**Section Cheat Sheet (PPT)**

Best Practices of Unit Tests

**Isolated / Stand-alone**

(separated from any other dependencies such as file system or database)

**Test single method at-a-time**

(should not test more than one method in a single test case)

**Unordered**

(can be executed in any order)

**Fast**

(Tests should take little time to run (about few milliseconds))

**Repeatable**

(Tests can run repeatedly but should give same result, if no changes in the actual source code)

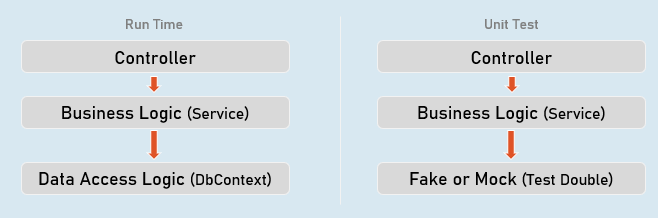
**Timely**

(Time taken for writing a test case should not take longer time, than then time taken for writing the code that is being tested)

Mocking the DbContext

**Test Double**

A "test double" is an object that look and behave like their production equivalent objects.



A "test double" is an object that look and behave like their production equivalent objects.

**Fake**

An object that providers an alternative (dummy) implementation of an interface

**Mock**

An object on which you fix specific return value for each individual method or property, without actual / full implementation of it.

**Mocking the DbContext**

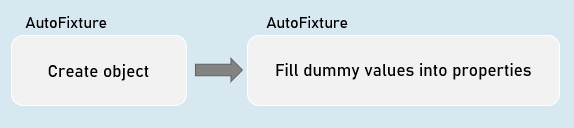
1. Install-Package Moq
2. Install-Package EntityFrameworkCoreMock.Moq

**Mocking the DbContext:**

1. var dbContextOptions = new DbContextOptionsBuilder<DbContextClassName>().Options;
3. //mock the DbContext
4. DbContextMock<DbContextClass> dbContextMock = new DbContextMock<DbContextClass>(dbContextOptions);
5. var initialData = new List<ModelClass>() { … };
7. //mock the DbSet
8. var dbSetMock = dbContextMock.CreateDbSetMock(temp => temp.DbSetName, initialData);
10. //create service instance with mocked DbContext
11. var service = newServiceClass(dbContextMock.Object);

AutoFixture

AutoFixture generates objects of the specified classes and their properties with some fake values based their data types.



**Normal object creation**

1. new ModelClass() {
2. Property1 = value,
3. Property2 = value
4. }

**With AutoFixture**

Fixture.Create<ModelClass>(); //initializes all properties of the specified model class with dummy values

**AutoFixture**

Install-Package AutoFixture

**Working with AutoFixture:**

1. var fixture = new Fixture();
3. //Simple AutoFixture
4. var obj1 = fixture.Create<ModelClass>();
6. //Customization with AutoFixture
7. var obj2 = fixture.Build<ModelClass>()
8. .With(temp => temp.Property1, value)
9. .With(temp => temp.Property2, value)
10. .Create();

Fluent Assertions

Fluent Assertions are a set of extension methods to make the assertions in unit testing more readable and human-friendly.

Install-Package FluentAssertions

**Assert**

1. //Equal
2. Assert.Equal(expected, actual);
4. //Not Equal
5. Assert.NotEqual(expected, actual);
7. //Null
8. Assert.Null(actual);
10. //Not Null
11. Assert.NotNull(actual);
13. //True
14. Assert.True(actual);
16. //False
17. Assert.False(actual);
19. //Empty
20. Assert.Empty(actual);
22. //Not Empty
23. Assert.NotEmpty(actual);
25. //Null or empty
26. Assert.True(string.IsNullOrEmpty(actual)); //string
27. Assert.True(actual == null || actual.Length == 0); //collection
29. //Should not be null or empty
30. Assert.False (string.IsNullOrEmpty(actual)); //string
31. Assert.False(actual == null || actual.Length == 0); //collection
33. //number should be positive
34. Assert.True(actual > 0);
36. //number should be negative
37. Assert.True(actual < 0);
39. //number should be >= expected
40. Assert.True(actual >= expected);
42. //number should be <= expected
43. Assert.True(actual <= expected);
45. //number should be in given range
46. Assert.True(actual >= minimum && actual <= maximum);
48. //number should not be in given range
49. Assert.True(actual < minimum || actual > maximum);
51. //check data type
52. Assert.IsType<ExpectedType>(actual);
54. //Compare properties of two objects (Equals method SHOULD BE overridden)
55. Assert.Equal(expected, actual);
57. //Compare properties (should not be equal) of two objects (Equals method SHOULD BE overridden)
58. Assert.NotEqual(expected, actual);

**Fluent Assertion**

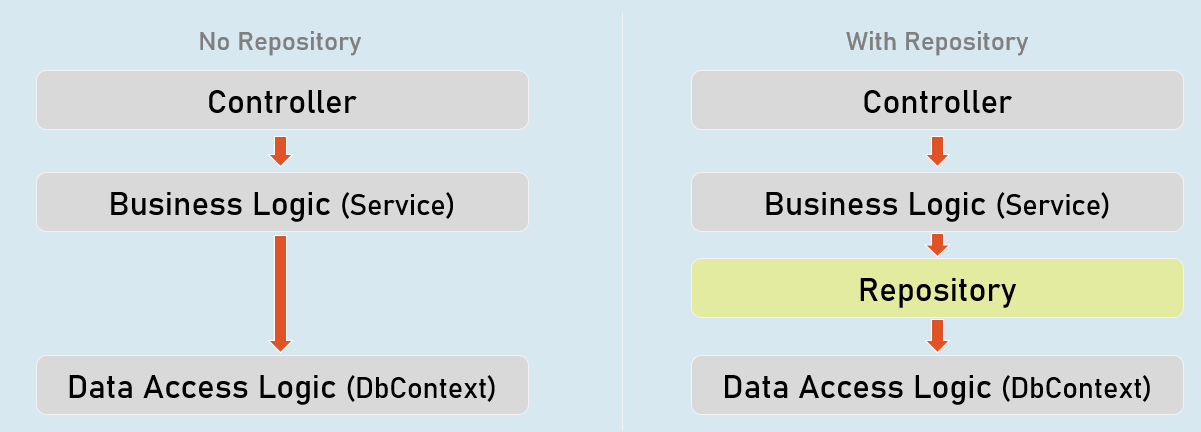
1. //Equal
2. actual.Should().Be(expected);
4. //Not Equal
5. actual.Should().NotBe(expected);
7. //Null
8. actual.Should().BeNull();
10. //Not Null
11. actual.Should().NotBeNull();
13. //True
14. actual.Should().BeTrue();
16. //False
17. actual.Should().BeFalse();
19. //Empty
20. actual.Should().BeEmpty();
22. //Not Empty
23. actual.Should().NotBeEmpty();
25. //Null or empty
26. actual.Should().BeNullOrEmpty();
28. //Should not be null or empty
29. actual.Should().NotBeNullOrEmpty();
31. //number should be positive
32. actual.Should().BePositive();
34. //number should be negative
35. actual.Should().BeNegative();
37. //number should be >= expected
38. actual.Should().BeGreaterThanOrEqualTo(expected);
40. //number should be <= expected
41. actual.Should().BeLessThanOrEqualTo(expected);
43. //number should be in given range
44. actual.Should().BeInRange(minimum, maximum);
46. //number should not be in given range
47. actual.Should().NotBeInRange(minimum, maximum);
49. //number should be in given range
50. actual.Should().BeInRange(minimum, maximum);
52. //number should not be in given range
53. actual.Should().NotBeInRange(minimum, maximum);
55. //check data type (same type)
56. actual.Should().BeOfType<ExpectedType>();
58. //check data type (same type or derived type)
59. actual.Should().BeAssignableTo<ExpectedType>();
61. //Compare properties of two objects (Equals method NEED NOT be overridden)
62. actual.Should().BeEquivalentTo(expected);
64. //Compare properties (should not equal) of two objects (Equals method NEED NOT be overridden)
65. actual.Should().BeNotEquivalentTo(expected);

**Fluent Assertions - Collections:**

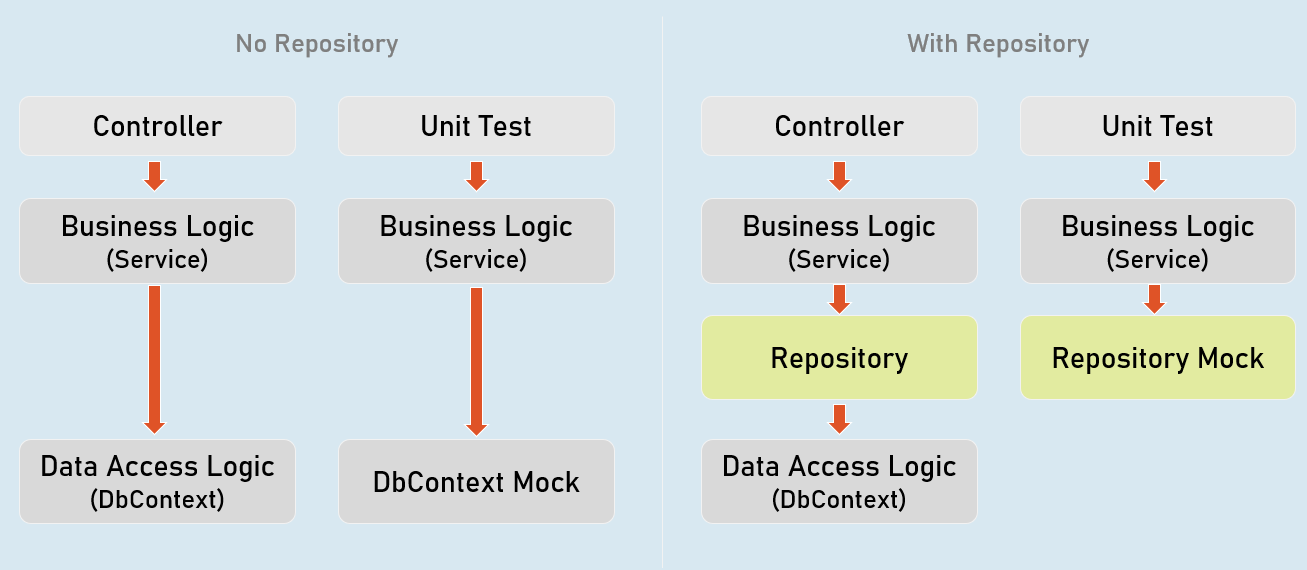
1. actualCollection.Should().BeEmpty();
2. actualCollection.Should().NotBeEmpty();
4. actualCollection.Should().HaveCount(expectedCount);
5. actualCollection.Should().NotHaveCount(expectedCount);
7. actualCollection.Should().HaveCountGreaterThanOrEqualTo(expectedCount);
8. actualCollection.Should().HaveCountLessThanOrEqualTo(expectedCount);
10. actualCollection.Should().HaveSameCount(expectedCollection);
11. actualCollection.Should().NotHaveSameCount(expectedCollection);
13. actualCollection.Should().BeEquivalentTo(expectedCollection);
14. actualCollection.Should().NotBeEquivalentTo(expectedCollection);
16. actualCollection.Should().ContainInOrder(expectedCollection);
17. actualCollection.Should().NotContainInOrder(expectedCollection);
19. actualCollection.Should().OnlyHaveUniqueItems(expectedCount);
20. actualCollection.Should().OnlyContain(temp => condition);
22. actualCollection.Should().BeInAscendingOrder(temp => temp.Property);
23. actualCollection.Should().BeInDescendingOrder(temp => temp.Property);
25. actualCollection.Should().NotBeInAscendingOrder(temp => temp.Property);
26. actualCollection.Should().NotBeInDescendingOrder(temp => temp.Property);
28. delegateObj.Should().Throw<ExceptionType>();
29. delegateObj.Should().NotThrow<ExceptionType>();
31. await delegateObj.Should().ThrowAsync<ExceptionType>();
32. await delegateObj.Should().NotThrowAsync<ExceptionType>();

Repository

Repository (or Repository Pattern) is an abstraction between Data Access Layer (EF DbContext) and business logic layer (Service) of the application.



**Unit Testing**



Benefits of Repository Pattern

**Loosely-coupled business logic (service) & data access.**

(You can independently develop them).

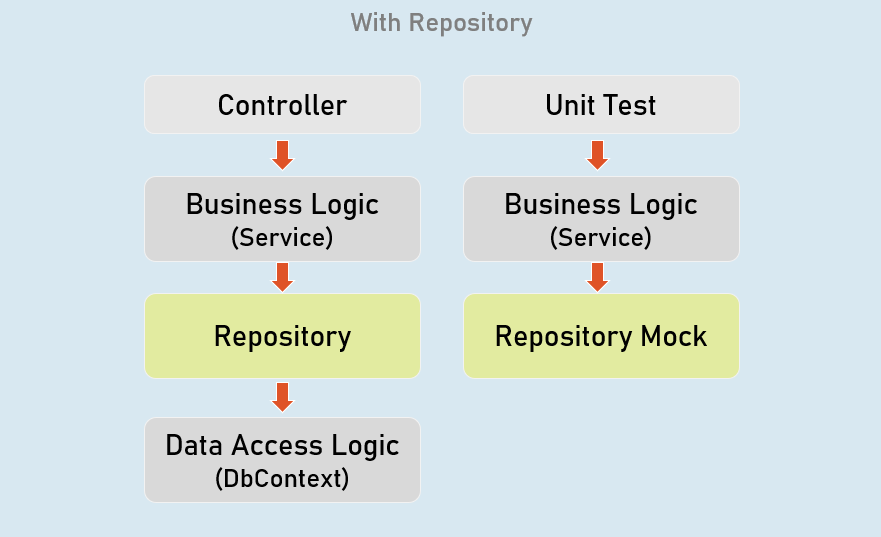
**Changing data store**

(You can create alternative repository implementation for another data store, when needed).

**Unit Testing**

(Mocking the repository is much easier (and preferred) than mocking DbContext).

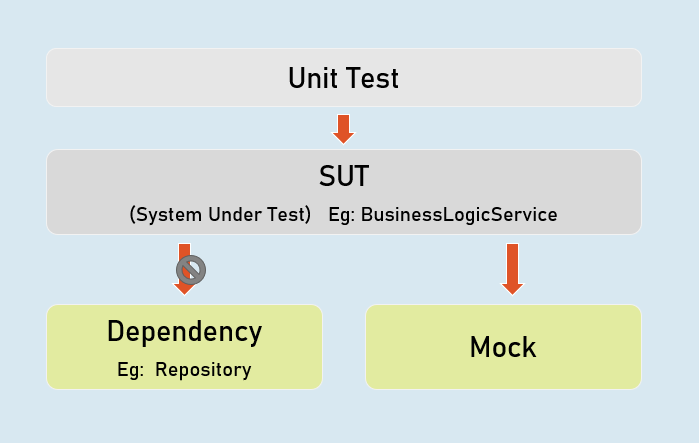
Mocking the Repository



Install-Package Moq

**Mocking the Repository:**

1. //mock the repository
2. Mock<IRepository> repositoryMock = new Mock<IRepository>();
4. //mock a method repository method
5. repositoryMock.Setup(temp => temp.MethodName(It.Any<ParameterType>()))
6. .Returns(return\_value);
8. //create service instance with mocked repository
9. var service = newServiceClass(repositoryMock.Object);



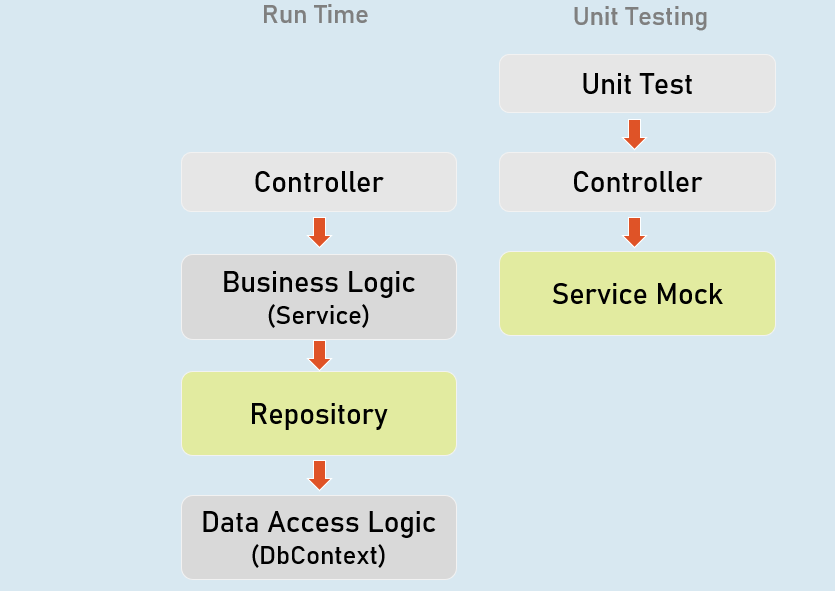
**Mock<IPersonsRepository>**

Used to mock the methods of IPersonsRepository.

**IPersonsRepository**

Represents the mocked object that was created by Mock<T>.

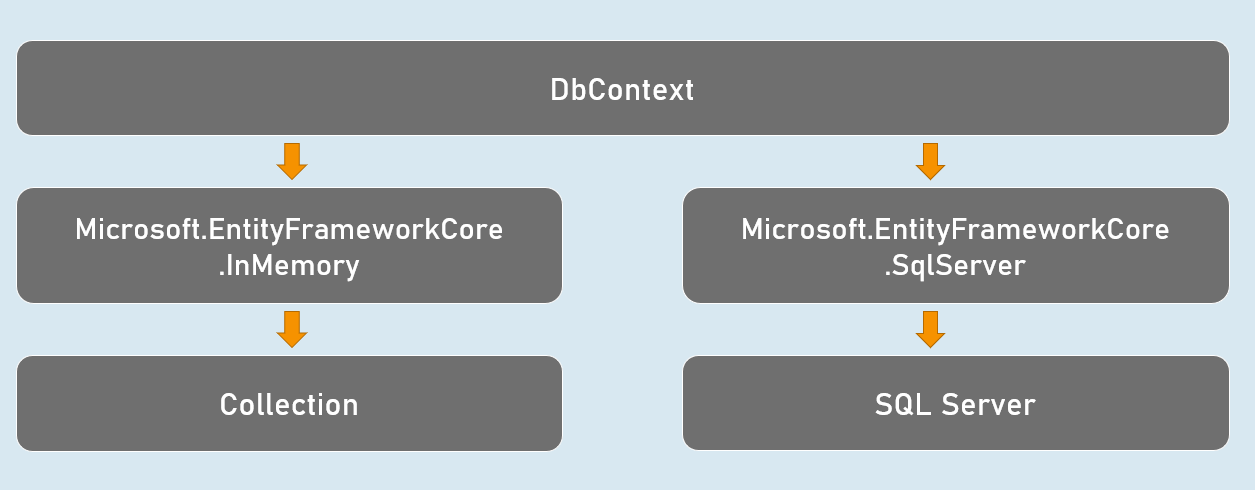
Unit Testing the Controller



**Unit Testing the Controller:**

1. //Arrange
2. ControllerName controller = new ControllerName();
4. //Act
5. IActionResult result = controller.ActionMethod();
7. //Assert
8. result.Should().BeAssignableTo<ActionResultType>(); //checking type of action result
9. result.ViewData.Model.Should().BeAssignableTo<ExpectedType>(); //checking type of model
10. result.ViewData.Model.Should().Be(expectedValue); //you can also use any other assertion

EFCore In-Memory Provider

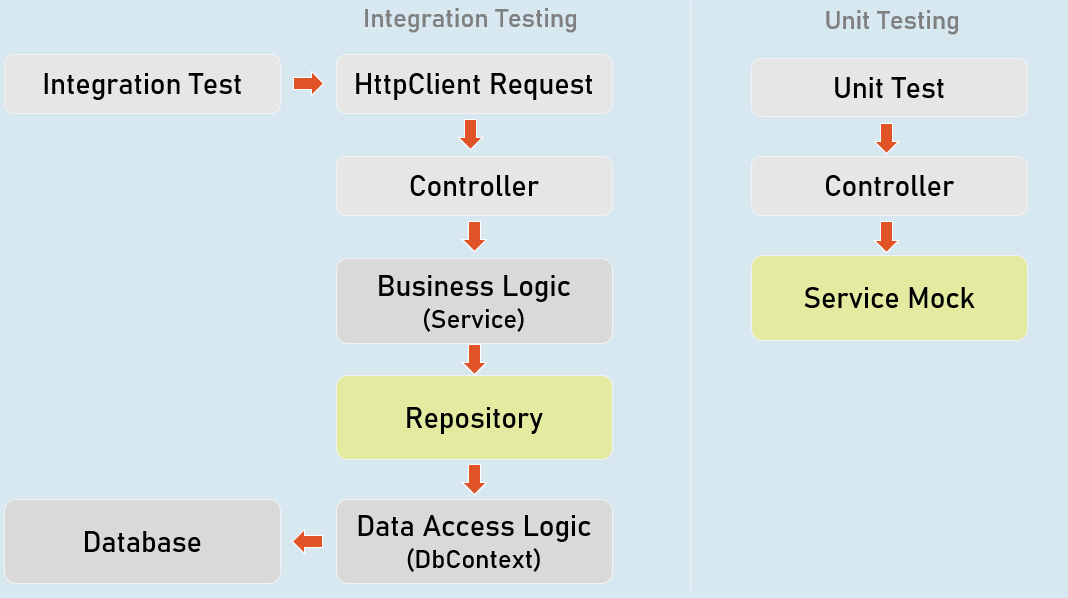


Install-Package Microsoft.EntityFrameworkCore.InMemory

**Using In-memory provider:**

1. var dbContextOptions =
2. new DbContextOptionsBuilder<DbContextClassName>()
3. .UseInMemoryDatabase("database\_name");
4. .Options;
6. var dbContext = newDbContextClassName(dbContextOptions);

Integration Test



1. //Create factory
2. WebApplicationFactory factory = new WebApplicationFactory();
4. //Create client
5. HttpClient client = factory.CreateClient();
7. //Send request client
8. HttpResponseMessage response = await client.GetAsync("url");
10. //Assert
11. result.Should().BeSuccessful(); //Response status code should be 200 to 299