✓ 100 XP

Exercise - Develop unit tests using GitHub Copilot tools

20 minutes

GitHub Copilot can help you write unit tests for your code. There are several ways to use GitHub Copilot to generate unit tests:

- **Generate test cases**: You can use GitHub Copilot to generate test cases for your code. Copilot can suggest test cases based on the code you have written. You can then use these test cases to create unit tests for your code.
- **Generate test methods**: Copilot can also generate test methods for your code. You can use these test methods to create unit tests for your code.
- **Generate test assertions**: Copilot can suggest assertions that you can use in your unit tests. These assertions can help you verify the behavior of your code.
- **Generate test mocks**: Copilot can generate mock objects that you can use in your unit tests. These mock objects can help you isolate the code you're testing from its dependencies.
- **Generate test data**: Copilot can generate test data that you can use in your unit tests. This test data can help you test your code with different inputs.
- **Generate test setup code**: Copilot can generate setup code for your unit tests. This code can help you set up the test environment before running your tests.
- **Generate test teardown code**: Copilot can generate teardown code for your unit tests. This code can help you clean up the test environment after running your tests.

Your existing solution includes a unit test project named UnitTests. This project already contains unit tests that partially cover the service classes in the ApplicationCore project.

In this exercise, you use GitHub Copilot to complete the following tasks:

1. Examine the approach to unit testing implemented by the UnitTests project.
2. Extend the UnitTests project to begin testing the data access classes in the `Library.Infrastructure` project.

Examine the approach to unit testing implemented by the UnitTests project

Your existing codebase includes a UnitTests project that implements the following folder structure:

- UnitTests\
  - ApplicationCore\
    - LoanService\
      - `ExtendLoan.cs`
      - `ReturnLoan.cs`
    - PatronService\
      - `RenewMembership.cs`
  - LoanFactory.cs
  - PatronFactory.cs

This structure mirrors and supports the `Services` portion of the `ApplicationCore` project.

- ApplicationCore\
  - Services\
    - LoanService.cs: Contains the `ExtendLoan` and `ReturnLoan` methods.
    - PatronService.cs: Contains the `RenewMembership` method.

In this section of the exercise, you use GitHub Copilot to examine the unit testing approach implemented by the UnitTests project.

Use the following steps to complete this section of the exercise:

1. Open the Chat view.

2. Add the following files to the Chat context: `LoanFactory.cs`, `PatronFactory.cs`, `ExtendLoan.cs`, `ReturnLoan.cs`, `RenewMembership.cs`, `LoanService.cs`, `PatronService.cs`.

   The files are located in the `tests/UnitTests`, `tests/UnitTests/ApplicationCore/LoanService`, `tests/UnitTests/ApplicationCore/PatronService`, and `src/Library.ApplicationCore/Services` folders.

3. Enter the following prompt:

   plaintext

   ```
   @workspace Explain the approach to unit testing that's been implemented in this workspace.
   ```

This prompt asks GitHub Copilot to explain the approach to unit testing in the context of the workspace. Several files are included in the prompt to provide context for the response.

4. Take a minute to review the response from GitHub Copilot.

   You should see an explanation that's similar to the following description:

   - Mocking dependencies: The tests use `NSubstitute` to create mock implementations of interfaces.
   - Test organization: Tests are organized into separate files based on the methods being tested. For example, `ExtendLoan` and `ReturnLoan` methods are tested in separate files under the `LoanService` folder.
   - Assertions: Assertions are used to verify the expected outcome of each test.
   - Test scenarios: The tests cover various scenarios for each method, such as returning different statuses based on the input.
   - Isolation: Each test is isolated from others to prevent side effects.
   - Factory methods: Factory methods are used to create test data consistently.

5. Enter the following prompt:

   plaintext

   ```plaintext
   @workspace What are the benefits of this approach to unit testing?
   ```

6. Take a minute to review the response from GitHub Copilot.

   You should see an explanation that identifies a list of benefits that's similar to the following list:

   - Isolation of Dependencies: Using mocks isolates the unit of work from its dependencies.
   - Repeatability and Consistency: Factory methods provide a consistent way to create test data.
   - Readability and Maintainability: Descriptive test methods and clear arrangement of test steps improve the readability of tests.
   - Comprehensive Coverage: Testing various scenarios ensures that the code is robust and handles different edge cases.
   - Separation of Concerns: The separation of test data creation from the test logic itself keeps the tests clean and focused on the behavior being tested.
   - Immediate Feedback: Running unit tests provides immediate feedback on the correctness of the code.

7. Add the following files to the Chat context: `ExtendLoan.cs`, `ReturnLoan.cs`, `RenewMembership.cs`, `LoanFactory.cs`, `PatronFactory.cs`, `LoanService.cs`, `PatronService.cs`.

8. Enter the following prompt:

   plaintext

   ```
   @workspace How can I extend the `UnitTests` project to begin testing methods in the `Library.Infrastructure` project? Provide a process overview.
   ```

   This prompt asks GitHub Copilot to provide a process overview for extending the `UnitTests` project to begin testing methods in the `Library.Infrastructure` project. Several files are included in the prompt to provide context for the response.

9. Take a minute to review the response from GitHub Copilot.

   You should see an explanation that's similar to the following description:

   markdown

   ```
   1. Add Project Reference:

     - Open the `UnitTests.csproj` file located at `tests/UnitTests/UnitTests.csproj`.
     - Add a reference to the `Library.Infrastructure` project.

   2. Identify methods to test:

     - Identify the methods in the `Library.Infrastructure` project that need to be tested. For example, if you want to test the `LoanService` class, identify the methods in the `LoanService` class that need to be tested.

   3. Create Test Classes:

     - Create new test classes in the `tests/UnitTests` directory to test the methods in the `Library.Infrastructure` project. For example, if you want to test `LoanService`, create a file named `LoanServiceTests.cs`.

   4. Write Unit Tests:

     - In the newly created test classes, write unit tests for the methods in the `Library.Infrastructure` project. Use the xUnit framework for writing tests and NSubstitute for mocking dependencies.

   5. Run Tests:
   ```

```
  - Use Visual Studio Code's integrated terminal or the Test Explorer
to run the tests.
  - Ensure that the tests are discovered and executed correctly.
```

Extend the UnitTests project to begin testing data access classes

In this section of the exercise, you use GitHub Copilot to help you create unit tests for the `Library.Infrastructure` project.

The `Library.Infrastructure` project contains data access classes that interact with the file system to load and save data. The project includes the following classes:

- JsonData: A class that loads and saves JSON data.
- JsonLoanRepository: A class that implements the ILoanRepository interface and uses the JsonData class to load and save loan data.
- JsonPatronRepository: A class that implements the IPatronRepository interface and uses the JsonData class to load and save patron data.

You'll start by writing unit tests for the `JsonLoanRepository` class.

Use the following steps to complete this section of the exercise:

1. Add the following files to the Chat context: `UnitTests.csproj`.

2. Enter the following prompt in the Chat view:

   plaintext

   ```
   @workspace Explain how to add a reference to the
   Library.Infrastructure project inside `UnitTests.csproj`.
   ```

   This prompt asks GitHub Copilot to explain how to add a reference to the `Library.Infrastructure` project inside the `UnitTests.csproj` file.

3. Use the GitHub Copilot response to update the UnitTests.csproj file.

   The updated UnitTests.csproj file should look similar to the following XML code:

   XML

   ```
   <Project Sdk="Microsoft.NET.Sdk">
     <PropertyGroup>
       <TargetFramework>net8.0</TargetFramework>
       <ImplicitUsings>enable</ImplicitUsings>
   ```

```
    <Nullable>enable</Nullable>
    <IsPackable>false</IsPackable>
    <IsTestProject>true</IsTestProject>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="coverlet.collector" Version="6.0.0" />
    <PackageReference Include="Microsoft.NET.Test.Sdk"
Version="17.8.0" />
    <PackageReference Include="NSubstitute" Version="5.1.0" />
    <PackageReference Include="xunit" Version="2.5.3" />
    <PackageReference Include="xunit.runner.visualstudio"
Version="2.5.3" />
  </ItemGroup>
  <ItemGroup>
    <Using Include="Xunit" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\..\src\Library.ApplicationCore\Li-
brary.ApplicationCore.csproj" />
    <ProjectReference Include="..\..\src\Library.Infrastructure\Li-
brary.Infrastructure.csproj" />
  </ItemGroup>
</Project>
```

4. Open the `JsonLoanRepository.cs` file.

   `JsonLoanRepository.cs` is located in the `src/Library.Infrastructure/Data/` folder.

5. Take a minute to review the `JsonLoanRepository.cs` file.

C#

```
using Library.ApplicationCore;
using Library.ApplicationCore.Entities;

namespace Library.Infrastructure.Data;

public class JsonLoanRepository : ILoanRepository
{
    private readonly JsonData _jsonData;

    public JsonLoanRepository(JsonData jsonData)
    {
        _jsonData = jsonData;
    }

    public async Task<Loan?> GetLoan(int id)
    {
        await _jsonData.EnsureDataLoaded();

        foreach (Loan loan in _jsonData.Loans!)
        {
            if (loan.Id == id)
            {
                Loan populated = _jsonData.GetPopulatedLoan(loan);
                return populated;
            }
        }
        return null;
    }

    public async Task UpdateLoan(Loan loan)
    {
        Loan? existingLoan = null;
        foreach (Loan l in _jsonData.Loans!)
        {
            if (l.Id == loan.Id)
            {
                existingLoan = l;
                break;
            }
        }

        if (existingLoan != null)
        {
            existingLoan.BookItemId = loan.BookItemId;
            existingLoan.PatronId = loan.PatronId;
            existingLoan.LoanDate = loan.LoanDate;
            existingLoan.DueDate = loan.DueDate;
            existingLoan.ReturnDate = loan.ReturnDate;

            await _jsonData.SaveLoans(_jsonData.Loans!);

            await _jsonData.LoadData();
        }
    }
}
```

6. Notice the following details about the `JsonLoanRepository` class:

    - The `JsonLoanRepository` class contains two methods: `GetLoan` and `UpdateLoan`.
    - The `JsonLoanRepository` class uses a `JsonData` object to load and save loan data.

    You'll start by writing unit tests for the `GetLoan` method.

7. Create the following folder structure under the `UnitTests` project.

    - Infrastructure\
        - JsonLoanRepository\

    This folder structure mirrors the approach used for the `Library.ApplicationCore` unit tests.

8. Create a class file named `GetLoan` in the `JsonLoanRepository` folder.

9. Take a minute to consider the field and class constructor requirements for the `GetLoan` unit tests.

   The `JsonLoanRepository.GetLoan` method receives a loan ID parameter when it's called. The method uses `_jsonData.EnsureDataLoaded` to get the latest JSON data, and `_jsonData.Loans` to search for a matching loan. If the method finds a matching loan ID, it returns a populated loan object (`populated`). If the method is unable to find a matching loan ID, it returns `null`.

   For the GetLoan unit tests:

   - You can use a mock loan repository object (`_mockLoanRepository`) to help test the case where a matching ID is found. Load the mock with the ID you want to find. The `ReturnLoanTest` class demonstrates how to mock the `ILoanRepository` interface and instantiate a mock loan repository object.

   - You can use a non-mock loan repository object (`_jsonLoanRepository`) to test the case where no matching ID is found. Just specify a loan ID that you know isn't in the file (anything over 100 should work).

   - You'll need a `JsonData` object to create a non-mock `JsonLoanRepository` object. Since the `UnitTests` project doesn't have access to the `JsonData` object created by the `ConsoleApp` project, you'll need to create one using the `IConfiguration` interface.

10. Add the following files to the Chat context: `JsonLoanRepository.cs`, `ReturnLoan.cs`, `LoanService.cs`, `LoanFactory.cs`, `JsonData.cs`.

11. Enter the following prompt:

   plaintext

   ```
   @workspace Create fields and a class constructor for the `GetLoan.cs`
   file. The class will be used to create unit tests for the GetLoan
   method in the `JsonLoanRepository.cs` file. Create the following pri-
   vate readonly fields: `_mockLoanRepository`, `_jsonLoanRepository`,
   `_configuration`, and `_jsonData`. Instantiate the fields in the `Get-
   LoanTest` constructor. Use `ConfigurationBuilder` to create a `_con-
   figuration` object that can be used to instantiate the JsonData
   object.
   ```

   This prompt asks GitHub Copilot to suggest fields and a class constructor.

12. Take a minute to review the response from GitHub Copilot.

    You should see a code suggestion that's similar to the following code snippet:

    C#

    ```csharp
    using NSubstitute;
    using Library.ApplicationCore;
    using Library.ApplicationCore.Entities;
    using Library.ApplicationCore.Interfaces;
    using Library.Infrastructure.Data;
    using Microsoft.Extensions.Configuration;

    namespace UnitTests.Infrastructure.JsonLoanRepository;

    public class GetLoanTest
    {
        private readonly ILoanRepository _mockLoanRepository;
        private readonly JsonLoanRepository _jsonLoanRepository;
        private readonly IConfiguration _configuration;
        private readonly JsonData _jsonData;

        public GetLoanTest()
        {
            _mockLoanRepository = Substitute.For<ILoanRepository>();
            _configuration = new ConfigurationBuilder()
                .AddJsonFile("appsettings.json")
                .Build();
            _jsonData = new JsonData(_configuration);
            _jsonLoanRepository = new JsonLoanRepository(_jsonData);
        }

        // Add test methods here
    }
    ```

13. Use the code suggestion provided by GitHub Copilot to update `GetLoan.cs`.

14. Notice the following issues:

    - There is a conflict between the namespace and the use of `JsonLoanRepository` in the code. You should update the namespace in GetLoans.cs to follow the pattern used in the `ReturnLoan.cs` and `RenewMembership.cs` files.

    - If `ILoanRepository` isn't recognized in your code, you may need to add a `using` directive for `Library.ApplicationCore` to the top of the file.

    - If the `_configuration` object isn't instantiated correctly, you may need to update the code line containing `ConfigurationBuilder`. You can simplify the code to use `_configuration = new ConfigurationBuilder().Build();`.

   - If a `using Library.ApplicationCore.Interfaces` is suggested by GitHub Copilot, you can delete it from the top of the file.

15. Update the `GetLoan.cs` file to match the following code snippet:

    C#

    ```csharp
    using NSubstitute;
    using Library.ApplicationCore;
    using Library.ApplicationCore.Entities;
    using Library.Infrastructure.Data;
    using Microsoft.Extensions.Configuration;

    namespace UnitTests.Infrastructure.JsonLoanRepositoryTests;

    public class GetLoanTest
    {
        private readonly ILoanRepository _mockLoanRepository;
        private readonly JsonLoanRepository _jsonLoanRepository;
        private readonly IConfiguration _configuration;
        private readonly JsonData _jsonData;

        public GetLoanTest()
        {
            _mockLoanRepository = Substitute.For<ILoanRepository>();
            _configuration = new ConfigurationBuilder().Build();
            _jsonData = new JsonData(_configuration);
            _jsonLoanRepository = new JsonLoanRepository(_jsonData);
        }

    }
    ```

16. Add the following files to the Chat context: `JsonLoanRepository.cs`, `ReturnLoan.cs`, `LoanService.cs`, `LoanFactory.cs`, `JsonData.cs`, `Loans.json`.

17. Select the contents of the `GetLoan.cs` file, and then enter the following prompt in the Chat view:

    plaintext

    ```plaintext
    @workspace Update the selection to include a unit test for the `Json-
    LoanRepository.GetLoan` method. The unit test should test the case
    where a loan ID is found in the data. Use `_mockLoanRepository` to
    arrange the expected return loan. Use `_jsonLoanRepository` to return
    an actual loan. Asserts should verify that the return loan ID matches
    the expected loan ID. Use a loan ID that exists in the `Loans.json`
    file.
    ```

This prompt asks GitHub Copilot to suggest a unit test for the `JsonLoanRepository.GetLoan` method. The unit test should test the case where a loan ID is found in the data. The test should use `_mockLoanRepository` to arrange the expected return loan, `_jsonLoanRepository` to return an actual loan, and asserts to verify that the return loan ID matches the expected loan ID. The loan ID should exist in the `Loans.json` file.

18. Take a minute to review the response from GitHub Copilot.

    You should see a code suggestion that's similar to the following code snippet:

    C#

```csharp
using NSubstitute;
using Library.ApplicationCore;
using Library.ApplicationCore.Entities;
using Library.Infrastructure.Data;
using Microsoft.Extensions.Configuration;
using Xunit;

namespace UnitTests.Infrastructure.JsonLoanRepositoryTests;

public class GetLoanTest
{
    private readonly ILoanRepository _mockLoanRepository;
    private readonly JsonLoanRepository _jsonLoanRepository;
    private readonly IConfiguration _configuration;
    private readonly JsonData _jsonData;

    public GetLoanTest()
    {
        _mockLoanRepository = Substitute.For<ILoanRepository>();
        _configuration = new ConfigurationBuilder().Build();
        _jsonData = new JsonData(_configuration);
        _jsonLoanRepository = new JsonLoanRepository(_jsonData);
    }

    [Fact(DisplayName = "JsonLoanRepository.GetLoan: Returns loan when loan ID is found")]
    public async Task GetLoan_ReturnsLoanWhenLoanIdIsFound()
    {
        // Arrange
        var loanId = 1; // Use a loan ID that exists in the Loans.json file
        var expectedLoan = new Loan { Id = loanId, BookItemId = 101, PatronId = 202, LoanDate = DateTime.Now, DueDate = DateTime.Now.AddDays(14) };
        _mockLoanRepository.GetLoan(loanId).Returns(expectedLoan);
        await _jsonData.EnsureDataLoaded(); // Ensure data is loaded

        // Act
        var actualLoan = await _jsonLoanRepository.GetLoan(loanId);

        // Assert
        Assert.NotNull(actualLoan);
        Assert.Equal(expectedLoan.Id, actualLoan?.Id);
    }
}
```

19. Use the code suggestion provided by GitHub Copilot to update `GetLoan.cs`.

    The `JsonLoanRepository.GetLoan` method calls `_jsonData.EnsureDataLoaded` to ensure that the JSON data is loaded. You can delete the code line that calls `_jsonData.EnsureDataLoaded` in the test method if it's included.

    If the `Loan` class isn't recognized in your code, ensure that you have a `using Library.ApplicationCore.Entities` statement at the top of the GetLoan.cs file. The `Loan` class is located in the `Library.ApplicationCore.Entities` namespace.

20. Build the `AccelerateDevGitHubCopilot` solution to ensure there are no errors.

21. Use GitHub Copilot's autocompletion feature to create a test for the case where the loan ID isn't found.

    Create a blank line after the `GetLoan_ReturnsLoanWhenLoanIdIsFound` method.

    Accept the autocompletion suggestions to create a new test method.

22. Take a minute to review the response from GitHub Copilot.

    GitHub Copilot's autocompletion feature will probably mock an expected loan even though it isn't needed. You can delete the code that mocks an expected loan, but you need a loan ID that doesn't exist in the `Loans.json` file.

    You should see a code suggestion that's similar to one of the following code snippets:

C#

```
[Fact(DisplayName = "JsonLoanRepository.GetLoan: Returns null when
loan ID is not found")]
public async Task GetLoan_ReturnsNullWhenLoanIdIsNotFound()
{
    // Arrange
    var loanId = 999; // Use a loan ID that does not exist in the
Loans.json file
    var expectedLoan = new Loan { Id = loanId, BookItemId = 101,
PatronId = 202, LoanDate = DateTime.Now, DueDate =
DateTime.Now.AddDays(14) };
    _mockLoanRepository.GetLoan(loanId).Returns(expectedLoan);
    // Act
    var actualLoan = await _jsonLoanRepository.GetLoan(loanId);

    // Assert
    Assert.Null(actualLoan);
}
```

23. Accept the autocompletion suggestion that assigns a `loanId` value that isn't in the data set.

    If none of the suggestions assign `loanId` numbers that aren't in the data set, you can use the **Ctrl+Enter** keyboard shortcut to list additional suggestions.

24. Notice that the unit tests require access to the JSON data files.

    The `JsonLoanRepository.GetLoan` method uses a `JsonData` object to load and save loan data.

    The JSON data files are located in the `Library.Console\Json` folder. You need to update the `UnitTests.csproj` file to include these files in the test project.

25. Add the following XML snippet to the `UnitTests.csproj` file:

    XML

    ```xml
    <ItemGroup>
        <None Include="..\..\src\Library.Console\Json\**\*">
            <Link>Json\%(RecursiveDir)%(FileName)%(Extension)</Link>
            <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
        </None>
    </ItemGroup>
    ```

    This ensures that the JSON data files are copied to the output directory when the tests are run.

Check your work

There are several ways to run the unit tests for the `JsonLoanRepository` class. You can use Visual Studio Code's Test Explorer, the integrated terminal, or the `dotnet test` command.

Use the following steps to complete this section of the exercise:

1. Ensure that you have the GetLoans.cs file open in editor.

2. Build the solution and ensure that there are no errors.

   Right-click **AccelerateDevGitHubCopilot** and then select **Build**.

3. Notice the "green play button" to the left of the test methods.

4. Open Visual Studio Code's Test Explorer view.

   To open the Test Explorer view, select the beaker-shaped icon on the left-side Activity bar. The Test Explorer is labeled "Testing" in the user interface.

   The Test Explorer is a tree view that shows all the test cases in your workspace. You can run/debug your test cases and view the test results using Test Explorer.

5. Expand **UnitTests** and the underlying nodes to locate `GetLoanTest`.

6. Run the **JsonLoanRepository.GetLoan: Returns loan when loan ID is found** test case.

7. Notice the test results in the Test Explorer view and the Editor.

   You should see a green checkmark that indicates the test passed.

8. Use the Editor to run the **JsonLoanRepository.GetLoan: Returns null when loan ID is not found** test case.

9. Notice the test results in the Test Explorer view and the Editor.

   To run the test from the Editor, select the green play button to the left of the test method.

10. Ensure that the **JsonLoanRepository.GetLoan: Returns null when loan ID is not found** test passes.

    You should see a green checkmark to the left of both tests.

Next unit: Exercise - Refactor and improve code sections using GitHub Copilot tools

< Previous Next >