Manish Jaiswal

EF Core Series

MASTERING

Testing with Entity Framework Core

swipe

Manish Jaiswal

EF Core Series

01

Why Testing is Crucial ?

Testing ensures your Entity Framework Core applications are reliable, maintainable, and free of bugs. Proper testing strategies can save time and resources in the long run.

swipe

Manish Jaiswal

EF Core Series

02

Unit Testing EF Core Applications

Unit testing focuses on testing **individual components** in isolation. For EF Core, it means testing your repository or service logic without hitting the actual database.

swipe

Manish Jaiswal

EF Core Series

03

Example

```
1  public class ProductServiceTests
2  {
3      private readonly ProductService _service;
4      private readonly Mock<IProductRepository> _mockRepo;
5
6      public ProductServiceTests()
7      {
8          _mockRepo = new Mock<IProductRepository>();
9          _service = new ProductService(_mockRepo.Object);
10     }
11
12     [Fact]
13     public void GetProduct_ShouldReturnProduct_WhenProductExists()
14     {
15         var productId = 1;
16         _mockRepo.Setup(repo => repo.GetProduct(productId))
17                  .Returns(new Product
18                      { Id = productId,
19                        Name = "Test Product"
20                      });
21
22         var result = _service.GetProduct(productId);
23
24         Assert.NotNull(result);
25         Assert.Equal("Test Product", result.Name);
26     }
27 }
```

swipe

Manish Jaiswal

EF Core Series

04

Mocking DbContext for Tests

Mocking **DbContext** helps you simulate database interactions in your tests, ensuring your business logic is tested without relying on a real database.

swipe

Manish Jaiswal

EF Core Series

05

Example

```

public class DbContextMock
{
    public static Mock<DbSet<T>> GetDbSet<T>(List<T> entities) where T : class
    {
        var queryable = entities.AsQueryable();
        var dbSet = new Mock<DbSet<T>>();

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.Provider).Returns(queryable.Provider);

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.Expression).Returns(queryable.Expression);

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.ElementType).Returns(queryable.ElementType);

        dbSet.As<IQueryable<T>>()
        .Setup(m => m.GetEnumerator()).Returns(queryable.GetEnumerator());
        return dbSet;
    }
}
```

swipe

Manish Jaiswal

EF Core Series

06

Integration Testing with In-Memory Databases

Integration tests verify that different parts of the application work together. Using in-memory databases like InMemoryDatabase ensures your tests run fast and in isolation.

swipe

Manish Jaiswal

EF Core Series

```
public class ProductServiceIntegrationTests
{
    private readonly DbContextOptions<AppDbContext> _options;

    public ProductServiceIntegrationTests()
    {
        _options = new DbContextOptionsBuilder<AppDbContext>()
                    .UseInMemoryDatabase(databaseName: "TestDatabase")
                    .Options;
    }

    [Fact]
    public void GetProduct_ShouldReturnProduct_WhenProductExists()
    {
        using (var context = new AppDbContext(_options))
        {
            context.Products.Add(new Product
                                { Id = 1,
                                  Name = "Integration Test Product"
                                });
            context.SaveChanges();
        }

        using (var context = new AppDbContext(_options))
        {
            var service = new ProductService(context);
            var result = service.GetProduct(1);

            Assert.NotNull(result);
            Assert.Equal("Integration Test Product", result.Name);
        }
    }
}
```

swipe

Manish Jaiswal

EF Core Series

08

Best Practices for Testing with EF Core

- Use in-memory databases for integration tests.
- Mock DbContext for unit tests.
- Isolate tests to ensure reliability.
- Always clean up data after tests.

swipe