Comprehensive Lab Exercise on MSTesting for a .NET Core Application

Scenario Overview

You are working on a .NET Core application for a fictional online bookstore, "BookHub." The application consists of various functionalities, including user registration, book catalog management, and order processing. The goal of this assessment is to evaluate your ability to write comprehensive unit tests using MSTest for these functionalities. You will be expected to cover positive cases, negative cases, and use mock testing to achieve 100% code coverage.

Assessment Tasks

1. **Setup and Configuration**
   - Create a new .NET Core solution named `BookHub`.
   - Add the following projects to your solution:
     - `BookHub.Core` (Class Library): Contains core business logic.
     - `BookHub.Tests` (MSTest Project): Contains unit tests for the core project.
   - Add necessary NuGet packages:
     - `MSTest.TestFramework`
     - `MSTest.TestAdapter`
     - `Moq` (for mocking dependencies)
2. **User Registration**
   - Implement a `UserService` class in `BookHub.Core` with methods to register a new user and validate user details.
   - Write unit tests for the `UserService` class:
     - **Positive Case**: Test user registration with valid details.
     - **Negative Case**: Test user registration with invalid details (e.g., missing email, password too short).
     - **Mock Testing**: Use Moq to mock the data repository and test the interaction between `UserService` and the repository.

```
// Example UserService class
public class UserService
{
    private readonly IUserRepository _userRepository;

    public UserService(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    public bool RegisterUser(User user)
    {
        if (string.IsNullOrEmpty(user.Email) || user.Password.Length
< 6)
            return false;

        _userRepository.Add(user);
```

```
            return true;
        }

        // Other methods...
    }
    // Example unit test for UserService
    [TestClass]
    public class UserServiceTests
    {
        private Mock<IUserRepository> _mockUserRepository;
        private UserService _userService;

        [TestInitialize]
        public void Setup()
        {
            _mockUserRepository = new Mock<IUserRepository>();
            _userService = new UserService(_mockUserRepository.Object);
        }

        [TestMethod]
        public void RegisterUser_ValidUser_ReturnsTrue()
        {
            // Arrange
            var user = new User { Email = "test@example.com", Password =
    "password123" };

            // Act
            var result = _userService.RegisterUser(user);

            // Assert
            Assert.IsTrue(result);
            _mockUserRepository.Verify(repo =>
    repo.Add(It.IsAny<User>()), Times.Once);
        }

        [TestMethod]
        public void RegisterUser_InvalidEmail_ReturnsFalse()
        {
            // Arrange
            var user = new User { Email = "", Password = "password123" };

            // Act
            var result = _userService.RegisterUser(user);

            // Assert
            Assert.IsFalse(result);
            _mockUserRepository.Verify(repo =>
    repo.Add(It.IsAny<User>()), Times.Never);
        }

        // More tests...
    }
```

3. **Book Catalog Management**
    - Implement a `BookService` class in `BookHub.Core` with methods to add, update, delete, and retrieve books.
    - Write unit tests for the `BookService` class:
        - **Positive Case**: Test adding, updating, and retrieving books with valid details.

- **Negative Case**: Test adding or updating books with invalid details (e.g., missing title, negative price).
- **Mock Testing**: Use Moq to mock the data repository and test the interaction between `BookService` and the repository.

```
// Example BookService class
public class BookService
{
    private readonly IBookRepository _bookRepository;

    public BookService(IBookRepository bookRepository)
    {
        _bookRepository = bookRepository;
    }

    public bool AddBook(Book book)
    {
        if (string.IsNullOrEmpty(book.Title) || book.Price < 0)
            return false;

        _bookRepository.Add(book);
        return true;
    }

    // Other methods...
}
// Example unit test for BookService
[TestClass]
public class BookServiceTests
{
    private Mock<IBookRepository> _mockBookRepository;
    private BookService _bookService;

    [TestInitialize]
    public void Setup()
    {
        _mockBookRepository = new Mock<IBookRepository>();
        _bookService = new BookService(_mockBookRepository.Object);
    }

    [TestMethod]
    public void AddBook_ValidBook_ReturnsTrue()
    {
        // Arrange
        var book = new Book { Title = "Valid Book", Price = 10.0 };

        // Act
        var result = _bookService.AddBook(book);

        // Assert
        Assert.IsTrue(result);
        _mockBookRepository.Verify(repo =>
repo.Add(It.IsAny<Book>()), Times.Once);
    }

    [TestMethod]
    public void AddBook_InvalidTitle_ReturnsFalse()
    {
        // Arrange
        var book = new Book { Title = "", Price = 10.0 };
```

```
        // Act
        var result = _bookService.AddBook(book);

        // Assert
        Assert.IsFalse(result);
        _mockBookRepository.Verify(repo =>
repo.Add(It.IsAny<Book>()), Times.Never);
    }

    // More tests...
}
```

4. **Order Processing**
    - Implement an `OrderService` class in `BookHub.Core` with methods to place an order, validate order details, and calculate total order cost.
    - Write unit tests for the `OrderService` class:
        - **Positive Case**: Test placing an order with valid details.
        - **Negative Case**: Test placing an order with invalid details (e.g., missing book, negative quantity).
        - **Mock Testing**: Use Moq to mock the data repository and test the interaction between `OrderService` and the repository.

```
// Example OrderService class
public class OrderService
{
    private readonly IOrderRepository _orderRepository;

    public OrderService(IOrderRepository orderRepository)
    {
        _orderRepository = orderRepository;
    }

    public bool PlaceOrder(Order order)
    {
        if (order.Books == null || order.Books.Count == 0)
            return false;

        _orderRepository.Add(order);
        return true;
    }

    // Other methods...
}
// Example unit test for OrderService
[TestClass]
public class OrderServiceTests
{
    private Mock<IOrderRepository> _mockOrderRepository;
    private OrderService _orderService;

    [TestInitialize]
    public void Setup()
    {
        _mockOrderRepository = new Mock<IOrderRepository>();
        _orderService = new
OrderService(_mockOrderRepository.Object);
    }
```

```
    [TestMethod]
    public void PlaceOrder_ValidOrder_ReturnsTrue()
    {
        // Arrange
        var order = new Order { Books = new List<Book> { new Book {
Title = "Book1", Price = 10.0 } } };

        // Act
        var result = _orderService.PlaceOrder(order);

        // Assert
        Assert.IsTrue(result);
        _mockOrderRepository.Verify(repo =>
repo.Add(It.IsAny<Order>()), Times.Once);
    }

    [TestMethod]
    public void PlaceOrder_EmptyBookList_ReturnsFalse()
    {
        // Arrange
        var order = new Order { Books = new List<Book>() };

        // Act
        var result = _orderService.PlaceOrder(order);

        // Assert
        Assert.IsFalse(result);
        _mockOrderRepository.Verify(repo =>
repo.Add(It.IsAny<Order>()), Times.Never);
    }

    // More tests...
}
```

5. **Achieving 100% Code Coverage**
    - Ensure that all methods, branches, and edge cases are covered in your unit tests.
    - Use a code coverage tool like Visual Studio's built-in code coverage or a third-party tool to measure and verify that you have achieved 100% code coverage.
    - Provide a code coverage report as part of your submission.

Submission Requirements

- Submit the entire solution with all projects, including the core application and unit tests.
- Include a README file explaining the structure of the solution and how to run the tests.
- Provide a code coverage report demonstrating 100% code coverage.

This assessment will test your ability to write effective unit tests using MSTest, achieve high code quality through comprehensive testing, and use mocking frameworks like Moq to isolate dependencies.