Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Курсова робота

«Тестування Web API, яке використовує N Layered архітектуру»

з дисципліни «Компоненти програмної інженерії. Частина 4. Якість та тестування програмного забезпечення»

Виконала:

студентка групи ІМ-11
Бащак Ярина Володимирівна

Перевірив:

пос. Таран В. І.

Київ 2023

Завдання.

Покрити модульними та інтеграційними тестами основні компоненти вже розробленого Web API.

В цій курсовій роботі дане API реалізує гаманець - керування власним бюджетом. Власний бюджет складається з декількох рахунків, які можна поповнювати за заданими категоріями прибутків (наприклад, «зарплата», «відсотки з депозиту», «подарунок»). Також гроші з цих рахунків можуть бути переведені з одного на інший і витрачатись за заданими категоріями витрат (наприклад, «їжа», «одяг», «розваги»).

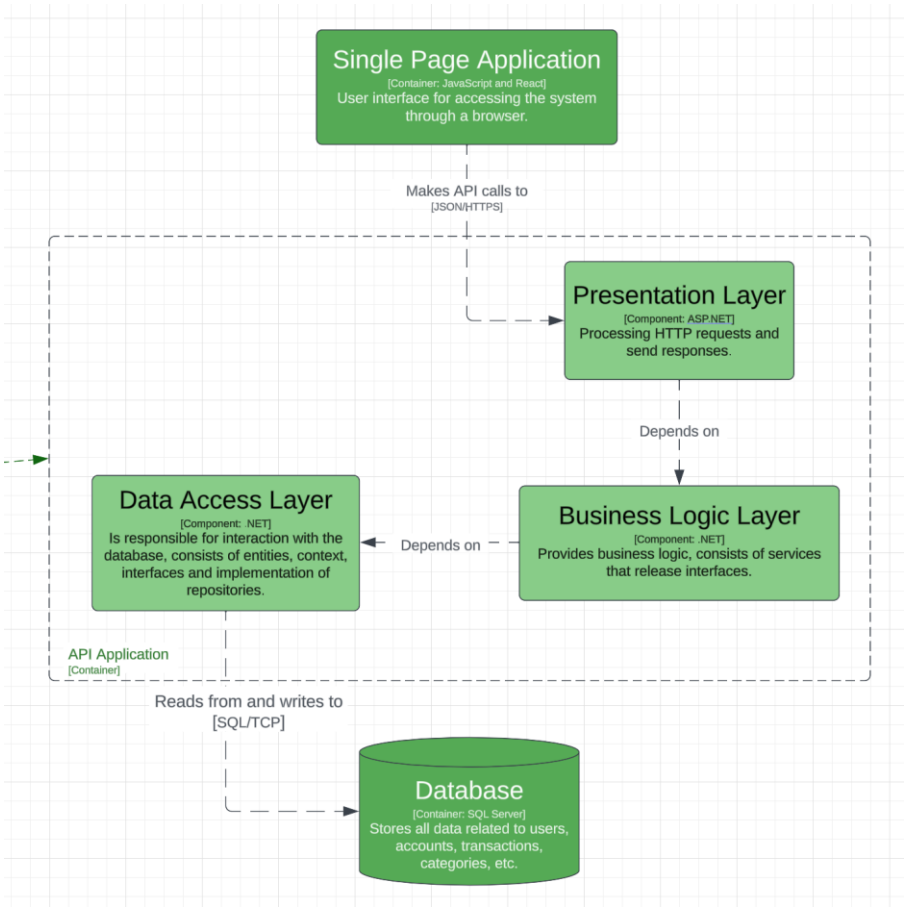
Підсумовуючи витрати та прибутки, можливо отримати інформацію скільки було витрачено або отримано загалом чи за певною категорією по заданому рахунку.

Код проекту.

[Посилання на github-репозиторій.]

Що таке N Layered архітектура і чому вона полегшує тестування?

N Layered архітектура – це поділ застосунку на шари, кожен з яких має свою область відповідальності. На рисунку нижче наведено діаграму компонентів для даного API.



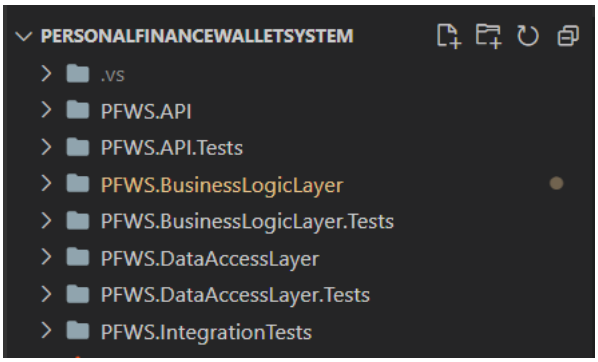
Є 3 компоненти (шари):

- Presentation Layer (тут всі контролери) – поверхнений шар, який через контролери приймає HTTP-запити і віддає відповідь;
- Business Logic Layer (тут всі сервіси) – шар, який виконує логіку додатку, складається з сервісів;
- Data Access Layer (тут всі репозиторії) – відповідає за взаємодію з базою даних та складається з контексту застосунку, сутностей і репозиторіїв.

І якраз за рахунок даної архітектури тестування можна організувати більш структуровано.

Нижче наведено структуру даної роботи, яка складається з 7-ми проектів:

- 3 проекти, що реалізують API
- 3 проекти для модульних тестів на кожен шар
- 1 проект для інтеграційних тестів.



Data Access Layer.

Цей шар містить загальний репозиторій RepositoryBase<T>. Для тестування обрано сутність Account (рахунок користувача).

Для написання тестів на нього потрібно створити моки на DbSet<Account> і WalletContext, щоб не взаємодіяти з базою даних напряму. Для моків я використала стандартну для .NET бібліотеку Moq.

В коді нижче наведено загальні налаштування цих моків в SetUp і тест для GetItemAsync. Решту тестів можна подивитися в github.

```
public class RepositoryBaseTests
{
    private Mock<DbSet<Account>> mockSet;
    private Mock<WalletContext> mockContext;
    private RepositoryBase<Account> repository;

    [SetUp]
    public void Setup()
```

```
    {
        var data = new List<Account>
        {
            new Account { Id = 1, Name = "Account 1", Balance = 1000, UserId = 1 },
            new Account { Id = 2, Name = "Account 2", Balance = 1500, UserId = 1 },
            new Account { Id = 3, Name = "Account 3", Balance = 500, UserId = 2 }
        };

        var queryableData = data.AsQueryable();

        mockSet = new Mock<DbSet<Account>>();
        mockSet.As<IQueryable<Account>>().Setup(m => m.Provider).Returns(new
TestAsyncQueryProvider<Account>(queryableData.Provider));
        mockSet.As<IQueryable<Account>>().Setup(m => m.Expression).Returns(queryableData.Expression);
        mockSet.As<IQueryable<Account>>().Setup(m => m.ElementType).Returns(queryableData.ElementType);
        mockSet.As<IQueryable<Account>>().Setup(m => m.GetEnumerator()).Returns(queryableData.GetEnumerator);

        mockSet.As<IAsyncEnumerable<Account>>().Setup(d =>
d.GetAsyncEnumerator(It.IsAny<CancellationToken>()))
            .Returns(new TestAsyncEnumerator<Account>(data.GetEnumerator()));

        mockContext = new Mock<WalletContext>();
        mockContext.Setup(c => c.Set<Account>()).Returns(mockSet.Object);
        mockContext.Setup(c => c.SaveChangesAsync(It.IsAny<CancellationToken>())).ReturnsAsync(1);

        repository = new RepositoryBase<Account>(mockContext.Object);
    }

    [Test]
    public async Task GetItemAsync_ReturnsCorrectItem()
    {
        var expectedAccount = new Account { Id = 1, Name = "Account 1", Balance = 1000, UserId = 1 };
        mockSet.Setup(m => m.FindAsync(It.IsAny<object[]>()))
               .ReturnsAsync(expectedAccount);

        var result = await repository.GetItemAsync(1);

        Assert.IsNotNull(result);
        var comparer = new AccountComparer();
        Assert.IsTrue(comparer.Equals(expectedAccount, result));
    }
```

Результати

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     5, Skipped:     0, Total:     5, Duration: 378 ms - PFWS.DataAccessLayer.Tests.dll (net7.0)
```

Business Logic Layer.

Цей шар містить інтерфейси сервісів і їх реалізацію для кожної сутності окремо. Більшість з них є стандарними, які не мають якоїсь особливої логіки, окрім одного – сервіс транзакцій, який має дуже багато валідації. Тому для тестування цього шару було обрано саме TransactionService. Тести на решту сервісів аналогічні.

Для написання тестів на нього потрібно створити моки на репозиторії, що використовуються, а саме на IRepositoryBase<Transaction>, IRepositoryBase<Account>, IRepositoryBase<Category> і IUserRepository.

В цьому звіті я наведу і поясню тести на один метод з обраного сервісу GetTransactionsByAccountId – отримання списку транзанцій для певного рахунку. Тести на решту методів можна подивитися в github.

Щоб якісно протестувати GetTransactionsByAccountId, потрібно перевірити такі випадки:

- користувач з даним іменем і рахунок існують та рахунок належить саме цьому користувачу => return list of transactions;
- не знайдено користувача з даним іменем => throw exeption;
- користувач запитує транзації рахунку, який йому не належить => throw exeption;
- не знайдено жодної транзанції для даного рахунку => return empty list;
- репозиторій повернув помилку, помилка бази даних => throw exeption;

В коді нижче наведено тести на всі ці випадки і загальні налаштування моків.

```
public class GetTransactionsByAccountIdTests
{
    private Mock<IRepositoryBase<Transaction>> _mockTransactionRepository;
    private Mock<IUserRepository> _mockUserRepository;
    private Mock<IRepositoryBase<Account>> _mockAccountRepository;
    private Mock<IRepositoryBase<Category>> _mockCategoryRepository;
    private TransactionService _transactionService;
    private string username = "testUser";
    private User user = new() { Id = 1, UserName = "testUser" };
    private Account account = new() { Id = 1, UserId = 1 };
    private Account accountOfDifferentUser = new() { Id = 3, UserId = 2 };
    private List<Category> defaultCategories = new()
    {
        new() { Id = 1, Name = "Food", Type = "expense" },
        new() { Id = 4, Name = "Salary", Type = "income" },
    };
    private List<Transaction> defaultTransactions = new()
    {
        new() { Id = 1, FromAccountId = 1, ExpenseCategoryId = 1 },
        new() { Id = 2, ToAccountId = 1, IncomeCategoryId = 4 },
    };
```

```
    [SetUp]
    public void Setup()
    {
        _mockTransactionRepository = new Mock<IRepositoryBase<Transaction>>();
        _mockUserRepository = new Mock<IUserRepository>();
        _mockAccountRepository = new Mock<IRepositoryBase<Account>>();
        _mockCategoryRepository = new Mock<IRepositoryBase<Category>>();

        _mockUserRepository.Setup(repo => repo.FindByNameAsync(username)).ReturnsAsync(user);
        _mockAccountRepository.Setup(repo => repo.GetItemAsync(account.Id)).ReturnsAsync(account);
        _mockAccountRepository.Setup(repo =>
repo.GetItemAsync(accountOfDifferentUser.Id)).ReturnsAsync(accountOfDifferentUser);
        _mockCategoryRepository.Setup(repo => repo.FindByConditionAsync(It.IsAny<Expression<Func<Category,
bool>>>())).ReturnsAsync(defaultCategories);
        _mockTransactionRepository.Setup(repo =>
repo.FindByConditionAsync(It.IsAny<Expression<Func<Transaction,
bool>>>())).ReturnsAsync(defaultTransactions);

        _transactionService = new TransactionService(_mockTransactionRepository.Object,
_mockUserRepository.Object, _mockAccountRepository.Object, _mockCategoryRepository.Object);
    }

    [Test]
    public async Task GetTransactionsByAccountId_WhenUserAndAccountExist_ReturnsTransactions()
    {
        var result = await _transactionService.GetTransactionsByAccountId(account.Id, username);

        Assert.IsNotNull(result);
        Assert.IsNotEmpty(result);
        Assert.That(result.Count, Is.EqualTo(defaultTransactions.Count));
        Assert.That(result[0].Id, Is.EqualTo(1));
        Assert.That(result[1].Id, Is.EqualTo(2));
        Assert.That(result[0].ExpenseCategory.Id, Is.EqualTo(defaultTransactions.First(t => t.Id ==
1).ExpenseCategoryId));
        Assert.That(result[1].IncomeCategory.Id, Is.EqualTo(defaultTransactions.First(t => t.Id ==
2).IncomeCategoryId));
    }

    [Test]
    public void GetTransactionsByAccountId_WhenUserNotFound_ThrowsException()
    {
        string username = "nonExistentUser";

        Assert.ThrowsAsync<Exception>(async () => await
_transactionService.GetTransactionsByAccountId(account.Id, username));
    }

    [Test]
    public void GetTransactionsByAccountId_WhenAccountNotFound_ThrowsException()
    {
        int accountId = 10;
```

```
        Assert.ThrowsAsync<Exception>(async () => await
_transactionService.GetTransactionsByAccountId(accountId, username));
    }

    [Test]
    public void GetTransactionsByAccountId_WhenUserNotAuthorized_ThrowsException()
    {
        int accountId = 3; // account of different user

        Assert.ThrowsAsync<Exception>(async () => await
_transactionService.GetTransactionsByAccountId(accountId, username));
    }

    [Test]
    public async Task GetTransactionsByAccountId_WhenNoTransactionsFound_ReturnsEmptyList()
    {
        _mockTransactionRepository.Setup(repo =>
repo.FindByConditionAsync(It.IsAny<Expression<Func<Transaction, bool>>>())).ReturnsAsync(new
List<Transaction>());

        var result = await _transactionService.GetTransactionsByAccountId(account.Id, username);

        Assert.IsNotNull(result);
        Assert.IsEmpty(result);
    }

    [Test]
    public void GetTransactionsByAccountId_WhenRepositoryThrowsException_ThrowsException()
    {
        _mockTransactionRepository.Setup(repo =>
repo.FindByConditionAsync(It.IsAny<Expression<Func<Transaction, bool>>>())).ThrowsAsync(new
Exception("Database error"));

        Assert.ThrowsAsync<Exception>(async () => await
_transactionService.GetTransactionsByAccountId(account.Id, username));
    }
}
```

Результати

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:    26, Skipped:     0, Total:    26, Duration: 223 ms - PFWS.BusinessLogicLayer.Tests.dll (net7.0)
```

Presentation Layer.

Цей шар містить реалізацію контролерів. Всі вони є стандарними. Я написала тести на одного з них – на AccountsController (контролер рахунків). Тести на решту контролерів аналочні.

Для написання тестів на нього потрібно створити моки на IAccountService і налаштувати HttpContext.

```
    private static void SetupControllerHttpContext(ControllerBase controller, string username)
    {
        var context = new DefaultHttpContext();
        var claims = new List<Claim> { new Claim(ClaimTypes.Name, username) };
        var identity = new ClaimsIdentity(claims);
        var principal = new ClaimsPrincipal(identity);
        context.User = principal;

        controller.ControllerContext = new ControllerContext
        {
            HttpContext = context
        };
    }
```

Це потрібно, тому що кожна кінцева точка, яка потребує авторизації, отримує ім’я користувача з User.Identity.Name і передає його у сервіс. Наприклад,

```
    [HttpGet("{id}")]
    public async Task<ActionResult<GetAccountDto>> GetAccountById(int id)
    {
        try
        {
            var username = User.Identity.Name;
            var account = await _accountService.GetAccountById(id, username);
            return Ok(account);
        }
        catch (Exception ex)
        {
            return BadRequest(ex.Message);
        }
    }
```

В коді нижче наведено тести на кожну кінцеву точку контролера рахунків для двох випадків: відповідь успішно отримано і під час виконання виникла виключна ситуація, яку має відловити catch.

```
public class AccountsControllerTests
{
    private Mock<IAccountService> mockService;
    private AccountsController controller;
    private readonly string username = "testUser";
```

```
[SetUp]
public void Setup()
{
    mockService = new();
    controller = new(mockService.Object);
    SetupControllerHttpContext(controller, username);
}

[Test]
public async Task GetAccountById_WithValidAccount_ReturnsOk()
{
    var expectedAccount = new GetAccountDto { Id = 1, UserId = 1, Balance = 1000 };
    mockService.Setup(s => s.GetAccountById(It.IsAny<int>(), username))
               .ReturnsAsync(expectedAccount);

    var result = await controller.GetAccountById(1);

    Assert.IsInstanceOf<OkObjectResult>(result.Result);
    var okResult = result.Result as OkObjectResult;
    Assert.That(okResult?.Value, Is.EqualTo(expectedAccount));
}

[Test]
public async Task GetUserAccounts_WithValidAccounts_ReturnsOk()
{
    var expectedAccounts = new List<GetAccountDto>
    {
        new GetAccountDto { Id = 1, UserId = 1, Balance = 1000 },
        new GetAccountDto { Id = 2, UserId = 1, Balance = 2000 }
    };

    mockService.Setup(s => s.GetUserAccounts(username))
               .ReturnsAsync(expectedAccounts);

    var result = await controller.GetUserAccounts();

    Assert.IsInstanceOf<OkObjectResult>(result.Result);
    var okResult = result.Result as OkObjectResult;
    Assert.That(okResult?.Value, Is.EqualTo(expectedAccounts));
}

[Test]
public async Task AddAccount_WithValidAccount_ReturnsOk()
{
    var newAccount = new AddAccountDto { Name = "Account name", Balance = 1000 };
    mockService.Setup(s => s.AddAccount(It.IsAny<AddAccountDto>(), username))
               .Returns(Task.CompletedTask);

    var result = await controller.AddAccount(newAccount);

    Assert.IsInstanceOf<OkResult>(result);
}
```

```
[Test]
public async Task UpdateAccount_WithValidAccount_ReturnsNoContent()
{
    var updatedAccount = new UpdateAccountDto { Name = "Updated account name" };
    mockService.Setup(s => s.UpdateAccount(It.IsAny<int>(), It.IsAny<UpdateAccountDto>(), username))
               .Returns(Task.CompletedTask);

    var result = await controller.UpdateAccount(1, updatedAccount);

    Assert.IsInstanceOf<NoContentResult>(result);
}

[Test]
public async Task DeleteAccount_WithValidId_ReturnsNoContent()
{
    mockService.Setup(s => s.DeleteAccount(It.IsAny<int>(), username))
               .Returns(Task.CompletedTask);

    var result = await controller.DeleteAccount(1);

    Assert.IsInstanceOf<NoContentResult>(result);
}

[Test]
public async Task GetAccountById_WhenExceptionThrown_ReturnsBadRequest()
{
    var exceptionMessage = "Error retrieving account";
    mockService.Setup(s => s.GetAccountById(It.IsAny<int>(), It.IsAny<string>()))
               .ThrowsAsync(new Exception(exceptionMessage));

    var result = await controller.GetAccountById(1);

    Assert.IsInstanceOf<BadRequestObjectResult>(result.Result);
    var badRequestResult = result.Result as BadRequestObjectResult;
    Assert.That(badRequestResult?.Value, Is.EqualTo(exceptionMessage));
}

[Test]
public async Task GetUserAccounts_WhenExceptionThrown_ReturnsBadRequest()
{
    var exceptionMessage = "Error retrieving accounts";
    mockService.Setup(s => s.GetUserAccounts(It.IsAny<string>()))
               .ThrowsAsync(new Exception(exceptionMessage));

    var result = await controller.GetUserAccounts();

    Assert.IsInstanceOf<BadRequestObjectResult>(result.Result);
    var badRequestResult = result.Result as BadRequestObjectResult;
    Assert.That(badRequestResult?.Value, Is.EqualTo(exceptionMessage));
}

[Test]
public async Task AddAccount_WhenExceptionThrown_ReturnsBadRequest()
```

```
    {
        var newAccount = new AddAccountDto { Balance = 1000 };
        var exceptionMessage = "Error adding account";
        mockService.Setup(s => s.AddAccount(It.IsAny<AddAccountDto>(), It.IsAny<string>()))
                   .ThrowsAsync(new Exception(exceptionMessage));

        var result = await controller.AddAccount(newAccount);

        Assert.IsInstanceOf<BadRequestObjectResult>(result);
        var badRequestResult = result as BadRequestObjectResult;
        Assert.That(badRequestResult?.Value, Is.EqualTo(exceptionMessage));
    }

    [Test]
    public async Task UpdateAccount_WhenExceptionThrown_ReturnsBadRequest()
    {
        var updatedAccount = new UpdateAccountDto { Name = "Updated account name"};
        var exceptionMessage = "Error updating account";
        mockService.Setup(s => s.UpdateAccount(It.IsAny<int>(), It.IsAny<UpdateAccountDto>(),
It.IsAny<string>()))
                   .ThrowsAsync(new Exception(exceptionMessage));

        var result = await controller.UpdateAccount(1, updatedAccount);

        Assert.IsInstanceOf<BadRequestObjectResult>(result);
        var badRequestResult = result as BadRequestObjectResult;
        Assert.That(badRequestResult?.Value, Is.EqualTo(exceptionMessage));
    }

    [Test]
    public async Task DeleteAccount_WhenExceptionThrown_ReturnsBadRequest()
    {
        var exceptionMessage = "Error deleting account";
        mockService.Setup(s => s.DeleteAccount(It.IsAny<int>(), It.IsAny<string>()))
                   .ThrowsAsync(new Exception(exceptionMessage));

        var result = await controller.DeleteAccount(1);

        Assert.IsInstanceOf<BadRequestObjectResult>(result);
        var badRequestResult = result as BadRequestObjectResult;
        Assert.That(badRequestResult?.Value, Is.EqualTo(exceptionMessage));
    }

    private static void SetupControllerHttpContext(ControllerBase controller, string username)
    {
        var context = new DefaultHttpContext();
        var claims = new List<Claim> { new Claim(ClaimTypes.Name, username) };
        var identity = new ClaimsIdentity(claims);
        var principal = new ClaimsPrincipal(identity);
        context.User = principal;

        controller.ControllerContext = new ControllerContext
        {
```

```
            HttpContext = context
        };
    }
}
```

Результати

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:    10, Skipped:     0, Total:    10, Duration: 232 ms - PFWS.API.Tests.dll (net7.0)
```

Інтеграційні тести.

Для написання інтеграційних тестів потрібно:

- створити моки репозиторіїв і задати їм потрібні налаштування, щоб напряму не взаємодіяти з базою даних;
- створити мок сервісу, в який передати раніше створені моки репозиторіїв, не задаючи ніяких додаткових налаштувань;
- створити екземпляр контролера та передати йому в параметри цей мок сервіса;
- налаштувати HttpContext, тому що контролер його використовує для отримання імені поточного користувача.

Таким чином ми тестуємо модулі разом.

В коді нижче наведено всі ці налаштування, про які було сказано раніше, та інтеграційні тести на Account (рахунок).

```
[TestFixture]
public class AccountIntegrationTests
{
    private Mock<AccountService> mockService;
    private Mock<IRepositoryBase<Account>> mockAccountRepository;
    private Mock<IUserRepository> mockUserRepository;
    private AccountsController controller;
    private readonly string username = "testUser";
    private readonly User expectedUser = new() { Id = 1, UserName = "testUser" };
    private readonly Account dbAccount = new() { Id = 1, UserId = 1, Balance = 1000 };

    [SetUp]
    public void Setup()
    {
        mockAccountRepository = new();
        mockUserRepository = new();

        mockUserRepository.Setup(s => s.FindByNameAsync(username))
                          .ReturnsAsync(expectedUser);
        mockAccountRepository.Setup(s => s.GetItemAsync(It.IsAny<int>()))
                    .ReturnsAsync(dbAccount);
```

```
        mockService = new(mockAccountRepository.Object, mockUserRepository.Object);
        controller = new(mockService.Object);
        SetupControllerHttpContext(controller, username);
    }

    [Test]
    public async Task GetAccountById_ReturnsCorrectData()
    {
        var dbAccount = new Account { Id = 1, UserId = 1, Balance = 1000 };

        var result = await controller.GetAccountById(1);

        var expectedAccount = new GetAccountDto { Id = 1, UserId = 1, Balance = 1000 };
        Assert.IsInstanceOf<OkObjectResult>(result.Result);
        var okResult = result.Result as OkObjectResult;
        var responseAccount = okResult?.Value as GetAccountDto;
        Assert.That(responseAccount?.Id, Is.EqualTo(expectedAccount.Id));
        Assert.That(responseAccount?.UserId, Is.EqualTo(expectedAccount.UserId));
        Assert.That(responseAccount?.Balance, Is.EqualTo(expectedAccount.Balance));
    }

    [Test]
    public async Task GetUserAccounts_ReturnsCorrectData()
    {
        var dbAccounts = new List<Account>
        {
            new Account { Id = 1, UserId = 1, Balance = 1000 },
            new Account { Id = 2, UserId = 1, Balance = 500 },
            new Account { Id = 3, UserId = 1, Balance = 1500 },
        };
        mockAccountRepository.Setup(s => s.FindByConditionAsync(It.IsAny<Expression<Func<Account, bool>>>()))
                             .ReturnsAsync(dbAccounts);

        var result = await controller.GetUserAccounts();

        var expectedAccounts = new List<GetAccountDto>
        {
            new GetAccountDto { Id = 1, UserId = 1, Balance = 1000 },
            new GetAccountDto { Id = 2, UserId = 1, Balance = 500 },
            new GetAccountDto { Id = 3, UserId = 1, Balance = 1500 },
        };

        Assert.IsInstanceOf<OkObjectResult>(result.Result);
        var okResult = result.Result as OkObjectResult;
        var responseAccount = okResult?.Value as List<GetAccountDto>;
        Assert.That(responseAccount?.Count, Is.EqualTo(expectedAccounts.Count));
        Assert.That(responseAccount?[0].UserId, Is.EqualTo(expectedAccounts[0].UserId));
        Assert.That(responseAccount?[1].UserId, Is.EqualTo(expectedAccounts[1].UserId));
        Assert.That(responseAccount?[2].UserId, Is.EqualTo(expectedAccounts[2].UserId));
    }

    [Test]
```

```
    public async Task AddAccount_ReturnsCorrectData()
    {
        var addAccountDto = new AddAccountDto { Balance = 1000, Name = "New account" };
        mockAccountRepository.Setup(s => s.AddItemAsync(It.IsAny<Account>()))
                     .Returns(Task.CompletedTask);

        var result = await controller.AddAccount(addAccountDto);

        Assert.IsInstanceOf<OkResult>(result);
    }

    [Test]
    public async Task UpdateAccount_ReturnsCorrectData()
    {
        var accountId = 1;
        var updatedAccountDto = new UpdateAccountDto { Name = "Updated account" };
        mockAccountRepository.Setup(s => s.UpdateItemAsync(It.IsAny<int>(), It.IsAny<Account>()))
                     .Returns(Task.CompletedTask);

        var result = await controller.UpdateAccount(accountId, updatedAccountDto);

        Assert.IsInstanceOf<NoContentResult>(result);
    }

    [Test]
    public async Task DeleteAccount_ReturnsCorrectData()
    {
        var accountId = 1;
        mockAccountRepository.Setup(s => s.DeleteItemAsync(It.IsAny<Account>()))
                     .Returns(Task.CompletedTask);

        var result = await controller.DeleteAccount(accountId);

        Assert.IsInstanceOf<NoContentResult>(result);
    }

    private static void SetupControllerHttpContext(ControllerBase controller, string username)
    {
        var context = new DefaultHttpContext();
        var claims = new List<Claim> { new Claim(ClaimTypes.Name, username) };
        var identity = new ClaimsIdentity(claims);
        var principal = new ClaimsPrincipal(identity);
        context.User = principal;

        controller.ControllerContext = new ControllerContext
        {
            HttpContext = context
        };
    }
}
```

Результати

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     5, Skipped:     0, Total:     5, Duration: 208 ms - PFWS.IntegrationTests.dll (net7.0)
```

Висновки: отже, в даній курсовій роботі виконано тестування кожного з шарів Web API окремо у модульних тестах, а також їх роботу разом в інтеграційних. Під час виконання я особливо багато попрактикувалася у створенні моків та їх налаштуванню. Було написано тести на загальний репозиторій на прикладі сутності рахунку, на сервіс транзакцій, який має багато валідації, що відобразилось на кількості написаних тестів, та на контролер рахунків.

Результати всіх тестів одночасно.

```
Passed!  - Failed:     0, Passed:     5, Skipped:     0, Total:     5, Duration: 708 ms - PFWS.DataAccessLayer.Tests.dll (net7.0)
Passed!  - Failed:     0, Passed:    26, Skipped:     0, Total:    26, Duration: 507 ms - PFWS.BusinessLogicLayer.Tests.dll (net7.0)
Passed!  - Failed:     0, Passed:    10, Skipped:     0, Total:    10, Duration: 128 ms - PFWS.API.Tests.dll (net7.0)
Passed!  - Failed:     0, Passed:     5, Skipped:     0, Total:     5, Duration: 151 ms - PFWS.IntegrationTests.dll (net7.0)
```