## **Unit Testing with NUnit and Moq in ASP.NET Core: A Comprehensive Guide**

Unit testing is paramount for building reliable software. It empowers developers to isolate and validate small, testable units of code, ensuring their correctness and safeguarding against unintended side effects during subsequent development cycles.

### **Understanding the API Under Test**

To illustrate our unit testing approach, we will focus on a straightforward API within an e-commerce context. This API, named CheckOut, resides within a CartController class. Its primary function is to process an Order object, which entails validating cart items, verifying payment details, and initiating the shipment process.

The CheckOut API method in our CartController is structured as follows:

| [HttpPost] public ActionResult<string> CheckOut(Order order) {  var result = \_cartService.ValidateCart(order);  return Ok(result); } |
| --- |

The Order object itself encapsulates vital transaction details, including card information, shipping address, and a list of items placed in the shopping cart.

The CartService class is central to our logic flow. Its responsibility is to perform initial cart validation. Upon successful validation, it delegates the payment processing task to an IPaymentService instance.

| public class CartService : ICartService {  IPaymentService \_paymentService;   public CartService(IPaymentService paymentService)  {  \_paymentService = paymentService;  }   public string ValidateCart(Order order)  {  if (order.CartItems.Count < 1)  return "Invalid Cart";  if (order.CartItems.Any(x => x.Quantity < 0 || x.Quantity > 10))  return "Invalid Product Quantity";   return \_paymentService.ChargeAndShip(order);  } } |
| --- |

The PaymentService, in turn, is responsible for validating card details and processing the payment. If the payment transaction is successful, it then proceeds to dispatch the ordered items via an IShipmentService.

| public class PaymentService : IPaymentService {  IShipmentService \_shipment;   public PaymentService(IShipmentService shipment)  {  \_shipment = shipment;  }   public string ChargeAndShip(Order order)  {  if (order.Card.Amount <= 0)  {  return "Amount Not Valid";  }  if (order.Card != null)  {  if (order.Card.ValidTo < DateTime.Now)  return "Card Expired";  if (order.Card.CardNumber.Length < 16)  return "CardNumber Not Valid";  }   bool paymentSuccess = MakePayment(order.Card); // This method involves an external call   if (paymentSuccess)  {  var shipment = \_shipment.Ship(order.Address); // This also involves an external call  if (shipment != null)  return "Item Shipped";  else  return "Something went wrong with the shipment!!!";  }  else  {  return "Payment Failed";  }  }   public virtual bool MakePayment(Card card)  {  // This method simulates a call to a third-party payment service provider.  return true; // Simplified for illustration  } } |
| --- |

### **Writing Tests for the Checkout API**

Our primary objective is to thoroughly test the ValidateCart method within the CartService class. Critically, we will employ **mocking** for dependencies such as PaymentService and ShipmentService, as their methods (MakePayment and Ship, respectively) represent interactions with external, third-party systems. By mocking these dependencies, we isolate the CartService's logic, ensuring that our unit tests truly focus on the CartService's behavior without relying on the actual external services.

#### **Setting Up the Services with Moq**

We will utilize NUnit for structuring our tests and Moq for creating mock objects of our dependencies. In this scenario, we will partially mock the PaymentService to control the outcome of its MakePayment method (which simulates a third-party call). We will entirely mock the ShipmentService, as its Ship method represents another external interaction.

Here is the setup for our test class, CartTest:

| using NUnit.Framework; using Moq; // Namespace for Moq using YourApiNamespace.Services; // Assuming your API services are here using YourApiNamespace.Models; // Assuming your API models are here using System; using System.Generic;  [TestFixture] public class CartTest {  private ICartService cartService;  private Mock<PaymentService> paymentServiceMock; // Mocking the concrete PaymentService  private Mock<IShipmentService> shipmentServiceMock; // Mocking the IShipmentService interface   [SetUp] // Executed before each test method  public void Setup()  {  // 1. Mock ShipmentService: This is a full mock of the interface.  // We use Mock<IShipmentService>() because we want to control all its behavior.  shipmentServiceMock = new Mock<IShipmentService>();   // 2. Partial Mock of PaymentService:  // We mock PaymentService, but crucially, we pass shipmentServiceMock.Object to its constructor.  // This ensures that when methods of PaymentService that ARE NOT mocked are called,  // they will use the mocked IShipmentService we provided.  // The PaymentService constructor requires IShipmentService, so we provide the mock object.  paymentServiceMock = new Mock<PaymentService>(shipmentServiceMock.Object);   // 3. Instantiate the Actual CartService:  // We are testing CartService, so we create a real instance.  // We inject the mock of PaymentService into CartService's constructor.  // This means CartService will interact with our controlled mock, not a real PaymentService.  cartService = new CartService(paymentServiceMock.Object);  } } |
| --- |

In this Setup method, Moq.Mock<T> is used to create mock instances. shipmentServiceMock is a complete mock, while paymentServiceMock is a *partial mock* of PaymentService. The constructor of Mock<PaymentService> accepts arguments that would typically be passed to the real PaymentService constructor, allowing us to control its dependencies. The cartService is an actual instance, but it receives its dependencies (the mocked paymentServiceMock) via dependency injection, enabling test isolation.

#### **Writing Test Cases with NUnit's [TestCase] Attribute**

Our objective is to test various scenarios for the ValidateCart method. NUnit's [TestCase] attribute is exceptionally useful for this, as it allows us to define multiple parameterized test scenarios within a single test method, significantly reducing code duplication and improving test readability. Each [TestCase] provides a distinct set of parameters that will be passed to the test method upon execution.

Here is the comprehensive test method designed to cover various validation and flow scenarios:

| [Test] // Marks this as a test method // Test Cases for ValidateCart scenarios: // Parameters: amount, cardNumber, validDate, quantity, paymentSuccess, shipmentSuccess, expectedResult [TestCase(-1, "4041000011114567", true, 1, true, true, "Amount Not Valid")] // Fail: amount < 0 [TestCase(10, "404100001111456", true, 2, true, true, "CardNumber Not Valid")] // Fail: card number invalid length [TestCase(12, "4041000011114561", false, 3, true, true, "Card Expired")] // Fail: card expiry date invalid [TestCase(11, "40410000111145610", true, 11, true, true, "Invalid Product Quantity")] // Fail: quantity > 10 [TestCase(5, "40410000111145610", true, 9, false, true, "Payment Failed")] // Fail: payment fails [TestCase(8, "40410000111145610", true, 9, true, false, "Something went wrong with the shipment!!!")] // Fail: shipment fails [TestCase(4, "40410000111145610", true, 9, true, true, "Item Shipped")] // Pass: all checks correct public void CartService\_Validated\_ShipsProduct(double amount, string cardNumber, bool validDate, int quantity,  bool paymentSuccess, bool shipmentSuccess, string expectedResult) {  // 1. Arrange: Prepare the input data and mock behaviors   // Prepare Card Object based on test case parameters  var card = new Card  {  CardNumber = cardNumber,  ValidTo = validDate ? DateTime.Now.AddDays(10) : DateTime.Now.AddDays(-10), // Set expiry date  Name = "Random User",  Amount = amount  };   // Prepare other objects required for the Order  var address = new AddressInfo();  var cartItems = new List<CartItem> { new CartItem { ProductId = "1001", Quantity = quantity, Price = 100 } };  var order = new Order { Address = address, CartItems = cartItems, Card = card };   // Mock ShipmentService behavior:  // Setup the Ship method to return a ShipmentDetails object (success) or null (failure)  // based on the 'shipmentSuccess' parameter of the test case.  var shipmentAck = new ShipmentDetails(); // A dummy object for successful shipment  shipmentServiceMock.Setup(x => x.Ship(It.IsAny<AddressInfo>()))  .Returns(shipmentSuccess ? shipmentAck : null);   // Mock PaymentService behavior:  // CallBase = true ensures that un-mocked methods of PaymentService still call their base implementation.  // This is crucial for partial mocks.  paymentServiceMock.CallBase = true;  // Setup the MakePayment method to return 'paymentSuccess' (true/false) based on the test case parameter.  paymentServiceMock.Setup(x => x.MakePayment(It.IsAny<Card>()))  .Returns(paymentSuccess);   // 2. Act: Execute the method under test  var result = cartService.ValidateCart(order);   // 3. Assert: Verify the outcome   // Verify that the mocked methods were called as expected (optional but good practice)  // For MakePayment, verify it was called if ValidateCart proceeded to it.  // For Ship, verify it was called if MakePayment succeeded.  paymentServiceMock.Verify(x => x.MakePayment(It.IsAny<Card>()), Times.AtLeastOnce());  if (paymentSuccess) // Only verify shipment if payment was successful  {  shipmentServiceMock.Verify(x => x.Ship(It.IsAny<AddressInfo>()), Times.Once());  }   // Assert that the actual result matches the expected result for the given test case  Assert.AreEqual(expectedResult, result); } |
| --- |

**Explanation of the Test Method:**

1. **Preparation (Arrange):**
   * An Order object is meticulously constructed using the parameters supplied by each [TestCase]. This allows a single test method to cover diverse input conditions.
   * **Mocking with Moq's Setup:**
     + shipmentServiceMock.Setup(x => x.Ship(It.IsAny<AddressInfo>())): This line configures the Ship method of our mocked IShipmentService. It.IsAny<AddressInfo>() is a Moq matcher that indicates the method should be set up to behave this way regardless of the AddressInfo object passed to it.
     + .Returns(shipmentSuccess ? shipmentAck : null): This part defines the return value. If shipmentSuccess is true (from the test case), Ship will return a dummy ShipmentDetails object; otherwise, it returns null, simulating a shipment failure.
     + paymentServiceMock.CallBase = true;: For partial mocks, this is essential. It ensures that if PaymentService has other methods not explicitly Setup, their original implementations in the PaymentService class will be invoked.
     + paymentServiceMock.Setup(x => x.MakePayment(It.IsAny<Card>())): This sets up the MakePayment method of our mocked PaymentService.
     + .Returns(paymentSuccess): MakePayment will return true or false based on the paymentSuccess parameter, simulating a payment gateway response.
2. **Execution (Act):**
   * var result = cartService.ValidateCart(order);: The actual method under test, ValidateCart of our real CartService instance, is invoked. Because CartService was injected with mocked dependencies, its internal calls to \_paymentService.ChargeAndShip (which in turn calls MakePayment and \_shipment.Ship) will interact with our controlled mock objects, not real external services.
3. **Verification (Assert):**
   * paymentServiceMock.Verify(...) and shipmentServiceMock.Verify(): These Moq methods are used to verify that specific methods on the mock objects were called during the test execution, and optionally, how many times (Times.AtLeastOnce(), Times.Once()). This ensures that the interactions between CartService and its dependencies occur as expected.
   * Assert.AreEqual(expectedResult, result);: NUnit's assertion method verifies that the result returned by ValidateCart matches the expectedResult defined in the [TestCase], confirming the correctness of the CartService's logic for that specific scenario.