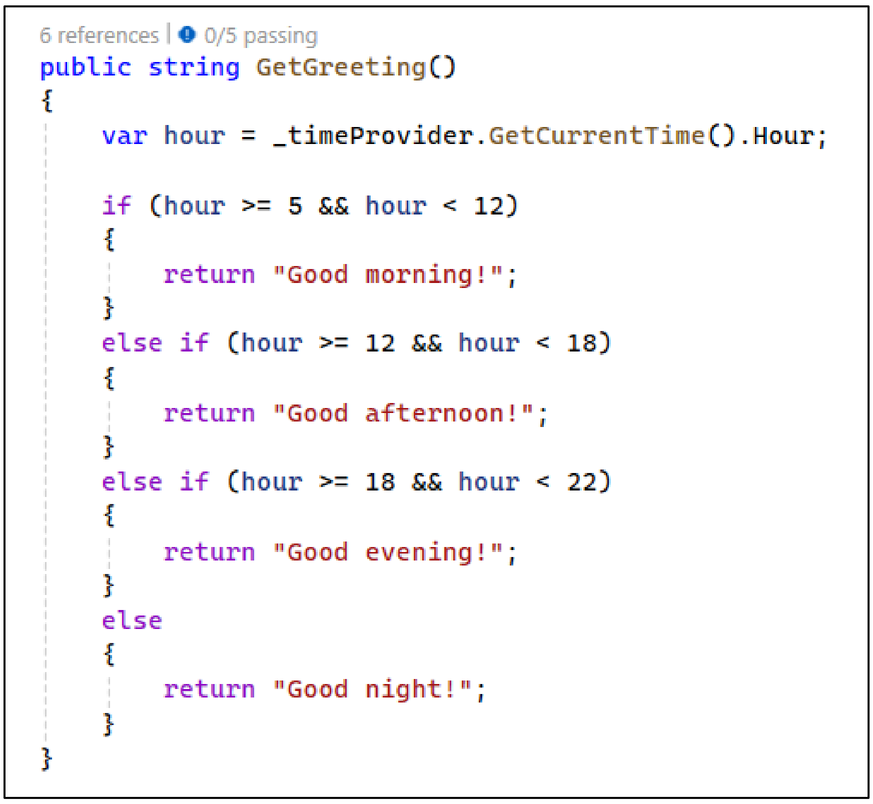
Lab: Unit Testing with Mocking

Lab problems for the ["Back-End Technologies Basics"]() Course @ SoftUni.

1. Get Greeting

You are given the following code for a simple application, GreetingProvider, which provides a greeting based on the time of day.



```csharp
public string GetGreeting()
{
    var hour = _timeProvider.GetCurrentTime().Hour;

    if (hour >= 5 && hour < 12)
    {
        return "Good morning!";
    }
    else if (hour >= 12 && hour < 18)
    {
        return "Good afternoon!";
    }
    else if (hour >= 18 && hour < 22)
    {
        return "Good evening!";
    }
    else
    {
        return "Good night!";
    }
}
```

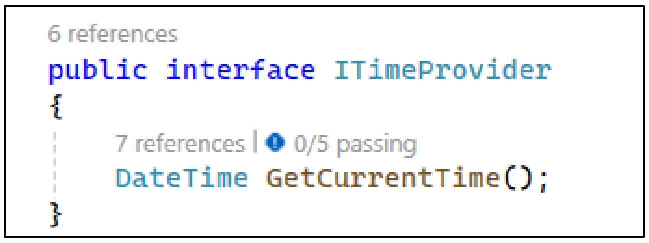
This class functions correctly but poses a significant challenge for unit testing because it directly uses DateTime.Now, making it dependent on the system clock and therefore unpredictable in test scenarios.

Your task is to refactor the GreetingProvider so that it uses an interface to retrieve the current time, instead of directly calling DateTime.Now. and then to write unit tests for the GreetingProvider using the Moq framework.

Steps

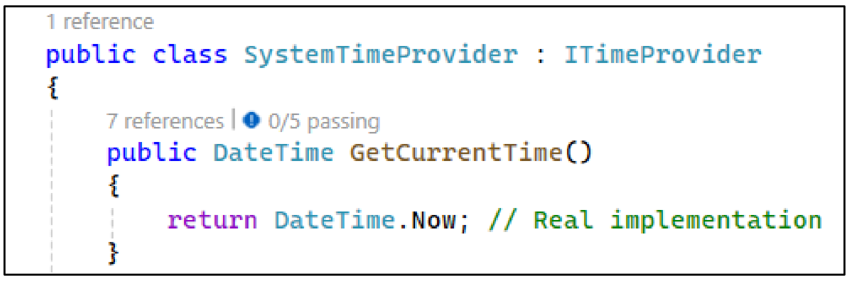
Create an Interface:

- Define an ITimeProvider interface with a method that returns the current time.



```csharp
public interface ITimeProvider
{
    DateTime GetCurrentTime();
}
```

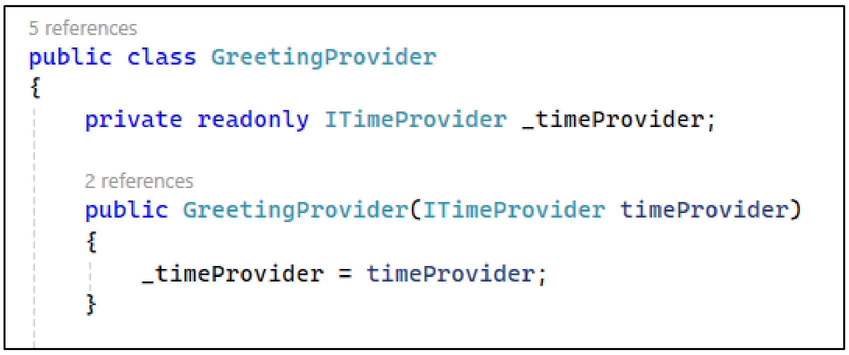
- Implement the interface in a SystemTimeProvider class that returns the actual system time.



```
1 reference
public class SystemTimeProvider : ITimeProvider
{
    7 references | ❶ 0/5 passing
    public DateTime GetCurrentTime()
    {
        return DateTime.Now; // Real implementation
    }
```

Refactor GreetingProvider:

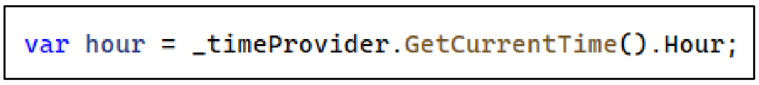
- Modify the GreetingProvider to accept an ITimeProvider through its constructor.



```
5 references
public class GreetingProvider
{
    private readonly ITimeProvider _timeProvider;

    2 references
    public GreetingProvider(ITimeProvider timeProvider)
    {
        _timeProvider = timeProvider;
    }
```

- Replace the direct DateTime.Now call with a call to the ITimeProvider instance.



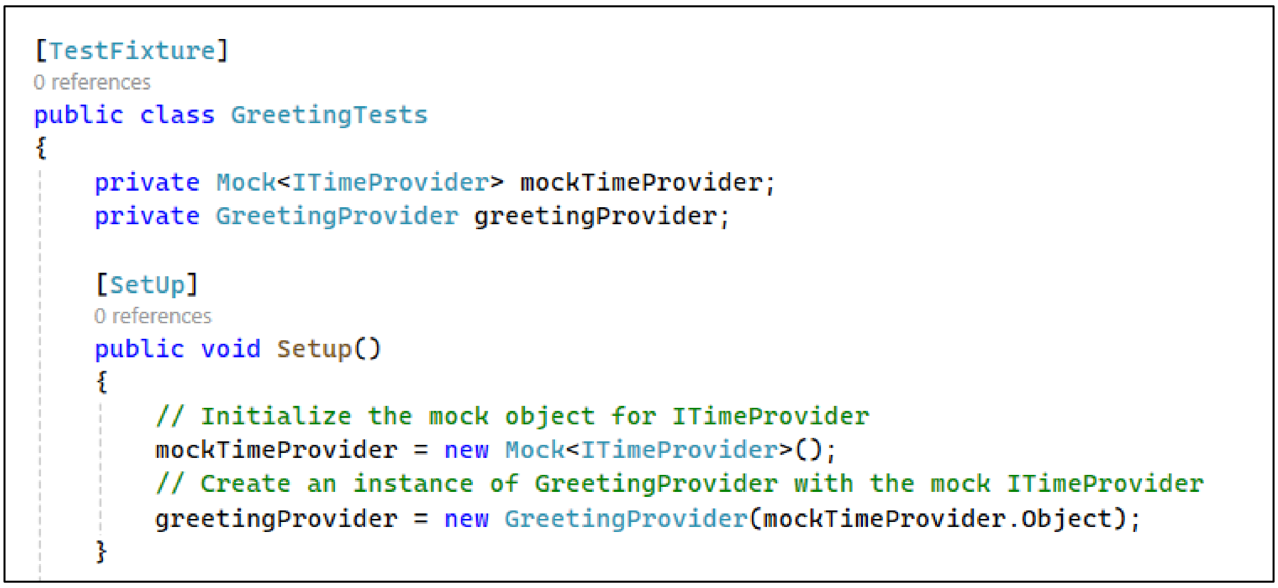
```
var hour = _timeProvider.GetCurrentTime().Hour;
```

Setting Up the Unit Test Environment:

- Create a new unit test project if needed.
- Install the Moq library via NuGet to mock dependencies in unit tests.

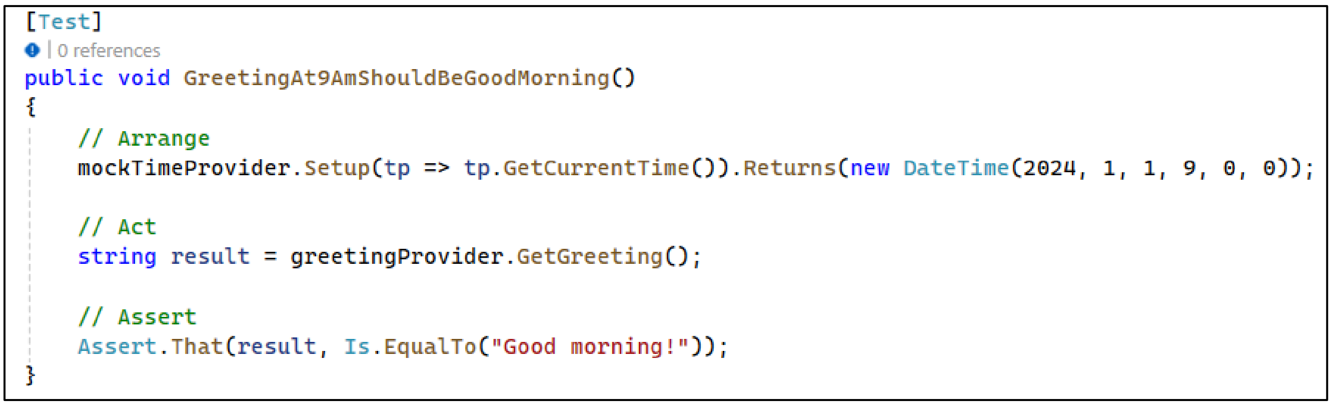
Writing Unit Tests:

Write a test for each type of greeting (morning, afternoon, evening, night), using Moq to mock the ITimeProvider and control the time returned during tests.



```
[TestFixture]
0 references
public class GreetingTests
{
    private Mock<ITimeProvider> mockTimeProvider;
    private GreetingProvider greetingProvider;

    [SetUp]
    0 references
    public void Setup()
    {
        // Initialize the mock object for ITimeProvider
        mockTimeProvider = new Mock<ITimeProvider>();
        // Create an instance of GreetingProvider with the mock ITimeProvider
        greetingProvider = new GreetingProvider(mockTimeProvider.Object);
    }
```

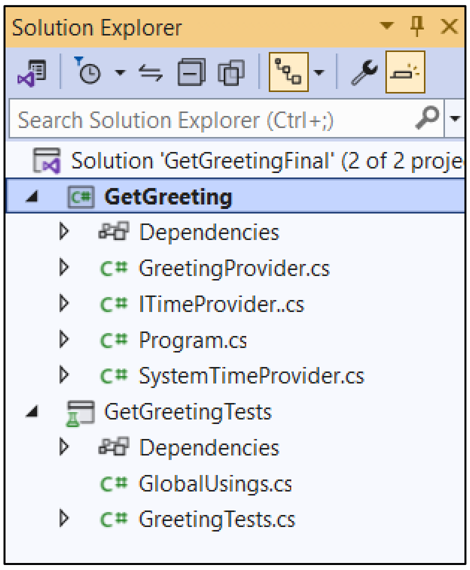


```
[Test]
| 0 references
public void GreetingAt9AmShouldBeGoodMorning()
{
    // Arrange
    mockTimeProvider.Setup(tp => tp.GetCurrentTime()).Returns(new DateTime(2024, 1, 1, 9, 0, 0));

    // Act
    string result = greetingProvider.GetGreeting();

    // Assert
    Assert.That(result, Is.EqualTo("Good morning!"));
}
```

At the end your project should be looking something like this:



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'GetGreetingFinal' (2 of 2 proje
  GetGreeting
    Dependencies
    GreetingProvider.cs
    ITimeProvider..cs
    Program.cs
    SystemTimeProvider.cs
  GetGreetingTests
    Dependencies
    GlobalUsings.cs
    GreetingTests.cs
```

2. Item Management

You are given a console application named ItemManagement. This application manages items through basic CRUD operations (Create, Read, Update, Delete). The application is structured into several components:

ItemManagementApp:

- Contains the ItemService class, which includes business logic.
- Hosts the main Program class for the console application.

ItemManagementLib:

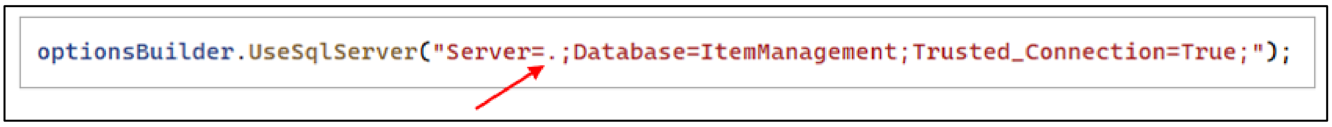
- Consists of the AppDbContext, ItemRepository, and the IItemRepository interface.
- Includes the Item entity that represents the data model.

ItemManagementTests:

- A dedicated project for writing unit tests is already created for you with useful comments.

Your task is to **write unit tests for the ItemService class**. These tests should validate the business logic in isolation, without relying on actual data from the database.

Note: If you're having troubles running the actual app, you will have to replace the server in the connection string with an appropriate server name for your environment. The connection string can be found in AppDbContext.cs and in Program.cs. Change it in both places.



```
optionsBuilder.UseSqlServer("Server=.;Database=ItemManagement;Trusted_Connection=True;");
```

Change this dot, with the name of your server.

Steps

Familiarize Yourself with the Application

Look at the Item Entity:

- Note the properties that define an item, such as Id and Name.

Understand the Data Context:

- Locate the AppDbContext.cs within the ItemManagementLib project.
- Observe how `AppDbContext inherits fromDbContextand contains aDbSet<Item>` for managing item records.

Review the Repository Interface and Implementation:

- Examine the IItemRepository.cs interface to understand the CRUD operations.
- Look at ItemRepository.cs to see how these operations interact with AppDbContext.

Examine the Service Layer:

- Open ItemService.cs in the ItemManagementApp project.
- This class uses IItemRepository to perform the actual operations. Pay attention to how it invokes repository methods to add, get, update, and delete items.

Inspect the Console Application:

- Run the application and use the console interface to perform different operations. Understand how ItemService is utilized here.

Check the Test Project:

- Explore the ItemManagementTests project.
- Notice the predefined tests names and comments that guide you on where to add test cases.

Prepare for Writing Tests:

- Before writing tests, ensure you understand what each method in ItemService is supposed to do.
- Think about the different scenarios you need to test for each method (e.g., what happens when each method receives valid inputs, invalid inputs, or edge cases).

Begin Writing Your Tests:

- Open the ItemManagementTests.cs file within the ItemManagementTests project.

Utilize Mocking:

- Use the Mock<IItemRepository> instance to simulate database operations.
- Setup the mock repository to return specific results or perform certain actions

- when the service layer calls its methods. This allows you to test the service layer's response to different data conditions.

Implement Test Logic:

- For each test, use the Arrange-Act-Assert pattern.
- Arrange: Set up the mock repository's expected behavior.
- Act: Call the method under test on the ItemService instance.
- Assert: Verify that the outcome is as expected, whether it's checking return values or confirming that certain methods were called on the mock repository.

Run and Refine Your Tests:

- Use your test runner to execute your tests and observe the results.
- If a test fails, examine why it failed and refine the test or the service logic if necessary.
- Write comments in your tests to explain the reasoning behind each test case, especially for complex scenarios.