Unit Testing

Writing Unit Test for a Void Method in C#: Easy, Medium, and Hard Level

Kristijan Kralj

How do you know that you play the game of life on the easy level?

Simply, when you have to write a unit test for a method that returns a value. Think about it. You create a class and call the method with the necessary parameters. Then, you get the result and assert against it. Easy.

Do you know what is not that straightforward? Writing tests for a void method. The method that doesn't return a value. So how to test it?

The appropriate way to test a void method in C# is to check the side-effect after you execute the void method.

In this post, you will see three examples of how to test various void methods.

Table of Contents

How to unit test a method with void return type?

In recent years, software testing has become an essential part of the software development process, and any team that wants to be a top organization must test their code and test it well.

Today we will focus on testing a void method. A void method is a method that doesn't return anything. Usually, it performs some checks and then passes the data to another class that does one of the following operations:

- **save changes to the database**
- **perform network call**
- **any other type of external calls (send an email, call another service)**

If any of this represents your void method, then you will see how to test that method.

Easy level – check properties

When testing a class, one of the most important things is to ensure that all of the methods that the class implements work as expected. The easiest way to test a void method is when a void method alters the property of the class under test. That way, in the assert phase, you only need to check the value of the property.

The following example shows the `InitializeAsync` method from the `AddTransactionViewModel`.

```
1  public async Task InitializeAsync(string id)
2  {
3      if (string.IsNullOrEmpty(id) || !int.TryParse(id, out int transactionId))
4      {
5          IsDeposit = true;
6          TransactionDate = DateTime.Today;
7          return;
8      }
9      var selectedTransaction = await _transactionRepository.GetById(transactionId)
10     IsDeposit = selectedTransaction.Status == Constants.TRANSACTION_DEPOSITED;
11     TransactionDate = selectedTransaction.TransactionDate;
12 }
```

As you can see, the method takes one input parameter, id. It then checks whether the id can be parsed to an `int`. If it can't, then in the guard clause (https://methodpoet.com/refactor-nested-if-statements/), `IsDeposit` and `TransactionDate` properties are set, and the execution stops.

After a successful parse operation, the method retrieves the transaction based on the id and sets the `IsDeposit` and `TransactionDate` properties.

In this case, the side-effects of this method are the new values stored in the `IsDeposit` and `TransactionDate` properties. The simple test that covers the case when the id can't be parsed is as

follows:

```
[Fact]
public async void InitializeAsync_sets_transaction_date_to_current_when_id_cant_b
{
    //CreateAddTransactionViewModel is a factory method that creates the viewmode
    AddTransactionViewModel viewModel = CreateAddTransactionViewModel();

    await viewModel.InitializeAsync("potato");

    Assert.True(viewModel.IsDeposit);
    Assert.Equal(DateTime.Today, viewModel.TransactionDate);
}
```

The test covers the void method by asserting that the properties have the correct state.

Medium level – use mock to verify the output

The most popular option to verify the output when testing the void method is to use the mock object (https://methodpoet.com/stub-vs-mock/). A mock object is an object that replicates the behavior of a real object for the purposes of unit testing. For example, you can observe how the class under test calls the dependency with a mock object.

You can use many frameworks in C# to create mocks and stubs and make the testing process easier. The mocking framework (https://methodpoet.com/unit-testing-with-moq/) I prefer to use is Moq (https://github.com/moq/moq4).

The next code snippet shows the `LoginUser` async method of the `LoginViewModel` .

```
public async Task LoginUser()
{
    //get user by email
    var user = (await _userRepository.GetAllAsync())
        .FirstOrDefault(x => x.Email == Email.Value);
    if (user == null)
    {
        await _dialogMessage.DisplayAlert("Error",
                                          "Credentials are wrong.",
                                          "Ok");
        return;
    }
    //check that the hashed passwords match
    if (!SecurePasswordHasher.Verify(Password.Value, user.HashedPassword))
    {
        await _dialogMessage.DisplayAlert("Error",
                                         "Credentials are wrong.",
                                         "Ok");
        return;
    }
    //navigate to the main flow
    _navigationService.GoToMainFlow();
}
```

The method tries to get the user that matches the entered email. If the user exists, then the password is checked. If the passwords match, then the navigation service navigates to the main flow.

The following test method checks that the dialog message displays the message when the user doesn't exist.

```
[Fact]
public async void LoginCommand_shows_error_when_email_is_not_correct()
{
    var mockDialogMessage = new Mock<IDialogMessage>();
    var stubRepository = new Mock<IRepository<User>>();
    stubRepository
        .Setup(x => x.GetAllAsync())
        .ReturnsAsync(new List<User> { });

    LoginViewModel viewModel = new LoginViewModel(null,
                                                  stubRepository.Object,
                                                  mockDialogMessage.Object,
                                                  _mockUserPreferences.Object);
    viewModel.Email.Value = "email@crypto.com";
    viewModel.Password.Value = "pass";

    await viewModel.LoginUser();

    mockDialogMessage
        .Verify(x => x.DisplayAlert(It.IsAny<string>(),
                                    "Credentials are wrong.",
                                    It.IsAny<string>()), Times.Once);
}
```

In this case, the test checks that the `DisplayAlert` of the `IDialogMessage` mock object was called exactly one time.

The mocking framework is especially useful when you use test driven development (https://methodpoet.com/tdd/), a process where you write test code before the production code. With a test framework such as Moq, you can simulate how the dependency of the class under the test will behave, even if the actual implementation of the dependency doesn't exist yet.

One downside of using mocks is that you check the interaction between classes. These tests are called "brittle" because they depend on the underlying code and should be used sparingly. While writing unit tests, it is important to avoid brittle code.

Brittle code is fragile to changes in functionality over time. For example, if a test is using a mock object to test the behavior of a given class, then if the class changes, the test might break and you need to rewrite it to pass again. This can result in more programming time.

How to not test implementation details and interaction between classes when the method returns void? By using integration testing.

Hard level – going one step further – integration testing

One of the first things to think about when making a new class is how you want to test it. There are two common techniques: unit testing and integration testing. Unit testing is best used when you want to test a small snippet of code. You usually write isolated tests in unit testing, meaning that you test the logic in one method.

On the other hand, you can use integration tests to test one of the big building blocks of your application. Integration tests usually test whole classes or even whole modules. Because they cover big blocks of code, integration tests get expensive when you have many dependencies in your system.

A good example might be testing if a user can create an account on your site. For example, the steps might include:

- Check the name
- Check the email address
- Check the password
- Create a new user and store it in a database

How to test this scenario? You can use the unit test suite to test the first three steps. A unit test case can check how your validation logic works when you enter various input data. Methods that validate input parameters will usually return the bool as a result type, making it easy to write tests for them.

The last point, creating a new user and storing it into a database, typically looks like a void method. As such, the most accurate way to test it would be to write an integration test. Inside it, you check that the new user is stored in the database when all entered data is correct.

There are pros and cons to each testing technique. Unit testing is faster but is very limited in scope. Integration tests are a lot more comprehensive, but they're also more time-consuming and difficult to execute. But needless to say, you need both to ensure the best possible code coverage (https://methodpoet.com/testing-pyramid/).

Conclusion

In conclusion, there are multiple ways to test the void method, which is dependent on the method's side-effects and what kind of test you wish to run. A unit test checks the method's functionality and can cover the void method if the side effect is stored in publicly available property. Integration test checks how the method interacts with the external environment.

In an ideal world, you should write a test for every type of method. However, time is (as always) against us, which is why you need to prioritize your testing efforts.