Chapter 36: Unit Testing

Section 36.1: Testing the view models

Before we start...

In terms of application layers your ViewModel is a class containing all the business logic and rules making the app do what it should according to the requirements. It's also important to make it as much independent as possible reducing references to UI, data layer, native features and API calls etc. All of these makes your VM be testable.
In short, your ViewModel:

- Should not depend on UI classes (views, pages, styles, events);
- Should not use static data of another classes (as much as you can);
- Should implement the business logic and prepare data to be should on UI;
- Should use other components (database, HTTP, UI-specific) via interfaces being resolved using Dependency Injection.

> Your ViewModel may have properties of another VMs types as well. For example `ContactsPageViewModel` will have propery of collection type like `ObservableCollection<ContactListItemViewModel>`

Business requirements

Let's say we have the following functionality to implement:

```
As an unauthorized user
I want to log into the app
So that I will access the authorized features
```

After clarifying the user story we defined the following scenarios:

```
Scenario: trying to log in with valid non-empty creds
Given the user is on Login screen
When the user enters 'user' as username
And the user enters 'pass' as password
And the user taps the Login button
Then the app shows the loading indicator
And the app makes an API call for authentication

Scenario: trying to log in empty username
Given the user is on Login screen
When the user enters '  ' as username
And the user enters 'pass' as password
And the user taps the Login button
Then the app shows an error message saying 'Please, enter correct username and password'
And the app doesn't make an API call for authentication
```

We will stay with only these two scenarios. Of course, there should be much more cases and you should define all of them before actual coding, but it's pretty enough for us now to get familiar with unit testing of view models.

Let's follow the classical TDD approach and start with writing an empty class being tested. Then we will write tests and will make them green by implementing the business functionality.

Common classes

```
public abstract class BaseViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Services

Do you remember our view model must not utilize UI and HTTP classes directly? You should define them as abstractions instead and not to depend on implementation details.

```
/// <summary>
/// Provides authentication functionality.
/// </summary>
public interface IAuthenticationService
{
    /// <summary>
    /// Tries to authenticate the user with the given credentials.
    /// </summary>
    /// <param name="userName">UserName</param>
    /// <param name="password">User's password</param>
    /// <returns>true if the user has been successfully authenticated</returns>
    Task<bool> Login(string userName, string password);
}

/// <summary>
/// UI-specific service providing abilities to show alert messages.
/// </summary>
public interface IAlertService
{
    /// <summary>
    /// Show an alert message to the user.
    /// </summary>
    /// <param name="title">Alert message title</param>
    /// <param name="message">Alert message text</param>
    Task ShowAlert(string title, string message);
}
```

Building the ViewModel stub

Ok, we're gonna have the page class for Login screen, but let's start with ViewModel first:

```
public class LoginPageViewModel : BaseViewModel
{
    private readonly IAuthenticationService authenticationService;
    private readonly IAlertService alertService;

    private string userName;
    private string password;
    private bool isLoading;

    private ICommand loginCommand;

    public LoginPageViewModel(IAuthenticationService authenticationService, IAlertService
alertService)
    {
        this.authenticationService = authenticationService;
```

```
        this.alertService = alertService;
    }

    public string UserName
    {
        get
        {
            return userName;
        }
        set
        {
            if (userName!= value)
            {
                userName= value;
                OnPropertyChanged();
            }
        }
    }

    public string Password
    {
        get
        {
            return password;
        }
        set
        {
            if (password != value)
            {
                password = value;
                OnPropertyChanged();
            }
        }
    }

    public bool IsLoading
    {
        get
        {
            return isLoading;
        }
        set
        {
            if (isLoading != value)
            {
                isLoading = value;
                OnPropertyChanged();
            }
        }
    }

    public ICommand LoginCommand => loginCommand ?? (loginCommand = new Command(Login));

    private void Login()
    {
        authenticationService.Login(UserName, Password);
    }
}
```

We defined two `string` properties and a command to be bound on UI. We won't describe how to build a page class, XAML markup and bind ViewModel to it in this topic as they have nothing specific.

How to create a LoginPageViewModel instance?

I think you were probably creating the VMs just with constructor. Now as you can see our VM depends on 2 services being injected as constructor parameters so can't just do `var viewModel = new LoginPageViewModel()`. If you're not familiar with Dependency Injection it's the best moment to learn about it. Proper unit-testing is impossible without knowing and following this principle.

Tests

Now let's write some tests according to use cases listed above. First of all you need to create a new assembly (just a class library or select a special testing project if you want to use Microsoft unit testing tools). Name it something like `ProjectName.Tests` and add reference to your original PCL project.

I this example I'm going to use NUnit and Moq but you can go on with any testing libs of your choise. There will be nothing special with them.

Ok, that's the test class:

```
[TestFixture]
public class LoginPageViewModelTest
{
}
```

Writing tests

Here's the test methods for the first two scenarios. Try keeping 1 test method per 1 expected result and not to check everything in one test. That will help you to receive clearer reports about what has failed in the code.

```
[TestFixture]
public class LoginPageViewModelTest
{
    private readonly Mock<IAuthenticationService> authenticationServiceMock =
        new Mock<IAuthenticationService>();
    private readonly Mock<IAlertService> alertServiceMock =
        new Mock<IAlertService>();

    [TestCase("user", "pass")]
    public void LogInWithValidCreds_LoadingIndicatorShown(string userName, string password)
    {
        LoginPageViewModel model = CreateViewModelAndLogin(userName, password);

        Assert.IsTrue(model.IsLoading);
    }

    [TestCase("user", "pass")]
    public void LogInWithValidCreds_AuthenticationRequested(string userName, string password)
    {
        CreateViewModelAndLogin(userName, password);

        authenticationServiceMock.Verify(x => x.Login(userName, password), Times.Once);
    }

    [TestCase("", "pass")]
    [TestCase("   ", "pass")]
    [TestCase(null, "pass")]
    public void LogInWithEmptyuserName_AuthenticationNotRequested(string userName, string password)
    {
        CreateViewModelAndLogin(userName, password);
```

```
        authenticationServiceMock.Verify(x => x.Login(It.IsAny<string>(), It.IsAny<string>()),
Times.Never);
    }

    [TestCase("", "pass", "Please, enter correct username and password")]
    [TestCase("    ", "pass", "Please, enter correct username and password")]
    [TestCase(null, "pass", "Please, enter correct username and password")]
    public void LogInWithEmptyUserName_AlertMessageShown(string userName, string password, string
message)
    {
        CreateViewModelAndLogin(userName, password);

        alertServiceMock.Verify(x => x.ShowAlert(It.IsAny<string>(), message));
    }

    private LoginPageViewModel CreateViewModelAndLogin(string userName, string password)
    {
        var model = new LoginPageViewModel(
            authenticationServiceMock.Object,
            alertServiceMock.Object);

        model.UserName = userName;
        model.Password = password;

        model.LoginCommand.Execute(null);

        return model;
    }
}
```

And here we go:

Now the goal is to write correct implementation for ViewModel's `Login` method and that's it.

Business logic implementation

```
private async void Login()
{
    if (String.IsNullOrWhiteSpace(UserName) || String.IsNullOrWhiteSpace(Password))
    {
        await alertService.ShowAlert("Warning", "Please, enter correct username and password");
    }
    else
    {
        IsLoading = true;
```

```
        bool isAuthenticated = await authenticationService.Login(UserName, Password);
    }
}
```

And after running the tests again:

Now you can keep covering your code with new tests making it more stable and regression-safe.