Database Testing with SQL and .NET Core

Overview

This guide explores essential concepts and techniques for database testing in .NET Core applications. Database testing ensures that your application's data layer functions correctly, verifying that your database operations, stored procedures, and data access code meet requirements for functionality, performance, and data integrity.

Database testing is a critical part of the software testing process, especially for applications that rely heavily on data persistence. Unlike API or UI testing, database testing focuses specifically on validating that your data storage, retrieval, and manipulation functions work as expected, and that your database schema supports your application's requirements.

Table of Contents

Introduction to Database Testing

Database testing is often overlooked in testing strategies, but it's crucial for ensuring application reliability. Database errors can be particularly problematic because they often affect data integrity and might not be immediately visible through the application interface. Thorough database testing helps identify issues early in the development process.

What is Database Testing?

Database testing is a type of software testing that examines the schema, tables, stored procedures, triggers, and other database components to ensure that data is stored, retrieved, updated, and deleted correctly. It verifies data integrity, business rules enforcement, and performance of database operations.

Types of Database Testing

Different types of database testing focus on different aspects of database functionality. Understanding these types helps create a comprehensive testing strategy that covers all potential issues in your database implementation:

- **Structural Testing**: Verifies the database schema, tables, columns, indexes, etc.
- **Functional Testing**: Validates that database operations (CRUD) work as expected
- **Integration Testing**: Ensures that application and database work together correctly
- **Performance Testing**: Checks response time and resource usage for database operations
- **Security Testing**: Verifies data protection and access controls

Why is Database Testing Important?

A robust database testing strategy provides several key benefits that contribute to the overall quality and reliability of your application:

- **Data Integrity**: Ensures data is accurate and consistent throughout its lifecycle
- **Business Logic Validation**: Verifies that business rules are enforced at the data level
- **Error Detection**: Identifies issues with database operations before deployment
- **Performance Optimization**: Helps identify and resolve performance bottlenecks
- **Regression Prevention**: Ensures database changes don't break existing functionality

SQL Fundamentals for Testing

Effective database testing requires a solid understanding of SQL. In testing contexts, SQL is used not just for querying data but also for setting up test environments, validating database state, and cleaning up after tests. The following sections cover essential SQL techniques specifically useful for testing scenarios.

Basic SQL Operations for Testing

In database testing, you'll frequently use SQL to prepare your test environment, execute the operations you're testing, and verify the results. Mastering these basic operations is fundamental to effective database testing.

SELECT Statements

SQL SELECT statements are fundamental for validating data in your tests. They allow you to query and verify that your database contains the expected data after operations. You can use simple queries for direct validation or complex joins to verify relationships between tables.

```sql
-- Basic SELECT statement
SELECT * FROM Users WHERE Id = 1;

-- Aggregate functions for verification
SELECT COUNT(*) FROM Orders WHERE CustomerID = 101;

-- Joining tables for complex validation
SELECT u.Username, o.OrderDate, o.TotalAmount
FROM Users u
JOIN Orders o ON u.Id = o.UserId
WHERE u.Email = 'test@example.com';
```

Data Manipulation Language (DML)

DML statements are essential for setting up test data (INSERT), modifying it during tests (UPDATE), and cleaning up afterward (DELETE). These statements help you create a controlled test environment with precisely the data your tests need.

```sql
-- Inserting test data
INSERT INTO Users (Username, Email, CreatedDate)
```

```
VALUES ('testuser', 'test@example.com', GETDATE());

-- Updating test data
UPDATE Users
SET Status = 'Active'
WHERE Username = 'testuser';

-- Deleting test data
DELETE FROM Users WHERE Username LIKE 'test%';
```

Transaction Control

Transactions are critical for test isolation. They allow you to wrap a series of database operations in a transaction that can be rolled back after the test completes, ensuring that your tests don't affect each other and the database returns to its previous state.

```
-- Beginning a transaction
BEGIN TRANSACTION;

-- Insert test data
INSERT INTO Orders (UserId, OrderDate, TotalAmount) VALUES (1, GETDATE(), 99.99);
INSERT INTO OrderItems (OrderId, ProductId, Quantity, Price) VALUES
(SCOPE_IDENTITY(), 123, 2, 49.99);

-- Verify the data or roll back
IF @@ERROR <> 0
    ROLLBACK TRANSACTION;
ELSE
    COMMIT TRANSACTION;
```

Advanced SQL for Test Data Management

Beyond basic operations, advanced SQL techniques can make your database tests more robust and maintainable. These techniques help you manage test data more effectively and create more isolated test environments.

Temporary Tables

Temporary tables provide an isolated workspace for your tests. They exist only for the duration of your connection and allow you to create and manipulate test data without affecting your actual database tables. This is particularly useful for complex test scenarios.

```
-- Creating temporary tables for test data
CREATE TABLE #TempUsers (
    Id INT IDENTITY(1,1),
    Username NVARCHAR(100),
    Email NVARCHAR(255)
);
```

```
-- Using temporary tables in tests
INSERT INTO #TempUsers (Username, Email) VALUES ('test1', 'test1@example.com');
INSERT INTO #TempUsers (Username, Email) VALUES ('test2', 'test2@example.com');

-- Join with actual tables
SELECT t.Username, u.Status
FROM #TempUsers t
LEFT JOIN Users u ON t.Email = u.Email;

-- Clean up
DROP TABLE #TempUsers;
```

Stored Procedures for Test Setup

Creating dedicated stored procedures for test data setup can make your tests more maintainable and efficient. These procedures can generate complex test data sets with a single call, ensuring consistent test environments across test runs.

```
-- Creating a stored procedure for test data setup
CREATE PROCEDURE SetupTestData
    @userCount INT
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @i INT = 0;
    WHILE @i < @userCount
    BEGIN
        INSERT INTO Users (Username, Email, CreatedDate)
        VALUES (CONCAT('testUser', @i), CONCAT('test', @i, '@example.com'),
GETDATE());

        SET @i = @i + 1;
    END
END;

-- Using the stored procedure
EXEC SetupTestData @userCount = 10;
```

Data Cleanup Scripts

Proper cleanup is essential for maintaining test independence. Cleanup procedures ensure that test data from one test doesn't affect subsequent tests. These scripts should delete data in the proper order to respect foreign key constraints.

```
-- Creating cleanup script for test data
CREATE PROCEDURE CleanupTestData
```

```sql
AS
BEGIN
    SET NOCOUNT ON;

    -- Delete in proper order (respecting foreign keys)
    DELETE FROM OrderItems WHERE OrderId IN (SELECT Id FROM Orders WHERE UserId IN
(SELECT Id FROM Users WHERE Email LIKE 'test%@example.com'));
    DELETE FROM Orders WHERE UserId IN (SELECT Id FROM Users WHERE Email LIKE
'test%@example.com');
    DELETE FROM Users WHERE Email LIKE 'test%@example.com';
END;

-- Using the cleanup procedure
EXEC CleanupTestData;
```

Automated Database Tests in .NET Core

Automating database tests in .NET Core involves integrating database operations with testing frameworks like NUnit or xUnit. This section covers approaches to structuring and implementing database tests in a .NET Core environment, focusing on practical patterns that ensure reliable and maintainable tests.

Setting Up Database Test Projects

A well-structured test project makes database testing more manageable. This section covers how to organize your test project and set up the necessary infrastructure for database testing.

Project Structure

A well-organized project structure helps maintain separation of concerns and makes your tests more maintainable. This example shows a typical structure for a .NET Core application with database testing, separating core domain models, data access, and different types of tests.

```
DatabaseTestingDemo/
├── src/
│   ├── DatabaseTestingDemo.Core/ (domain models, interfaces)
│   └── DatabaseTestingDemo.Data/ (EF Core context, repositories)
└── tests/
    └── DatabaseTestingDemo.Tests/
        ├── Integration/ (database integration tests)
        │   ├── RepositoryTests.cs
        │   └── DatabaseFixture.cs (test data setup)
        ├── Unit/ (unit tests with database mocking)
        └── appsettings.json (test database config)
```

Test Project Setup

Setting up your test project correctly is crucial for database testing. This setup configures the test environment, including loading configuration, setting up dependency injection, and registering the database context and repositories that will be used across your tests.

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using NUnit.Framework;
using System.IO;

namespace DatabaseTestingDemo.Tests.Integration
{
    [SetUpFixture]
    public class TestSetup
    {
        public static IConfiguration Configuration { get; private set; }
        public static IServiceProvider ServiceProvider { get; private set; }

        [OneTimeSetUp]
        public void Setup()
        {
            // Load configuration from appsettings.json
            var builder = new ConfigurationBuilder()
                .SetBasePath(Directory.GetCurrentDirectory())
                .AddJsonFile("appsettings.json");

            Configuration = builder.Build();

            // Set up DI container
            var services = new ServiceCollection();

            // Register database context
            services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("TestDatabase")));

            // Register other services
            services.AddScoped<IUserRepository, UserRepository>();

            ServiceProvider = services.BuildServiceProvider();
        }
    }
}
```

Database Validation Tests

Database validation tests verify that your data access code correctly interacts with the database and that database operations produce the expected results. These tests typically focus on repository implementations, stored procedures, and direct SQL operations.

Repository Tests

Repository tests verify that your data access layer correctly interacts with the database. These tests check that your repository methods can perform CRUD operations correctly, ensuring data is properly stored, retrieved, updated, and deleted from the database.

```
using NUnit.Framework;
using Microsoft.Extensions.DependencyInjection;
using System.Threading.Tasks;
using DatabaseTestingDemo.Core.Models;
using DatabaseTestingDemo.Data.Repositories;

namespace DatabaseTestingDemo.Tests.Integration
{
    [TestFixture]
    public class UserRepositoryTests
    {
        private IUserRepository _userRepository;
        private AppDbContext _dbContext;

        [SetUp]
        public void SetUp()
        {
            // Get a fresh scoped instance for each test
            var scope = TestSetup.ServiceProvider.CreateScope();
            _dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();
            _userRepository =
scope.ServiceProvider.GetRequiredService<IUserRepository>();

            // Clear test data and set up initial state
            _dbContext.Users.RemoveRange(_dbContext.Users);
            _dbContext.SaveChanges();
        }

        [Test]
        public async Task GetUserById_ReturnsCorrectUser()
        {
            // Arrange
            var user = new User
            {
                Username = "testuser",
                Email = "test@example.com"
            };
            await _dbContext.Users.AddAsync(user);
            await _dbContext.SaveChangesAsync();

            // Act
            var result = await _userRepository.GetByIdAsync(user.Id);

            // Assert
            Assert.That(result, Is.Not.Null);
            Assert.That(result.Username, Is.EqualTo("testuser"));
            Assert.That(result.Email, Is.EqualTo("test@example.com"));
        }
        [Test]
        public async Task CreateUser_SavesUserToDatabase()
        {
            // Arrange
            var user = new User
            {
                Username = "newuser",
                Email = "new@example.com"
            };

            // Act
            await _userRepository.AddAsync(user);
            await _userRepository.SaveChangesAsync();

            // Assert - Verify directly from context
            var savedUser = await _dbContext.Users.FindAsync(user.Id);
            Assert.That(savedUser, Is.Not.Null);
            Assert.That(savedUser.Username, Is.EqualTo("newuser"));
        }
    }
}
```

Stored Procedure Tests

Testing stored procedures is important to verify that your database logic works correctly. These tests set up the necessary test data, execute the stored procedure, and then verify the results match the expected output, ensuring your stored procedures handle data properly.

```
using NUnit.Framework;
using Microsoft.EntityFrameworkCore;
using System.Data;
using System.Data.SqlClient;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;

namespace DatabaseTestingDemo.Tests.Integration
{
    [TestFixture]
    public class StoredProcedureTests
    {
        private string _connectionString;

        [SetUp]
        public void SetUp()
        {
            _connectionString =
TestSetup.Configuration.GetConnectionString("TestDatabase");
        }

        [Test]
        public async Task GetUserOrders_ReturnsCorrectOrderCount()
        {
            // Arrange - Set up test data
            await SetupTestUserWithOrders();

            // Act - Call stored procedure
            var orderCount = await GetOrderCountFromProcedure(1); // UserId = 1

            // Assert
            Assert.That(orderCount, Is.EqualTo(3));
        }

        private async Task SetupTestUserWithOrders()
        {
            using (var connection = new SqlConnection(_connectionString))
            {
                await connection.OpenAsync();

                // First clean any existing test data
                using (var command = new SqlCommand("DELETE FROM Orders WHERE
UserId = 1; DELETE FROM Users WHERE Id = 1;", connection))
                {
                    await command.ExecuteNonQueryAsync();
                }

                // Insert test user
                using (var command = new SqlCommand("INSERT INTO Users (Id,
Username, Email) VALUES (1, 'testuser', 'test@example.com')", connection))
                {
                    await command.ExecuteNonQueryAsync();
                }

                // Insert test orders
                using (var command = new SqlCommand(@"
                    INSERT INTO Orders (UserId, OrderDate, TotalAmount) VALUES (1,
GETDATE(), 100);
                    INSERT INTO Orders (UserId, OrderDate, TotalAmount) VALUES (1,
GETDATE(), 200);
                    INSERT INTO Orders (UserId, OrderDate, TotalAmount) VALUES (1,
GETDATE(), 300);", connection))
                {
                    await command.ExecuteNonQueryAsync();
                }
            }
        }

        private async Task<int> GetOrderCountFromProcedure(int userId)
        {
            using (var connection = new SqlConnection(_connectionString))
            {
                await connection.OpenAsync();

                using (var command = new SqlCommand("GetUserOrderCount",
connection))
```

```
                {
                    command.CommandType = CommandType.StoredProcedure;
                    command.Parameters.Add(new SqlParameter("@UserId", userId));

                    var result = await command.ExecuteScalarAsync();
                    return (int)result;
                }
            }
        }
    }
}
```

Direct SQL Tests

Sometimes you need to test database features directly with SQL, especially for schema validation or when testing triggers and other database objects. These tests use raw SQL queries to verify database behavior beyond what is exposed through your application's data layer.

```
using NUnit.Framework;
using System.Data.SqlClient;
using System.Threading.Tasks;
using Dapper;

namespace DatabaseTestingDemo.Tests.Integration
{
    [TestFixture]
    public class DirectSqlTests
    {
        private string _connectionString;

        [SetUp]
        public void SetUp()
        {
            _connectionString =
TestSetup.Configuration.GetConnectionString("TestDatabase");
        }

        [Test]
        public async Task UserTable_HasCorrectSchema()
        {
            // Act
            var columns = await GetTableColumns("Users");

            // Assert
            Assert.That(columns, Contains.Item("Id"));
            Assert.That(columns, Contains.Item("Username"));
            Assert.That(columns, Contains.Item("Email"));
            Assert.That(columns, Contains.Item("CreatedDate"));
        }

        [Test]
        public async Task InsertUser_TriggersAuditLog()
        {
            // Arrange - Clean up test data
            using (var connection = new SqlConnection(_connectionString))
            {
                await connection.ExecuteAsync("DELETE FROM AuditLogs WHERE
EntityName = 'Users' AND Action = 'INSERT'");
                await connection.ExecuteAsync("DELETE FROM Users WHERE Email =
'audit@example.com'");
            }

            // Act - Insert user (should trigger audit log via database trigger)
            using (var connection = new SqlConnection(_connectionString))
            {
                await connection.ExecuteAsync(
                    "INSERT INTO Users (Username, Email, CreatedDate) VALUES
(@Username, @Email, GETDATE())",
                    new { Username = "audituser", Email = "audit@example.com" });
            }

            // Assert - Check audit log was created
            using (var connection = new SqlConnection(_connectionString))
            {
                var auditCount = await connection.ExecuteScalarAsync<int>(
                    "SELECT COUNT(*) FROM AuditLogs WHERE EntityName = 'Users' AND
Action = 'INSERT'");
                Assert.That(auditCount, Is.GreaterThan(0));
            }
        }

        private async Task<List<string>> GetTableColumns(string tableName)
        {
            using (var connection = new SqlConnection(_connectionString))
            {
                var query = @"
                    SELECT COLUMN_NAME
                    FROM INFORMATION_SCHEMA.COLUMNS
                    WHERE TABLE_NAME = @TableName";

                var columns = await connection.QueryAsync<string>(query, new {
TableName = tableName });
                return columns.ToList();
            }
        }
    }
}
```

Entity Framework Core in Testing

Entity Framework Core provides an object-relational mapping (ORM) layer that simplifies database operations in .NET applications. When testing with EF Core, you need strategies for creating test data, managing test isolation, and verifying that your EF Core context and entities function correctly.

Test Data Setup with EF Core

Setting up test data with EF Core involves creating a database context configured for testing and populating it with the data needed for your tests. This section covers approaches to creating and managing test data with EF Core.

Creating a Test Database Context

A dedicated test database context allows you to control the database environment for your tests. This example shows how to create a fixture that sets up either an in-memory database or a real SQL Server database with test data, ensuring a consistent starting point for your tests.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using System;

namespace DatabaseTestingDemo.Tests.Integration
{
    public class DatabaseFixture : IDisposable
    {
        public AppDbContext Context { get; private set; }

        public DatabaseFixture()
        {
            // Create a new service provider for each test class
            var serviceProvider = new ServiceCollection()
                .AddDbContext<AppDbContext>(options =>
                {
                    // Use an in-memory database for testing
                    options.UseInMemoryDatabase("TestDatabase_" +
Guid.NewGuid().ToString());

                    // Or use SQL Server with a unique database name for isolation
                    // options.UseSqlServer("Server=
(localdb)\\mssqllocaldb;Database=TestDb_" + Guid.NewGuid().ToString() +
";Trusted_Connection=True;");
                })
                .BuildServiceProvider();

            // Get the context
            Context = serviceProvider.GetRequiredService<AppDbContext>();

            // Ensure database is created
            Context.Database.EnsureCreated();

            // Seed with test data
            SeedDatabase();
        }

        private void SeedDatabase()
        {
            // Add users
            var users = new List<User>
            {
                new User { Username = "user1", Email = "user1@example.com",
CreatedDate = DateTime.Now },
                new User { Username = "user2", Email = "user2@example.com",
CreatedDate = DateTime.Now }
            };
            Context.Users.AddRange(users);

            // Add orders
            var orders = new List<Order>
            {
                new Order { UserId = 1, OrderDate = DateTime.Now, TotalAmount =
100.00m },
                new Order { UserId = 1, OrderDate = DateTime.Now, TotalAmount =
150.00m },
                new Order { UserId = 2, OrderDate = DateTime.Now, TotalAmount =
200.00m }
            };
            Context.Orders.AddRange(orders);

            Context.SaveChanges();
        }

        public void Dispose()
        {
            Context.Database.EnsureDeleted();
            Context.Dispose();
        }
    }
}
```

Using the Database Fixture in Tests

The database fixture pattern allows you to share a database context across related tests. This approach is efficient because it avoids recreating the database for each test while still providing isolation. The example shows how to use a shared fixture in a test class.

```
using NUnit.Framework;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace DatabaseTestingDemo.Tests.Integration
{
    [TestFixture]
    public class UserServiceTests : IClassFixture<DatabaseFixture>
    {
        private readonly AppDbContext _context;
        private readonly UserService _userService;
        public UserServiceTests(DatabaseFixture fixture)
        {
            _context = fixture.Context;
            _userService = new UserService(_context);
        }

        [Test]
        public async Task GetActiveUsers_ReturnsOnlyActiveUsers()
        {
            // Arrange - Set some users as inactive
            var user = await _context.Users.FirstAsync();
            user.Status = "Inactive";
            await _context.SaveChangesAsync();

            // Act
            var activeUsers = await _userService.GetActiveUsersAsync();

            // Assert
            Assert.That(activeUsers, Is.Not.Null);
            Assert.That(activeUsers.Count(), Is.EqualTo(1));
            Assert.That(activeUsers.All(u => u.Status != "Inactive"), Is.True);
        }
    }
}
```

Database Transaction Management

Proper transaction management is crucial for test isolation when working with real databases. This section covers techniques for using transactions to prevent test interference and ensure consistent database state.

Using Transaction Scope

Using TransactionScope is a powerful way to ensure test isolation. It automatically wraps your test operations in a transaction that is rolled back when the scope is disposed, regardless of whether the test passes or fails. This ensures your tests don't affect each other or the database state.

```
using NUnit.Framework;
using System.Threading.Tasks;
using System.Transactions;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace DatabaseTestingDemo.Tests.Integration
{
    [TestFixture]
    public class TransactionTests
    {
        private AppDbContext _context;
        private IUserRepository _repository;

        [SetUp]
        public void SetUp()
        {
            var serviceProvider = TestSetup.ServiceProvider;
            _context = serviceProvider.GetRequiredService<AppDbContext>();
            _repository = serviceProvider.GetRequiredService<IUserRepository>();
        }

        [Test]
        public async Task UserRepository_OperationsRolledBack_AfterTestExecution()
        {
            // Count users before test
            var userCountBefore = await _context.Users.CountAsync();

            // Use TransactionScope to automatically roll back after test
            using (var transaction = new
TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
            {
                // Arrange
                var user = new User
                {
                    Username = "transactiontest",
                    Email = "transaction@example.com"
                };

                // Act
                await _repository.AddAsync(user);
                await _repository.SaveChangesAsync();

                // Verify user was added inside transaction
                var userExists = await _context.Users.AnyAsync(u => u.Email ==
"transaction@example.com");
                Assert.That(userExists, Is.True);

                // Don't commit - allows automatic rollback
                // transaction.Complete();
            }

            // Assert user count is the same after transaction rolled back
            var userCountAfter = await _context.Users.CountAsync();
            Assert.That(userCountAfter, Is.EqualTo(userCountBefore));

            // Verify user doesn't exist after rollback
            var userStillExists = await _context.Users.AnyAsync(u => u.Email ==
"transaction@example.com");
            Assert.That(userStillExists, Is.False);
        }
    }
}
```

Data Setup and Teardown

Effective test data management includes both setting up the data needed for tests and cleaning it up afterward. This section explores patterns for managing test data lifecycle in EF Core tests.

Test Data Builders

The builder pattern is an elegant way to create test data. It provides a fluent interface for constructing complex objects and can set reasonable defaults while allowing you to override specific properties. This makes your test data creation both flexible and readable.

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

namespace DatabaseTestingDemo.Tests.Utils
{
    // Builder for creating test users
    public class UserBuilder
    {
        private readonly User _user = new User();

        public UserBuilder WithId(int id)
        {
            _user.Id = id;
            return this;
        }

        public UserBuilder WithUsername(string username)
        {
            _user.Username = username;
            return this;
        }

        public UserBuilder WithEmail(string email)
        {
            _user.Email = email;
            return this;
        }

        public UserBuilder WithStatus(string status)
        {
            _user.Status = status;
            return this;
        }

        public UserBuilder AsActive()
        {
            _user.Status = "Active";
            return this;
        }

        public UserBuilder AsInactive()
        {
            _user.Status = "Inactive";
            return this;
        }

        public User Build()
        {
            // Set default values if not specified
            if (string.IsNullOrEmpty(_user.Username))
                _user.Username = "user" + Guid.NewGuid().ToString().Substring(0,
8);

            if (string.IsNullOrEmpty(_user.Email))
                _user.Email = _user.Username + "@example.com";

            if (string.IsNullOrEmpty(_user.Status))
                _user.Status = "Active";

            if (_user.CreatedDate == default)
                _user.CreatedDate = DateTime.Now;

            return _user;
        }

        // Create a list of users
        public static List<User> CreateMany(int count, Action<UserBuilder, int>
customize = null)
        {
            var users = new List<User>();
            for (int i = 0; i < count; i++)
            {
                var builder = new UserBuilder();
                customize?.Invoke(builder, i);
                users.Add(builder.Build());
            }
            return users;
        }
    }
}
```

Using the Builders in Tests

Incorporating data builders into your tests makes them more readable and maintainable. This example shows how to use the builder pattern to create a collection of test users with different statuses, demonstrating how builders can make complex test data setup concise and clear.

```
using NUnit.Framework;
using System.Linq;
using System.Threading.Tasks;
using DatabaseTestingDemo.Tests.Utils;

namespace DatabaseTestingDemo.Tests.Integration
{
    [TestFixture]
    public class DataBuilderTests
    {
        private AppDbContext _context;

        [SetUp]
        public void SetUp()
        {
            _context = TestSetup.ServiceProvider.GetRequiredService<AppDbContext>
();

            // Clear existing data
            _context.Users.RemoveRange(_context.Users);
            _context.SaveChanges();
        }

        [Test]
        public async Task UserService_FindsUsersByStatus()
        {
            // Arrange - Create test data with builders
            var users = UserBuilder.CreateMany(5, (builder, i) =>
            {
                if (i < 3)
                    builder.AsActive();
                else
                    builder.AsInactive();
            });

            _context.Users.AddRange(users);
            await _context.SaveChangesAsync();

            var userService = new UserService(_context);

            // Act
            var activeUsers = await userService.GetUsersByStatusAsync("Active");

            // Assert
            Assert.That(activeUsers.Count(), Is.EqualTo(3));
        }
    }
}
```

Best Practices

1. **Use a Dedicated Test Database**: Never test against production databases.
2. **Clean Up Test Data**: Always clean up test data after tests run to prevent test interference.
3. **Control Test Data**: Use builders or fixtures to create consistent, reusable test data.
4. **Use Transactions for Isolation**: Wrap tests in transactions to prevent test pollution.

5. **Test Against Real Databases**: In-memory databases don't fully simulate real database behavior, especially for complex queries or stored procedures.
6. **Focus on Business Requirements**: Test database functionality against business requirements, not implementation details.
7. **Avoid Hardcoded Connection Strings**: Use configuration files for database connection strings.
8. **Test Database Schema Changes**: Create tests that verify schema changes don't break existing functionality.
9. **Use Database Snapshots**: For SQL Server, consider using database snapshots for test data restoration.
10. **Performance Test Critical Queries**: Include performance testing for critical database operations.

Resources

- Entity Framework Core Documentation
- SQL Server Documentation
- NUnit Documentation
- Dapper GitHub Repository
- TestContainers for .NET
- SQL Server Database Testing Best Practices

This guide is meant as a reference. Adapt the examples to your specific database testing requirements and project structure.