BeginTransactionAsync();

    try
    {
        // Setup test data
        var customer = new Customer
        {
            Name = "Transaction Test",
            Email = "transaction@example.com"
        };

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

        // Verify data exists
        var exists = await _context.Customers
            .AnyAsync(c => c.Email == "transaction@example.com");
        Assert.True(exists);

        // Don't commit - let the transaction roll back automatically
    }
    finally
    {
        // Explicitly roll back (though disposal will do this too)
        await transaction.RollbackAsync();
    }

    // Verify data is gone after rollback
    var stillExists = await _context.Customers
        .AnyAsync(c => c.Email == "transaction@example.com");
    Assert.False(stillExists);
}
```

Using EF Core for Data Setup and Teardown in Tests

Creating a Test Database Context

```
public class TestDbContext : DbContext
{
    public TestDbContext(DbContextOptions<TestDbContext> 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)
    {
        // Configure test-specific model settings
        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);
    }
}
```

Setting Up Test Data

```
public class OrderServiceTests : IClassFixture<DatabaseFixture>
{
    private readonly DatabaseFixture _fixture;

    public OrderServiceTests(DatabaseFixture fixture)
    {
        _fixture = fixture;
    }

    [Fact]
    public async Task CalculateOrderTotal_ShouldSumItems()
    {
        // Arrange
        var context = _fixture.CreateContext();
        var customer = await CreateTestCustomerAsync(context);
        var order = await CreateTestOrderAsync(context, customer.Id);
        await AddTestOrderItemsAsync(context, order.Id);

        var orderService = new OrderService(context);

        // Act
        var total = await orderService.CalculateOrderTotalAsync(order.Id);

        // Assert
        Assert.Equal(125.40m, total);
    }
    private async Task<Customer> CreateTestCustomerAsync(TestDbContext context)
    {
        var customer = new Customer
        {
            Name = "Test Customer",
            Email = $"test-{Guid.NewGuid()}@example.com" // Unique email
        };

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

    private async Task<Order> CreateTestOrderAsync(TestDbContext context, int
customerId)
    {
        var order = new Order
        {
            CustomerId = customerId,
            OrderDate = DateTime.Now,
            Status = "Pending"
        };

        context.Orders.Add(order);
        await context.SaveChangesAsync();
        return order;
    }

    private async Task AddTestOrderItemsAsync(TestDbContext context, int orderId)
    {
        var items = new List<OrderItem>
        {
            new OrderItem { OrderId = orderId, ProductName = "Product 1", Price =
25.50m, Quantity = 2 },
            new OrderItem { OrderId = orderId, ProductName = "Product 2", Price =
74.40m, Quantity = 1 }
        };

        context.OrderItems.AddRange(items);
        await context.SaveChangesAsync();
    }
}
```

Test Fixture for Database Setup

```
public class DatabaseFixture : IDisposable
{
    private const string ConnectionString =
        "Server=localhost;Database=OrderServiceTests;Trusted_Connection=True;";
    private static readonly object _lock = new object();
    private static bool _databaseInitialized;

    public DatabaseFixture()
    {
        lock (_lock)
        {
            if (!_databaseInitialized)
            {
                using (var context = CreateContext())
                {
                    context.Database.EnsureDeleted();
                    context.Database.EnsureCreated();

                    // Seed with any baseline data all tests might need
                    SeedDatabase(context);
                }

                _databaseInitialized = true;
            }
        }
    }

    public TestDbContext CreateContext()
    {
        return new TestDbContext(
            new DbContextOptionsBuilder<TestDbContext>()
                .UseSqlServer(ConnectionString)
                .Options);
    }

    private void SeedDatabase(TestDbContext context)
    {
        // Add any seed data common to all tests
        var productCategories = new[]
        {
            new ProductCategory { Name = "Electronics" },
            new ProductCategory { Name = "Books" },
            new ProductCategory { Name = "Clothing" }
        };

        context.ProductCategories.AddRange(productCategories);
        context.SaveChanges();
    }

    public void Dispose()
    {
        // Uncomment if you want to delete the test database when done
        // using (var context = CreateContext())
        // {
        //     context.Database.EnsureDeleted();
        // }
    }
}
```

Leveraging EF Core's InMemory Provider for Faster Tests

```
public class InMemoryOrderServiceTests
{
    [Fact]
    public async Task InMemoryTest_OrderCalculation()
    {
        // Arrange
        var options = new DbContextOptionsBuilder<TestDbContext>()
            .UseInMemoryDatabase(databaseName: "OrderCalcTest")
            .Options;

        // Create the context and add test data
        using (var context = new TestDbContext(options))
        {
            var customer = new Customer { Name = "Memory Test", Email =
"memory@example.com" };
            context.Customers.Add(customer);
            await context.SaveChangesAsync();

            var order = new Order { CustomerId = customer.Id, OrderDate =
DateTime.Now };
            context.Orders.Add(order);
            await context.SaveChangesAsync();

            context.OrderItems.Add(new OrderItem { OrderId = order.Id, ProductName
= "Product A", Price = 10, Quantity = 2 });
            context.OrderItems.Add(new OrderItem { OrderId = order.Id, ProductName
= "Product B", Price = 15, Quantity = 1 });
            await context.SaveChangesAsync();

            // Act
            var orderService = new OrderService(context);
            var total = await orderService.CalculateOrderTotalAsync(order.Id);

            // Assert
            Assert.Equal(35, total);
        }
    }
}
```

Advanced: Testing with Database Snapshots

```
public class SnapshotTests : IDisposable
{
    private readonly string _masterConnectionString =
        "Server=localhost;Database=master;Trusted_Connection=True;";
    private readonly string _dbConnectionString =
        "Server=localhost;Database=SnapshotTest;Trusted_Connection=True;";
    private readonly string _snapshotName = "SnapshotTest_snapshot";

    public SnapshotTests()
    {
        // Create a test database
        using (var connection = new SqlConnection(_masterConnectionString))
        {
            connection.Open();

            // Drop existing DB if needed
            using (var command = connection.CreateCommand())
            {
                command.CommandText = @"
                    IF EXISTS(SELECT * FROM sys.databases WHERE name =
'SnapshotTest')
                    BEGIN
                        ALTER DATABASE SnapshotTest SET SINGLE_USER WITH ROLLBACK
IMMEDIATE;
                        DROP DATABASE SnapshotTest;
                    END
                    CREATE DATABASE SnapshotTest;
                ";
                command.ExecuteNonQuery();
            }
        }

        // Set up test database
        using (var context = CreateContext())
        {
            context.Database.EnsureCreated();
            SeedDatabase(context);

            // Create a snapshot
            using (var connection = new SqlConnection(_masterConnectionString))
            {
                connection.Open();
                using (var command = connection.CreateCommand())
                {
                    command.CommandText = $@"
                        CREATE DATABASE {_snapshotName} ON
                        (NAME = SnapshotTest, FILENAME = 'C:\Temp\
{_snapshotName}.ss')
                        AS SNAPSHOT OF SnapshotTest;
                    ";
                    command.ExecuteNonQuery();
                }
            }
        }
    }

    private TestDbContext CreateContext()
    {
        return new TestDbContext(
            new DbContextOptionsBuilder<TestDbContext>()
                .UseSqlServer(_dbConnectionString)
                .Options);
    }

    private void SeedDatabase(TestDbContext context)
    {
        // Add baseline data
        var customer = new Customer
        {
            Name = "Snapshot Test",
            Email = "snapshot@example.com"
        };
        context.Customers.Add(customer);
        context.SaveChanges();
    }

    [Fact]
    public async Task TestWithSnapshot_ShouldResetDataAfterTest()
    {
        // Arrange
        using (var context = CreateContext())
        {
            // Act - modify data
            var customer = await context.Customers.FirstAsync();
            customer.Name = "Modified Name";
            await context.SaveChangesAsync();

            var newCustomer = new Customer
            {
                Name = "New Customer",
                Email = "new@example.com"
            };
            context.Customers.Add(newCustomer);
            await context.SaveChangesAsync();

            // Verify changes
            Assert.Equal(2, await context.Customers.CountAsync());
            Assert.Equal("Modified Name", customer.Name);
        }

        // Reset database to snapshot
        using (var connection = new SqlConnection(_masterConnectionString))
        {
            connection.Open();
            using (var command = connection.CreateCommand())
            {
                command.CommandText = @"
                    ALTER DATABASE SnapshotTest SET SINGLE_USER WITH ROLLBACK
IMMEDIATE;
                    RESTORE DATABASE SnapshotTest FROM DATABASE_SNAPSHOT = '" +
_snapshotName + @"';
                    ALTER DATABASE SnapshotTest SET MULTI_USER;
                ";
                command.ExecuteNonQuery();
            }
        }

        // Verify database is reset
        using (var context = CreateContext())
        {
            Assert.Equal(1, await context.Customers.CountAsync());
            var customer = await context.Customers.FirstAsync();
            Assert.Equal("Snapshot Test", customer.Name);
        }
    }

    public void Dispose()
    {
        // Clean up - drop snapshot and test database
        using (var connection = new SqlConnection(_masterConnectionString))
        {
            connection.Open();
            using (var command = connection.CreateCommand())
            {
                command.CommandText = $@"
                    IF EXISTS(SELECT * FROM sys.databases WHERE name =
'{_snapshotName}')
                    BEGIN
                        DROP DATABASE {_snapshotName};
                    END

                    IF EXISTS(SELECT * FROM sys.databases WHERE name =
'SnapshotTest')
                    BEGIN
                        ALTER DATABASE SnapshotTest SET SINGLE_USER WITH ROLLBACK
IMMEDIATE;
                        DROP DATABASE SnapshotTest;
                    END
                ";
                command.ExecuteNonQuery();
            }
        }
    }
}
```

Best Practices and Conclusion

Best Practices

1. **Isolation**: Keep test databases separate from production
2. **Idempotence**: Tests should be repeatable with the same results
3. **Clean State**: Each test should start from a known state
4. **Performance**: Use in-memory providers for unit tests, real databases for integration tests
5. **Mocking**: Consider mocking the DbContext for pure unit tests

6. **Data Generation**: Use tools like Bogus for generating realistic test data
7. **Test Real SQL**: Test stored procedures and complex queries directly

Conclusion

Effective database testing combines proper SQL knowledge with .NET Core's testing capabilities and EF Core's data management features. By following the patterns shown in this tutorial, you can ensure your database interactions are reliable, performant, and maintainable.

Remember that different testing scenarios call for different approaches:

- Use in-memory providers for fast unit tests
- Use real database connections for integration tests
- Use transactions for test isolation
- Use snapshots for complex scenarios

Happy testing!