Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка веб-додатку для оптимізації ведення бюджету із використанням C#

Виконав: студент IV курсу, групи СП-41
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Барабаш В. О.
(підпис) (прізвище та ініціали)

Керівник Стоянов Ю. М.
(підпис) (прізвище та ініціали)

Нормоконтроль Стоянов Ю. М.
(підпис) (прізвище та ініціали)

Завідувач кафедри Петрик М. Р.
(підпис) (прізвище та ініціали)

Рецензент
(підпис) (прізвище та ініціали)

Тернопіль
2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р
(підпис) (прізвище та ініціали)
«__» _____ 2024 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Барабаш Віталій Олегович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка веб-додатку для оптимізації ведення бюджету із використанням C#

Керівник роботи к.т.н., ст. викладач каф. ПІ Стоянов Ю. М.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*15*» *квітня* 2024 року № *4/7-341*

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи Предметна область, технічне завдання, вимоги та специфікація, програмне рішення

4. Зміст роботи (перелік питань, які потрібно розробити): Вступ. Розділ 1. Аналіз предметної області та задачі. Розділ 2. Проєктування програмного рішення. Розділ 3. Тестування програмного рішення. Розділ 4.Безпека життєдіяльності та основи охорони праці. Висновок. Список використаних джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів): Ілюстративні зображення, інформативні зображення для доповнення тексту, діаграми, знімки екрану з проробленою роботою.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності та основи охорони праці	к. т. н. доцент каф. МТ Лазарюк В. В.		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи		
2.	Підбір джерел по темі кваліфікаційної роботи		
3.	Опрацювання джерел по темі кваліфікаційної роботи		
4.	Аналіз та підготовка технічної специфікації		
5.	Розробка архітектури програмного рішення		
6.	Розробка шару доступу до даних програмного рішення		
7.	Розробка шару бізнес логіки програмного рішення		
8.	Розробка шару користувацького інтерфейсу програмного рішення		
9.	Виконання звіту по проробленій роботі		
10.	Оформлення кваліфікаційної роботи		
11.	Виконання завдання до підрозділу «Безпека життєдіяльності»		
12.	Виконання завдання до підрозділу «Основи охорони праці»		
13.	Нормоконтроль		
14.	Перевірка на плагіат		
15.	Попередній захист кваліфікаційної роботи		
16.	Захист кваліфікаційної роботи		

Студент _____ Барабаш В. О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Стоянов Ю. М.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Розробка веб-додатку для оптимізації ведення бюджету із використанням C# // Кваліфікаційна робота освітнього рівня «Бакалавр» // Барабаш Віталій Олегович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-41 // Тернопіль, 2024 // С. – 77, рис. – 40, табл. – 0, кресл. – 0, додат. – 2, бібліогр. – 18.

Ключові слова: програмне рішення, фінанси, бюджет, бюджетування, облік, C#, ASP.NET Core, база даних, Entity Framework Core, SQL Server, SQL, GIT, JetBrains Rider, Unit тести, xUnit.

Кваліфікаційна робота досліджує процеси аналізу, проєктування, розробку та тестування програмного рішення веб-додатку з оптимізації бюджетування передовими технологіями та інструментами в сфері розробки.

В першому розділі кваліфікаційної роботи досліджено предметну область, конкурентні рішення, проаналізовано вимоги і засоби та інструменти розробки. Розділ дозволяє ознайомитись з темою та актуальністю проблематики цієї теми.

В другому розділі кваліфікаційної роботи описано процеси вибору архітектури рішення, проєктування та програмної реалізації. Розділ дозволяє ознайомитись з процесами безпосередньої розробки на основі даних із першого розділу.

В третьому розділі кваліфікаційної роботи описано тестування та верифікацію розробленого програмного рішення. Розділ дозволяє ознайомитись з інструментами та підходами до тестування і сформулювати кінцеву оцінку задовільності виконаної роботи.

ANNOTATION

Development of a web application for optimising budgeting using C# // Qualification work for the educational level "Bachelor" / Barabash Vitaliy Olehovych // Ternopil National Technical University named after Ivan Puluj, Faculty of Computer Information Systems and Software Engineering, Department of Software Engineering, group SP-41 // Ternopil, 2024 // P. - 77, fig. - 40, tabl. - 0, drawings - 0, addition - 2, literature - 18.

Keywords: software solution, finance, budget, budgeting, accounting, C#, ASP.NET Core, database, Entity Framework Core, SQL Server, SQL, GIT, JetBrains Rider, Unit tests, xUnit.

The qualification work explores the processes of analyzing, designing, developing and testing a software solution for a web application for budgeting optimization using advanced technologies and tools in the field of development.

The first section of the qualification work explores the subject area, competitive solutions, analyzes requirements and development tools and techniques. The section allows you to familiarize yourself with the topic and the relevance of the issues of this topic.

The second section of the qualification work describes the processes of choosing a solution architecture, design, and program implementation. The section allows you to familiarize yourself with the processes of direct development based on the data from the first section.

The third section of the qualification paper describes the testing and verification of the developed software solution. The section allows you to familiarize yourself with the tools and approaches to testing and formulate a final assessment of the satisfactory performance of the work performed

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Фінанси – це галузь міжлюдських економічних відносин у сфері розподілу та використання капіталу, суспільно-національних економічних багатств, надбань.

Бюджет – це грошове вираження економічної стратегії ефективного розподілу витрат та надходжень протягом певного періоду часу.

.NET – це платформа із засобами розробки програмних рішень із відкритим кодом, що розроблена Microsoft.

C# – це мова програмування, що є частиною платформи .NET, з об’єктно-орієнтованою парадигмою.

SQL (Structured Query Language) – це мова запитів для взаємодії з базами даних.

БД (База даних) – це сховище даних, що організоване у певному форматі та зберігається й обробляється за допомогою комп’ютерної системи.

MS SQL Server – це інструмент для оптимізації керування базою даних, що розроблений Microsoft.

EF Core (Entity Framework Core) – це засіб об’єктно-реляційного відображення для взаємодії з базою даних на платформі .NET.

ASP.NET Core – це фреймворк із засобам розробки веб-додатків із відкритим кодом.

HTML (HyperText Markup Language) – це мова розмітки структури веб-сторінок.

CSS (Cascading Style Sheets) – це мова стилів для оформлення вигляду веб-сторінок.

JS (JavaScript) – це мова програмування для встановлення динамічної поведінки веб-сторінки з користувачем.

Bootstrap – це фреймворк із заздалегідь написаними стилями та скриптами.

MVC (Model-View-Controller) – це архітектурний патерн проєктування програмного забезпечення широко застосовуваний в розробці веб-додатків.

GIT – це розподілена система призначена для ведення управління версіями.

GITHUB – це веб-платформа для збереження та управління локальною версією GIT.

Unit тестування – це процес перевірки «блоків» коду, виокремлених від іншої частини програмного забезпечення.

xUnit – це фреймворк із засобами реалізації процесу Unit тестування в межах мови програмування C#.

IDE (Integrated Development Environment) – це спеціалізований інструмент-редактор для роботи з кодом мови програмування.

JetBrains Raider – це інтегроване середовище розробки призначене для роботи з проєктами

UML (Unified Modeling Language) – це мова моделювання для встановлення візуального представлення документації програмних систем.

ЗМІСТ

ВСТУП.....9
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАДАЧІ.....10
1.1 Постановка задачі.....10
1.2 Аналіз та актуальність предметної області.....12
1.3 Огляд конкурентних рішень.....14
1.4 Огляд вимог.....17
1.5 Технології та інструменти розробки.....20
РОЗДІЛ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО РІШЕННЯ.....23
2.1 Архітектура програмного рішення.....23
2.2 Проєктування сутностей системи та бази даних.....25
2.3 Розробка користувацького інтерфейсу.....30
2.4 Розробка логіки системи.....34
РОЗДІЛ 3. ТЕСТУВАННЯ ПРОГРАМНОГО РІШЕННЯ.....40
3.1 Стратегія тестування.....40
3.2 Розробка та застосування тестування.....43
РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ.....46
4.1 Менеджмент безпеки життєдіяльності.....46
4.2 Вимоги ергономіки до організації робочого місця ПК.....48
ВИСНОВКИ.....52
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....54
ДОДАТКИ.....56
Додаток А – Лістинг коду основних компонентів.....57
Додаток Б – Диск з роботою.....77

ВСТУП

У сучасному цифровому світі, що швидко розвивається, ключовим фактором досягнення персонального благополуччя та зростання, є уміння ефективно розпоряджатись фінансами. З плином часу, фінансові зобов'язання складнішають, різноманіття операції збільшується, а традиційні методи не поспівають за потребами. Отож, попит на інтуїтивні та ефективні інструменти бюджетування збільшується.

Із застосуванням сучасних засобів та заходів, вирішення такої складної задачі спрощується до створення веб-додатку із зв’язкою мови програмування C# та веб-фреймворку ASP.NET Core, що є потужною та перспективною платформою для розробки програмного рішення та стане фундаментом для проведення дослідження процесу розробки веб-додатку для цієї кваліфікаційної роботи.

Основною метою цієї кваліфікаційної роботи є проєктування, розробка та реалізації веб-додатку із застосуванням передових технологій мови програмування C# та веб-фреймворку ASP.NET Core для оптимізації бюджетування, що надає користувачам інтуїтивно зрозумілий інструмент для ефективного управління власними фінансами.

В межах кваліфікаційної різні планується зачепити різні етапи розробки із передовими підходами до проєктування та впровадження. Також, основною метою у всіх деталях оглянути етапи аналізу, моделювання, проєктування, відладки та тестування.

Очікується, що висновки та ідеї, отримані в результаті виконання цього дослідження являтимуться достатньо цінними внесками у сферу фінансових технологій та сферу розробки комплексних веб-додатків.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ЗАДАЧІ

1.1 Постановка задачі

У контексті сучасного економічного становища, ведення бюджету є ключовим аспектом фінансового життя кожного учасника суспільства, від окремих осіб до цілих держав. Тому, навички та уміння доцільного використання фінансів є надважливими й здатні позитивно вплинути на фінансове становище та якість життя, покращуючи побутове, соціальне, фізичне, психологічне благополуччя.

Попри перелічені переваги більшість людей уникають процесів бюджетування, адже традиційні методи, що ілюстративно наведені на рисунку 1.1, фактично нівелюють позитивні ефекти потребою в клопітливій, неефективній та складній роботі з купою залежних, важких та дуже чутливих даних. Тому й виникає потреба в засобах для спрощення та оптимізації бюджетування, яким являтиметься розроблюване в межах кваліфікаційної роботи програмне рішення.

Рисунок 1.1 – Процеси та засоби традиційних методів бюджетування

Тобто, фактична мета та задача кваліфікаційної роботи за темою “Розробка веб-додатку для оптимізації ведення бюджету із використанням C#” є

забезпечення, засобами та заходами сучасного створення програмного забезпечення, надійного та функціонально-ефективного веб-додатку на базі мови програмування C# [1] і платформи ASP.NET Core [2], що вирішуватиме наступні задачі користувача:

1. Створення, збереження та маніпулювання даними рахунків;
2. Створення, збереження та маніпулювання даними бюджетів;
3. Створення, збереження та маніпулювання даними транзакцій;
4. Забезпечення безпечного доступ до даних через користувацький обліковий запис;
5. Надійна та ефективна логіка, що пов’язуватиме дані та забезпечуватиме точність;
6. Доступний та комфортний інтерфейс користувача.

Отож, проблематика та задача цієї роботи є дослідження теоретичного та практичного етапів розробки програмного рішення з використанням стеку сучасних технологій для досягнення кінцевої мети в форматі функціонального програмного забезпечення, що має структурований та підтримуваний, масштабований та модульний код і придатне до подальшого доопрацювання та модифікації, процеси яких ілюстративно наведено на рисунку 1.2.

Рисунок 1.2 – Процеси та складові розробки програмного забезпечення

Отже, можна констатувати факт, що мета та ідея поставленої задачі в поетапному виконанні дозволить дослідити та описати процеси розробки

програмного забезпечення, а також представити власне програмне рішення, що потенційно може стати суттєвим внеском в предметну область.

1.2 Аналіз та актуальність предметної області

Розробка програмного забезпечення є комплексним та ресурсоємним процесом, що потребує великих об’ємів вхідної інформації про розробку, як от:

- цілі та завдання програмного забезпечення;
- вимоги та побажання замовника;
- сферу застосування та цільову аудиторія кінцевого продукту;
- плани та перспективи.

Кожна деталь та крихта інформації дозволяє ретельніше та якісніше підготуватись до майбутніх викликів етапів планування, розробки та тестування. Тому, етап аналіз предметної області є надважливим та дозволяє покращити якість та ефективність розробки веб-додатку, що має на меті вирішувати визначену проблематику.

Тож, предметною областю конкретно цієї кваліфікаційної роботи є сфера фінансів, а точніше процеси бюджетування та обліковування фінансових ресурсів. Тому в межах підрозділу розглядатимуться фінансові та економічні терміни та явища, для подальшого застосування в розробці.

Сфера фінансів – це давній рукотворний конструкт, який виник від потреби в чіткому регламентуванні взаємодії людини з людиною в процесах перерозподілу матеріальних та нематеріальних ресурсів. Сфера фінансів є сукупним образом усіх підпроцесів: від укладення договорів та формування зобов'язань до безпосереднього виконання робіт та оплату за них. Таким чином, сфера фінансів є де-факто складової усіх інших сфер. На рисунку 1.3 наведено ілюстративне зображення процесів та складових фінансової сфери.

Рисунок 1.3 – Процеси та складові фінансової сфери

Бюджетування є фундаментальна складова фінансової сфери, що охоплює й поєднує купу процесів у нерозривний ланцюжок, де невиконання однієї з ланок неминуче призведе до провалу інших та безпосередньо стосується систематичного розподілу з документуванням надходжень та витрат для розуміння перспектив досягнення поставлених цілей та формування подальших планів.

Облік та класифікація фінансових потоків є фактором, що безпосередньо впливає на процес бюджетування та де-факто є підґрунтям для інших процесів в межах бюджетування. Тобто, за відсутності достатньо деталізованого обліку неможливо будувати бюджет, адже інформація потрібна для цього відсутня. Майже аналогічна ситуація з класифікацією, що ускладнює процеси аналізу та планування.

Загалом, ефективний фінансовий менеджмент є дечим більшим чим тільки бюджетування з підпроцесами закладених в нього. Це комплекс дій із застосуванням фінансових стратегій та тактик для оптимізації використання фінансових ресурсів з метою примноження капіталу та його використання в цілях покращення благополуччя.

Таким чином, фінансова сфера є доцільним об’єктом для введення програмного забезпечення, як проміжної ланки для виконання складних операції із кількома пов’язаними змінними в реальному часі зі надійним збереженням даних. Використання саме веб-додатків є можливістю підготувати легкий та комфортний інтерфейс і забезпечити користувача постійним доступом до даних. З урахуванням важливості фінансів у повсякденному житті варто зазначити, що дані в межах додатку повинні бути актуальними і точними, а також убезпеченими від стороннього впливу.

Отож, сфера фінансів є дуже непростою та вибагливою предметною областю. Однак, достатнє розуміння сфери, термінології та явищ дозволяє отримати більш точне бачення проєкту і краще підготуватися до нагальних потреб, вимог та уникнути різних помилок, що неминуче призвели б до ускладнення розробки та збільшення її ресурсозатратності й найголовніше зіпсували б користувацький досвід у використанні розроблюваного веб-додатку.

1.3 Огляд конкурентних рішень

Ринок програмного забезпечення для фінансової сфери та оптимізації бюджетування не є відкриттям для компаній та розробників, адже налічує уже безліч різноманітних та різнонаправлених додатків.

Одним із можливих конкурентних рішень у царині веб-додатків може стати програмне рішення від української стартап компанії Saldo Apps [3], що входить до холдингу Netpeak Group, та є розробником програмного забезпечення для різного рівня автоматизації процесів обліку фінансової діяльності Saldo Finance [4]. Також, компанія має кілька інших фінансово технологічних розробок поєднаних в одну екосистему та напрямлених на полегшення ведення бухгалтерії для малого бізнесу від Америки до Азії [3].

Рисунок 1.4 – Логотип компанії розробника конкурентного рішення

Saldo Finance – це програмне рішення представлене у форматі веб-додатку та мобільного застосунку, надаючи користувачам обширний функціонал із бюджетування та планування, відстежування та класифікації транзакції, облік та аналіз капіталу [4].

Користувачі можуть використовувати зручний та привабливий веб-інтерфейс(рис. 1.5) після входу до власного облікового запису. Обліковий запис, дозволяє зберігати власні фінансові дані в зручному форматі розділивши їх на три логічні групи: рахунки, транзакції, бюджети. Кожна з груп відповідає за детальне та точне відображення визначеної заздалегідь інформації.

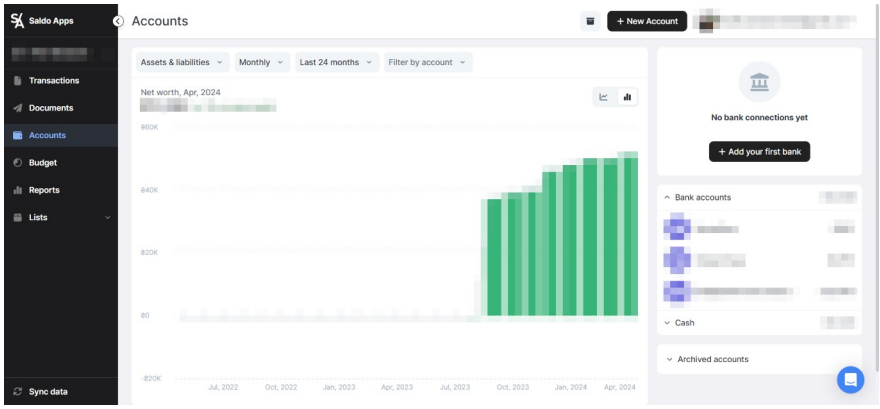

Рисунок 1.5 – Інтерфейс головної сторінки веб-додатку

Також, користувачеві доступна детальна інфографіка по обраній категорії, або загальні стовпчикові чи кругові діаграми по фінансовій активності. Таким

чином дозволяючи в ілюстративній формі проаналізувати дані, що є комфортнішим та зручнішим способом ніж перегляд купи табличних записів з безліччю даних.

Щодо роботи користувача з даними, то програмне рішення має зручні бокові панелі, відповідно для кожної з груп, що дозволяють зручно та легко вносити нові записи, редагувати чи видаляти наявні(рис. 1.6). Також, такий підхід дозволяє уникнути потреби в різних веб-сторінках, що не мали б достатнього наповнення і виглядали б пусто.

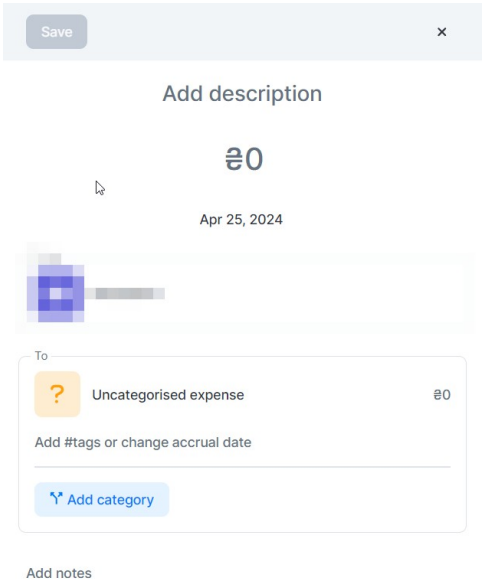

Рисунок 1.6 – Бічна форма для внесення даних

Підсумовуючи усе описане вище можна підмітити, що це програмне рішення є дуже якісним та надійним веб-додатком. Додаток реалізовує зручний та повноцінний інтерфейс, що наповнений корисною і важливою інформацією. Щодо функціональної сторони, то тут веб-додаток також є зразком хорошої роботи команди розробки, яка створила ефективну систему із якісною та надійною передачею даних і їх розподілом до потрібних вузлів програми.

Отже, можна визнати, що програмне рішення є суттєвим гравцем в ніші фінансового програмного забезпечення та підходи і засоби використані в цьому веб-додатку є вартими до уваги і могли б чудово бути застосовані в розробці програмного рішення цієї кваліфікаційної роботи.

1.4 Огляд вимог

Розробка програмного рішення в якості повноцінного проєкту завжди є етапом залежним від успіху раніше виконаних етапів підготовки з подальшою конвертацією їх у вимоги до програмного забезпечення та безпосереднього проєктуванням архітектури програмного рішення.

Ретельне дослідження вимог та аналіз раніше підготованими даними і результатами ознайомлення з ідеєю, баченням та предметною областю дозволяють максимально точно наблизитись до бажаного кінцевого продукту в форматі заготовленої документації та даних готових до застосування в проєктуванні системи.

Проєкт цієї кваліфікаційної роботи – розробка веб-додатку із використанням мови програмування C# [1] та платформи ASP.NET Core [2] в сфері контролю фінансів, а точніше бюджетування та обліку з класифікацією. Вимоги що висуватимуться до системи є складовими від ідеї та предметної області, які раніше були проаналізовані та виокремлені для майбутнього застосування в кінцевому варіанті продукту. Тож, відповідно до поставленої задачі програмне забезпечення повинно ефективно та надійно виконувати на відповідати наступним функціональним вимогам:

1. Ведення обліку транзакцій по надходженням та витратам:

- Збір даних та класифікація за відповідними категоріями та типами транзакцій.

2. Ведення обліку активів (рахунки):

- Збір даних та класифікація за відповідними типами рахунків;
- Встановлення залежностей для оновлення даних активів залежно від транзакцій.

3. Ведення обліку бюджетів:

- Збір даних про бюджет у місячний період часу;
- Встановлення залежності для оновлення даних бюджету залежно від транзакцій.

4. Обов’язкову реєстрацію та вхід до облікового запису:

- Збір даних користувача для створення облікового запису;
- Встановлення залежності між усіма даними та користувацьким обліковим записом.

5. Збереження та маніпуляція даними у актуальному та прийнятному форматі;

6. Виконання точних та надійних обрахунків з грошовими одинцями.

Також, додаток повинен відповідати наступним нефункціональним вимогам:

1. Збереження та структура даних:

- Дані повинні бути чітко структуровані та важливі дані;
- Дані повинні бути актуальними та зберігатись довготривалий період часу;
- Сховище даних повинне бути надійним та ефективним, підтримувати потрібні типи даних;
- Система контролю бази даних повинна забезпечувати розробників зручним інструментарієм для роботи з базою даних;

2. Вимоги до програмного забезпечення:

- Платформа: Веб-браузер;
- Веб-фреймворк: ASP.NET Core [2];
- Мова програмування: C# 11.0 [1];

- Фреймворк Unit тестування: xUnit [5];
- ORM система: Entity Framework Core [6].

3. Вимоги до користувацького інтерфейсу:

- Відповідність стандартам якості;
- Мінімалістичність, комфортність та інтуїтивна легкість;
- Забезпечення доступу до визначених раніше функціональних вимог;

4. Архітектурні особливості:

- Модульність: Компоненти веб-додатку повинні бути визначені окремими модулями;
- Масштабованість: Компоненти веб-додатку повинні бути визначені з перспективою збільшення та модифікації;
- Відповідність принципам SOLID [7];
- Відповідність архітектурному шаблону MVC [8];
- Взаємодія з базою даних та передача даних.

5. Надійність та продуктивність:

- Час відгуку: не більше 2 секунд для стандартних операцій (завантаження сторінки, надсилання форми) та не більше 5 секунд для складний операцій (формування аналізу, діаграми, прогнозу);
- Пропускна здатність: мінімум 25 операцій в секунду під час пікового навантаження;
- Контроль версій: ведення локального та віддаленого репозиторію контроль версій;
- Обробка помилок: механізми обробки помилки, логування, визначення помилок для окремих компонентів.

6. Безпека:

- Аутентифікація: аутентифікації користувачів за допомогою безпечних та надійних систем на основі токенів, багатофакторної автентифікації.

В межах розділу викладені основні функціональні та нефункціональні вимоги до розроблюваного веб-додатку. Визначення чітких та детальних вимоги

дозволяє розбудувати міцний фундамент для етапі проєктування та розробки, уникаючи зайвих затрат.

1.5 Технології та інструменти розробки

В процесі виконання постановки задачі було заздалегідь підготовлено основу технологічного стеку, адже розробка комплексного веб-додатку потребує сучасні та надійні засоби та заходи.

Мова програмування C# – це C-подібна, платформонезалежна мова програмування орієнтована на імплементацію основних принципів об’єктно-орієнтованого програмування із автоматизованим менеджментом пам’яті та суворою типізацією [1].

Рисунок 1.7 – Логотип мови програмування

Мова програмування має велетенську спільноту та утримує провідні місця в списках найбільш використовуваних засобів, адже набір технологій, принципів та парадигм доступних до реалізації дозволяє виконувати найрізноманітніші задачі від створення програм та веб-застосунків до розробки ігор та розумних систем інтернету речей.

Однак, розробка веб-додатків не зовсім є основним призначенням лише мови програмування, тому для здійснення задуманого було обрано платформу

ASP.NET Core – високо-ефективний веб-фреймворк із засобами для створення та підтримки інтернет додатків [2].

Рисунок 1.8 – Логотип платформи

Платформа надає розробникам засоби для розробки та розгортання зручних та надійних веб-додатків, що можуть бути розроблені з інтерактивним користувацьким інтерфейсом, підтримкою прогресивних веб-технологій.

В якості бази даних в розробці вирішено використати MS SQL Server – сучасне та надійне програмне рішення, що дозволяє просто налаштувати сховище даних при використанні зручного інтерфейсу системи контролю бази даних [9].

Рисунок 1.9 – Логотип рушія бази даних

Для забезпечення зручності та контрольованості в процесі розробки вирішено використати GIT – розподілену систему контролю версій, яка широко використовується та глибоко інтегровна в процеси розробки програмного забезпечення [10]. Цей зручний інструмент дозволяє зробити розробку більш надійною та забезпечувати довготривалу підтримку проєктів.

Рисунок 1.10 – Логотип системи контролю версій

Таким чином, збільшиться надійність та ефектність розробки, адже система контролю версій дозволятиме легше вносити нові функції, модифікувати наявні, переписувати модулі та компоненти коду уникаючи можливих проблем з уже функціональним додатком.

Отож, сформований перелік технологій є мінімальною основою для розробки проєкту та дозволить швидко, ефективно та надійно розбудувати архітектуру, наповнити її функціоналом та представити базову версію веб-додатку.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ПРОГРАМНОГО РІШЕННЯ

2.1 Архітектура програмного рішення

Розробка передового програмного забезпечення, у сучасних реаліях, справа багатоетапна та ресурсозатратна. Подекуди не так важливий процес написання коду, а те які принципи та підходи лежать у його основі, адже це напряму впливають на продуктивність і модифікованість системи.

Архітектура програмного забезпечення – комплекс оптимальних рішень з побудови структури, ієрархії та комунікації компонентів з метою ефективного рішення певною проблематики. Так, для розробки програмного забезпечення було прийнято рішення застосувати архітектурний шаблон проєктування Model-View-Controller [8] у тісному поєднанні з трирівневою архітектурою [11]. Поєднання цих підходів дозволить спростити процеси та покращити кінцевий продукт.

Model-View-Controller – архітектурний шаблон проєктування, що найчастіше застосовується у програмних системах з користувацьким інтерфейсом та потребою у взаємодії з даними від користувача та сховища даних, із чітким поділом компонентів додатку на три категорії: моделі, представлення та контролери [8]. Ілюстративне зображення основних компонентів та їх взаємозв’язку наведено на рисунку 2.1.

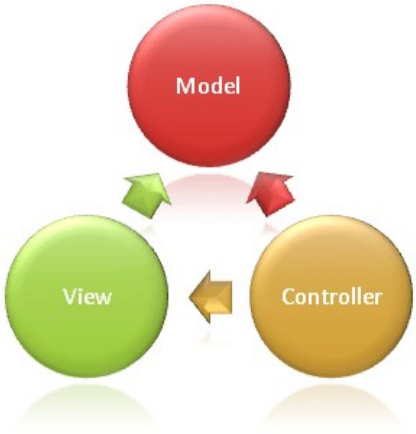

Рисунок 2.1 – Архітектурний шаблон Model-View-Controller [8]

Кожна з груп має на меті ізолювати власні процеси та дані від іншої групи, аби уникнути небажаних нагромаджень коду, неефективної логіки та небезпечних взаємодій між даними та логікою. Тобто, кожен компонент конкретної групи має своє призначення та відповідальності:

- Моделі: компоненти групи відповідальні лише за дані, їх стан та бізнес-логіку зав'язану на них. Таким чином, компоненти замикають у собі логіку та забезпечують надійність та цілісність даних [8];
- Подання: компоненти групи відповідальні лише за дані представлені безпосередньо користувачеві із застосуванням мінімуму логіки лише за потреби та без зміни даних [8];
- Контролери: компоненти групи відповідальні за встановлення комунікації між моделями та поданнями, створюючи ланцюжок в котрому контролер визначає як та яку модель застосовувати для конкретного представлення [8].

Тришарова архітектура – це архітектурний підхід для проєктування програмного забезпечення з поділом на три фундаментальних шари: шар доступу до даних, шар бізнес-логіки, шар інтерфейсу. Архітектурний підхід має на меті створити абстракції основних компонентних груп для забезпечення модульності та абстрактності [11]. Ілюстративне зображення тришарової архітектури наведено на рисунку 2.2.

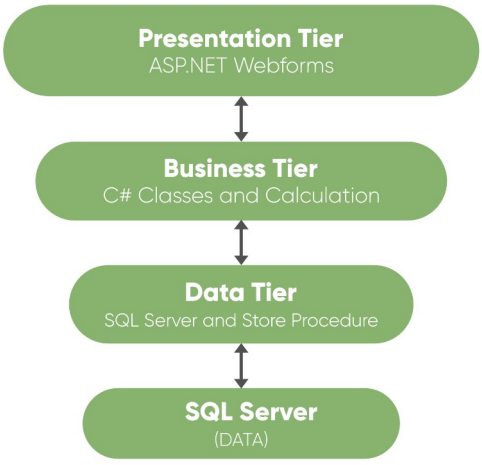

Рисунок 2.2 – Тришаровий архітектурний підхід

Архітектурний підхід дозволяє, подібно до раніше згаданого архітектурного шаблону, виділити і абстрагувати компоненти додатку, чим підвищити надійність та масштабованість коду. Тобто, програмне рішення стає поєднанням трьох модулі, або ж проєктів:

- Шар доступу до даних – найнижчий шар, що призначається для доступу і взаємодії бази даних із усією програмою. Шар містить засоби ти заходи виконання операцій бази даних [11];
- Шар бізнес-логіки – проміжний шар, що призначається для отримання та верифікації запитів на дані та самих даних. Шар містить засоби та заходи для проведення потрібних перевірок, обробок і таке інше [11];
- Шар інтерфейсу – шар безпосередньої взаємодії з користувач, що поєднує дані від ланцюжка попередніх шарів з користувацьким інтерфейсом і демонструє підсумок роботи програми за певним запитом та очікує на нові запити [11].

Отож, обраний архітектурний шаблон та підхід в межах розробки цього програмного рішення дозволять забезпечити надійну та ефективну структуру додатку. Застосування подібних архітектурних засобів є типовою практикою, дозволяючи зручно та ефективно поділити усе рішення та його компонент користувацького інтерфейсу. Також, це дає змогу уникнути неефективного, ненадійного та занадто нагромадженого коду, чим покращать його підтримку, модифікування, швидкодію та надійність.

2.2 Проєктування сутностей системи та бази даних

Дані відіграють ключову роль у сучасному світі, вони слугують основою, на якій створюють нові розробки, будуються стратегії та приймаються рішення. Безумовно, дані є рушієм сучасного світу, а отож і рушієм сфери розробки

програмного забезпечення. В інженерії програмного забезпечення дані визначають саму суть розроблюваного програмного рішення, те які задачі воно виконує, які результати дозволяє отримати та як воно безпосередньо функціонує. Тобто, те якими сутностями оперуватиме програмне рішення визначає можливі сфери та варіанти його застосування.

Фінансовий веб-додаток, що розробляється для дослідження в цій кваліфікаційній роботі, не є винятком та потребує дуже ретельно ставлення до даних на усіх етапах їх взаємодії з додатком, від отримання до збереження. Сутності повинні бути наповнені достатньою кількістю базової інформації в потрібному та актуальному форматі з відповідністю до всіх якісних характеристик програмної сутності та сутності бази даних.

Процеси проєктування є складними та дуже важливими, тому для уникнення помилок проєктування, можливої мішанини з невідповідністю даних програми та бази даних вирішено використати передові підходи до проєктування. Спершу, аби візуалізувати отримані дані буде створено ряд UML діаграм, що виконуватимуть роль шаблону для майбутньої реалізації. Наступним кроком, буде застосовано інструментарій Entity Framework Core [6], аби забезпечити ідентичність збережених даних в сховищі даних та моделей, котрі оперуватимуться в розроблюваному рішенні, використовуватимуться міграції та підхід Code First. Це методика доступна із використанням цього інструмент, завдячуючи їй процес проєктування може бути проведений в програмній реалізації та в подальшому перенесений з допомогою міграції у базу даних. Такий підхід дозволяє уникати помилок при перенесенні сутностей між складовими системи та можливість в подальшому гнучко та швидко модифікувати сховище даних та програму одномоментно.

Отож, маючи відповідні відомості з попередньо пройдених етапів та загального бачення проєктованої системи можна припустити наступні сутності для використання та збереження:

- Користувач;

- Транзакція;
- Тип транзакції;
- Категорія транзакції.
- Рахунок;
- Тип рахунку;
- Бюджет.

Приступимо до більш ретельного та проєктування обґрунтувавши їх використання та потребу з представленням в форматі діаграми.

Користувач (User) – сутність, основна мета якої забезпечення функцій реєстрації, входу та ідентифікації належності даних до конкретної особи. Користувач повинен містити унікальний ідентифікатор та помічні данні для оперування програмою. Діаграма сутності представлена на рисунку 2.3.

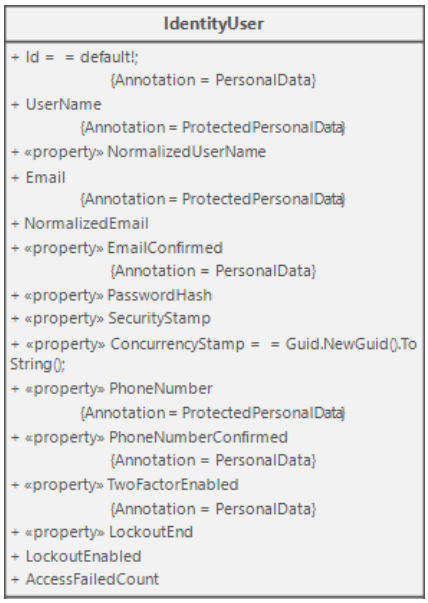

Рисунок 2.3 – Діаграма користувацької сутності

Транзакція (Transaction) – сутність, основна мета якої забезпечення обліку, типізації та категоризації фінансових потоків. Транзакція повинна містити інформації про суму витрати, тип, дату, категорію, рахунок і помічні дані. Діаграма сутності представлена на рисунку 2.4.

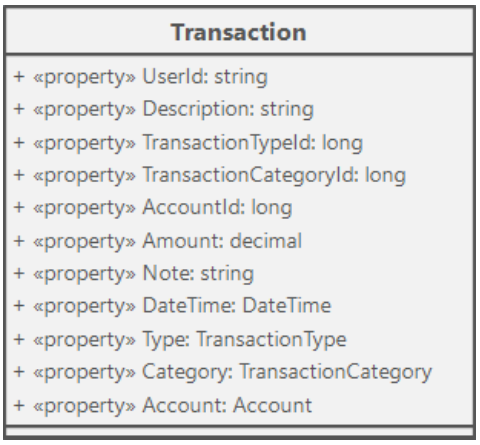

Рисунок 2.4 – Діаграма сутності транзакції

Тип транзакції (TransactionType) – сутність, основна мета якої забезпечення транзакції інформацією про тип для зручного поділу. Тип транзакції повинна містити інформацію про назву та поміжні дані. Діаграма сутності представлена на рисунку 2.5.

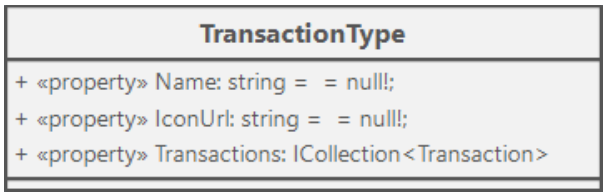

Рисунок 2.5 – Діаграма сутності типу транзакції

Категорія транзакції (TransactionCategory) – сутність, основна мета якої забезпечення транзакції інформацією про категорію для зручного поділу. Категорія транзакції повинна містити інформацію про назву та поміжні дані. Діаграма сутності представлена на рисунку 2.6.

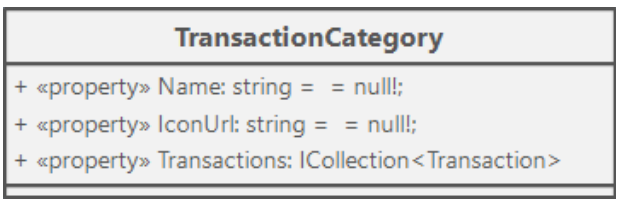

Рисунок 2.6 – Діаграма сутності категорії транзакції

Рахунок (Account) – сутність, основна мета якої забезпечення інформації про активи. Рахунок повинен містити інформацію про баланс, тип та назву. Діаграма сутності представлена на рисунку 2.7.

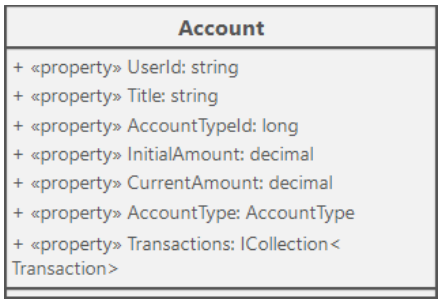

Рисунок 2.7 – Діаграма сутності рахунок

Тип рахунку (AccountType) – сутність, основна мета якої забезпечення інформації про тип рахунку. Тип рахунку повинен містити інформацію про назву та поміжну інформацію. Діаграма сутності представлена на рисунку 2.8.

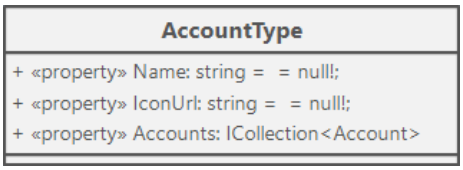

Рисунок 2.8 – Діаграма сутності тип рахунку

Бюджет (Budget) – сутність, основна мета якої забезпечення інформації про бюджет, заплановані витрати та надходження. Бюджет повинен містити інформацію про період бюджету, очікувані та реальні витрати й надходження. Діаграма сутності представлена на рисунку 2.9.

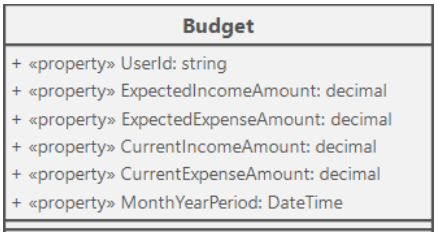

Рисунок 2.9 – Діаграма сутності бюджет

Діаграми класів усіх сутностей забезпечує достатнє розуміння бажаної структури збереження та оперування даними. Таким чином, створені діаграми в комбінації з інструментами фреймворку були налаштовані та застосовані до бази даних. В кінцевому результаті отримано діаграму сутностей, що була програмно реалізована та засобами інструменту міграцій застосована до бази даних, наведену на рисунку 2.10.

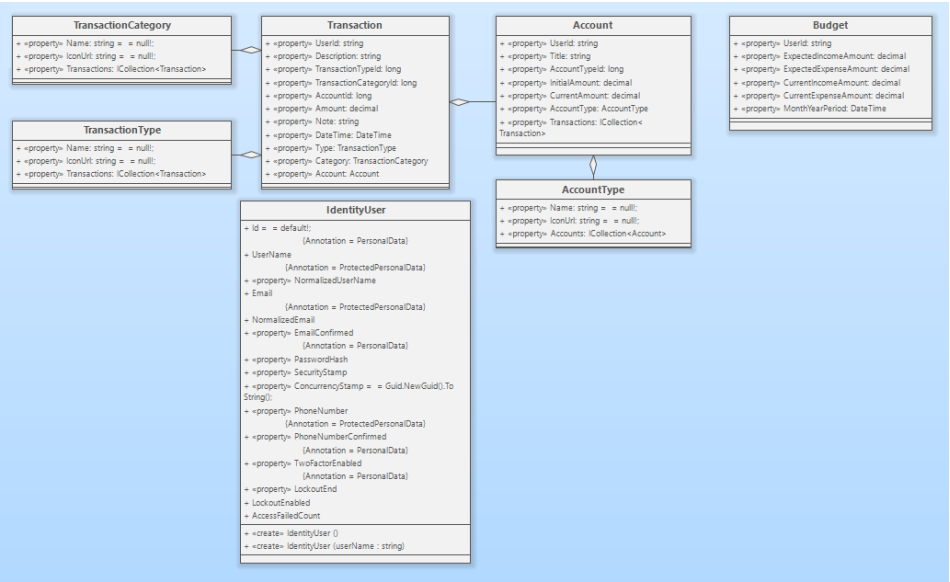

Рисунок 2.10 – Загальна діаграма класів усіх сутностей

Отже, відповідно з діаграмою система міститиме повний спектр мінімально необхідної кількості даних для бюджетування та обліку зі зручними варіаціями класифікації та типізації.

2.3 Розробка користувацького інтерфейсу

Розробка веб-додатку є завданням не тривіальним та вимагає умінь та навичок у багатьох сферах від створення макетів користувацького інтерфейсу до функціонального наповнення інтерфейсу бізнес-логікою. Тим не менш

використання інструментів обраної платформи та мови програмування дозволяють швидко та якісно розробити комфортний додаток із інтерфейсом, що міститиме в собі логіку та функціонал достатній для повноцінного використання.

Отож, веб-інтерфейс повинен задовільняти увесь спектр передбачених в етапі аналізу функціональних можливостей. Фактично усі ці функції компонуються у діаграму варіантів використання представлену на рисунку 2.11.

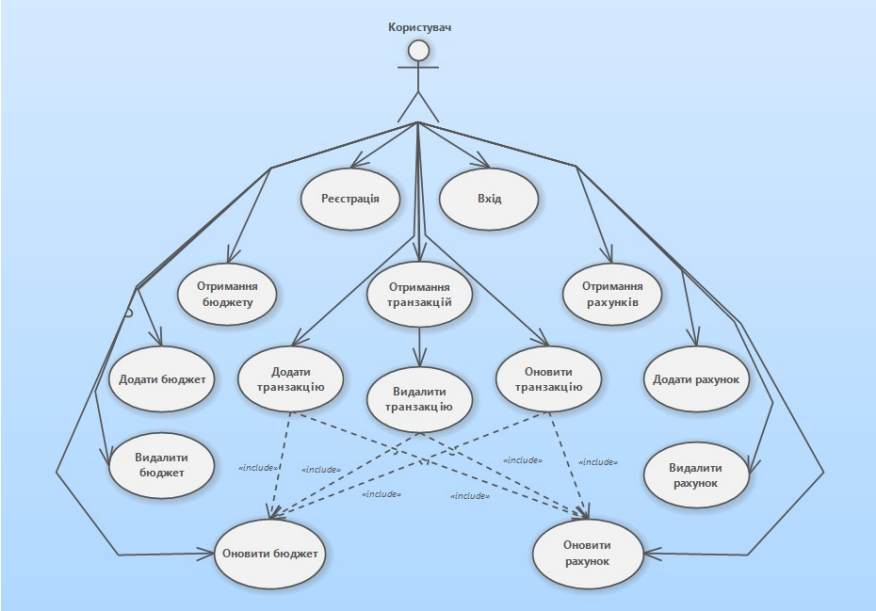

Рисунок 2.11 – Діаграма варіантів використання

Як можна побачити із діаграми варіантів використання, система містить мінімум 14 шляхів застосування програмного рішення, що умовно можуть бути поділені на 5 групи:

- Варіанти для входу в обліковий запис;
- Варіанти для створення облікового запису;
- Варіанти для використання транзакцій;
- Варіанти для використання рахунків;
- Варіанти для використання бюджетів.

Таким чином, для комфортного використання додатку можна скомпонувати кожну з груп у відповідні сторінки, тобто користувацький інтерфейс повинен налічувати щонайменше 5 сторінок:

- сторінка реєстрації(рис. 2.12) – основний спосіб створення облікового запису із даними про користувача;

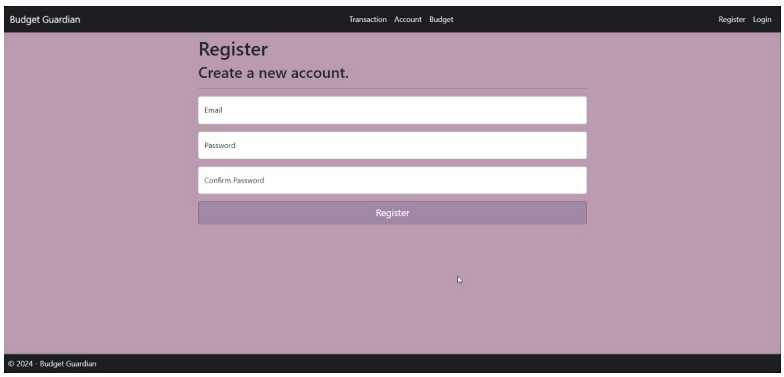

Рисунок 2.12 – Базовий дизайн сторінки реєстрації

- сторінка входу(рис. 2.13) – основний спосіб увійти та отримати доступ до облікового запису та усіх функцій веб-додатку;

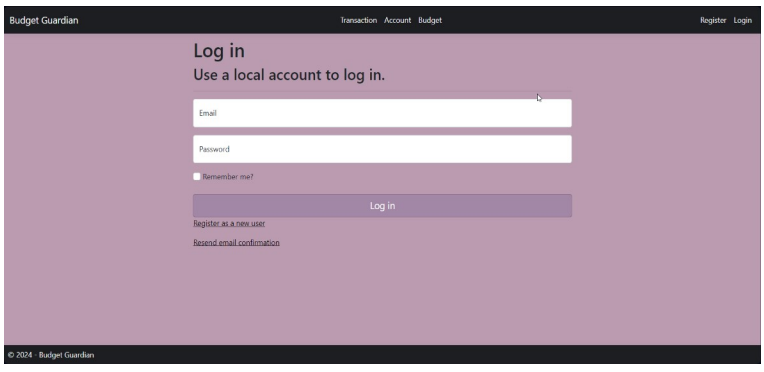

Рисунок 2.13 – Базовий дизайн сторінки входу

- сторінка транзакцій(рис. 2.14) – основний спосіб перегляду усіх транзакцій користувача та місце для переспрямування до вікон створення, редагування та видалення транзакцій;

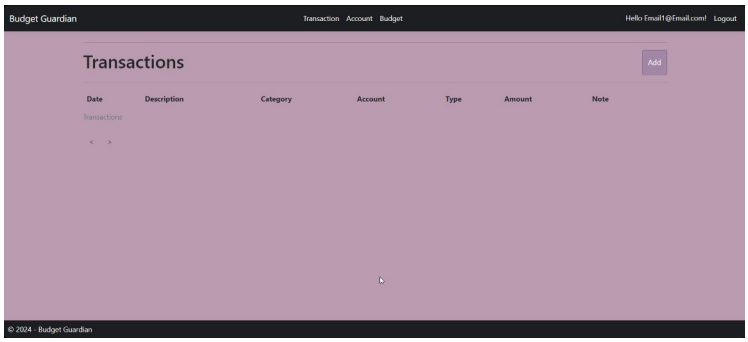

Рисунок 2.14 – Базовий дизайн сторінки транзакцій

• сторінка рахунків(рис. 2.15) – основний спосіб перегляду усіх рахунків користувача та місце для переспрямування до вікон створення, редагування та видалення рахунків;

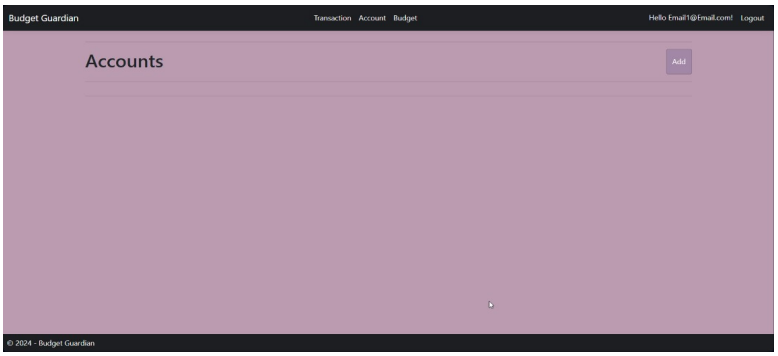

Рисунок 2.15 – Базовий дизайн сторінки рахунків

• сторінка бюджету(рис. 2.16) – основний спосіб перегляду бюджету за певний період часу та місце створення або редагування бюджету..

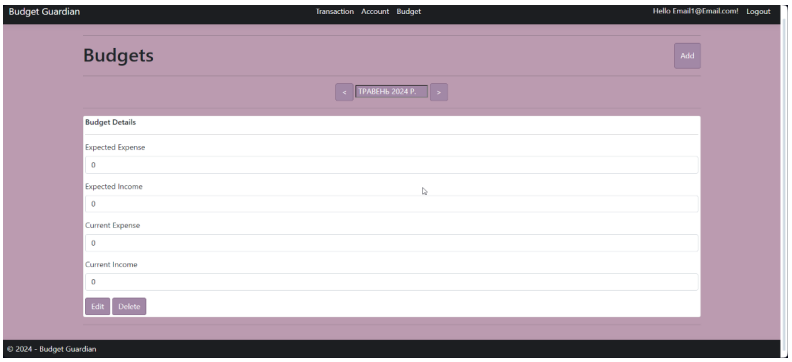

Рисунок 2.16 – Базовий дизайн сторінки бюджету

2.4 Розробка логіки системи

Розробка проєкту на даному етапі полягає в реалізації достатньої бізнес-логіки програмного рішення для забезпечення користувацького інтерфейсу функціональними можливостями з комунікацією із засобами верифікації та отримання й збереження даних. Тобто, завдання полягає в реалізації компонентів безпосередньо відповідальних за передачу даних між шарами та передача їх до кінцевих точок кожного варіанту використання.

Функціонування такого типу додатку, як згадувалось раніше, це оперування даними, великими об’ємами даних. Тому, першим ділом реалізовується шар доступу до даних частину якого уже було імплементовано, однак вони потребують додаткової обробки. Для аби досягнути бажаних результатів в проєкті використано комбінацію двох надійних та якісних шаблонів роботи з даними – Repository [12] та Unit Of Work [12].

Repository – потужний та продуктивний шаблон роботи з даними, що дозволяє виділити увесь спектр операцій з даними в окремий клас, де відповідно встановленому типу даних відбуватиметься доступ до цих самих даних [12]. Тобто, операції над сутностями транзакцій, типів й категорій транзакцій, рахунків, типів рахунків, бюджетів буде виокремлено у відповідні репозиторії. Для досягнення хороших принципів написання коду базові операції винесено в інтерфейс зі застосуванням загальних типів, що є хорошим тоном в програмуванні та підтримці коду.

Таким чином, інтерфейс дозволяє створити один уніфікований варіант та уникнути повторень, що збільшують ресурсозатратність та знижують надійність. За потреби, методи зі специфічними потребами та операціями можуть бути винесені в більш конкретний інтерфейс репозиторію, чим покращуючи чистоту та структурованість коду. Структура коду базового інтерфейсу репозиторію наведена на рисунку 2.17.

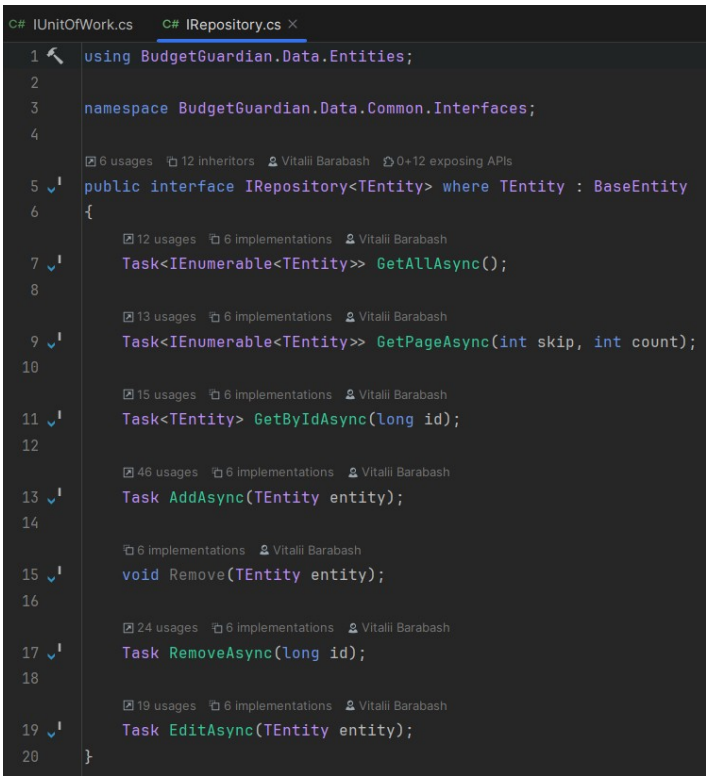

Рисунок 2.17 – Код базового інтерфейсу репозиторію

Unit Of Work – шаблон проєктування, який вибудовує архітектуру для підтримки цілісності та узгодженості даних в межах роботи із базою даних [12]. Також, шаблон здебільшого використовується за підтримки репозиторіїв, аби уникнути поганої структури та порушення принципів проєктування та розробки. Тобто, при створенні об’єкту цього класу йому надається весь спектр репозиторіїв додатку та встановлюється контроль над операцією збереження чи відміни внесених змін у якості окремого методу об’єкту.

Таким чином, даний шаблон є фактично реалізованою симуляцією транзакції бази даних, адже певним чином дозволяє досягнути аналогічних транзакції якостям та властивостям. Аналогічно до раніше описаних репозиторіїв, код цієї структури також виконано у відповідності до усіх принципів написання чистого і якісного коду, що у поєднанні з репозиторіями збільшує якісні характеристики коду. Структура коду інтерфейсу з репозиторіями та потрібними методами наведено на рисунку 2.18.

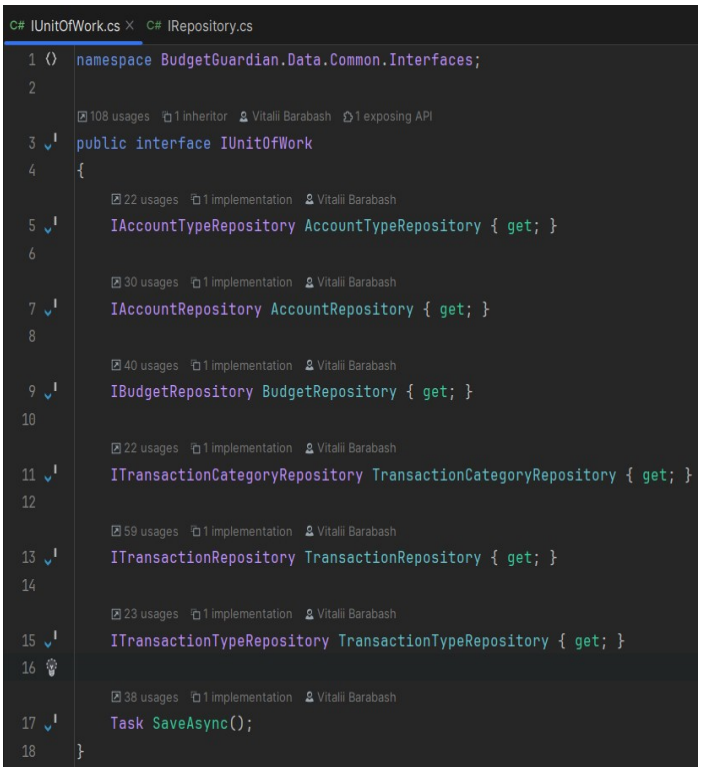

Рисунок 2.18 – Код базового інтерфейсу одиниці роботи

Реалізація усіх компонентів шару доступу до даних дозволяє приступати до подальшої розробки уже шару бізнес-логіки програмного рішення. Шар є проміжною ланкою між явним представленням даних на інтерфейсі та транзитом даних до, або від, сховища даних. Тобто, очікувана складність реалізації шару не є надто високою, адже виконувані операції не є складними й комплексними та заздалегідь відомі. Очікується, що розроблювані сервіси згрупують усі репозиторії в три групи:

- транзакції;
- рахунки;
- бюджети.

Таке групування обране через зручність подібного структурування, адже кілька репозиторіїв можуть бути поєднані в єдиний сервіс. Тобто, операції що стосуються конкретних видів даних з їх помічними будуть виконуватись лише в чітко визначеному сервісі, що є хорошою практикою в написані сервісів для додатку з великою кількістю сутностей.

Таким чином, додаток отримує не лише виділену логіку, що враховує усі важливі аспекти операцій над даними з їх актуалізацією та забезпеченням цілісності, а й зручний спосіб валідації даних на сервері для забезпечення додаткової надійності у виконанні операцій. На рисунку 2.19 наведено структура проєкту.

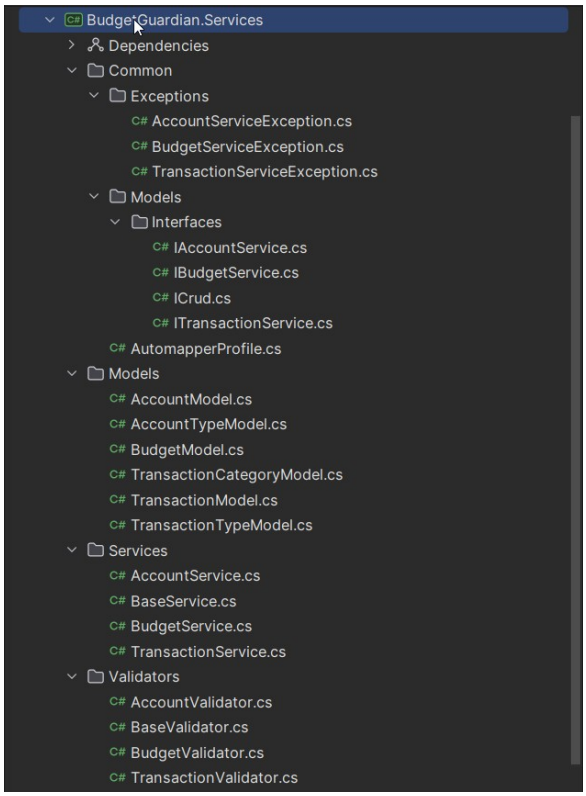

Рисунок 2.19 – Структура проєкту бізнес-логіки

Як видно з рисунку 2.19, проєкту виділяє власні моделі даних для зручного маніпулювання відповідними даними в сервісах, що згруповані відповідно до обраної структури проєкту бізнес-логіки. Також, шар виділяє відповідно до кількості сервісів, валідатори. Валідатори написані для забезпечення перевірки даних на відповідність критеріям серверної частини додатку і виконують перевірку вхідних даних з поверненням виняткових ситуацій при виникненні невідповідності. Для зручної обробки винятків на шарі користувацького інтерфейсу сервіси додають три види виняткових ситуацій, що відповідають кожному з сервісів та в подальшому дозволяють обробляти та прослідковувати

помилки в запитах простіше та ефективніше. Знову ж, як і у всьому проєкті використовуються інтерфейси для виокремлення логіки та забезпечення чистої та якісної архітектури.

Налагоджений шар доступу до даних та транзитний шар бізнес-логіки й створені макети користувацького інтерфейсу залишають потребу лише в реалізації моделей даних для взаємодії веб-сторінки з даними сервера та контролерів, що оброблятимуть ці дані. Моделі даних, у випадку шару користувацького інтерфейсу, створені для кожної з операцій задля чіткого розмежування даних і зручного присвоєння даних із тіла запиту засобами платформи. На рисунку 2.20 наведено перелік розроблених моделей даних.

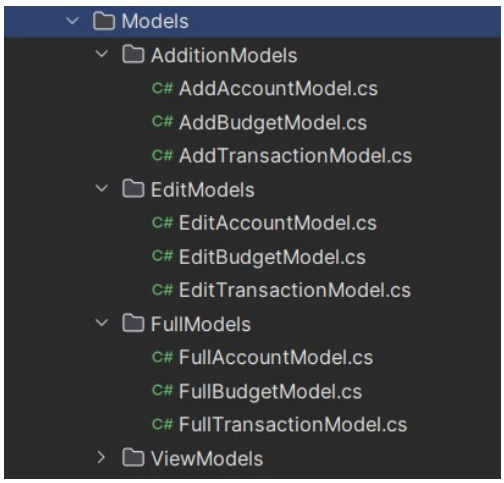

Рисунок 2.20 – Моделі даних для різних запитів

Створені моделі даних забезпечують надійне отримання даних у зручному форматі від запиту веб-клієнту. Тобто, із відповідними налаштуваннями платформа може отримати дані із запиту, записавши їх у структуру даних, котру далі використовуватиме контролер в цілях подальшої обробки та виконання потрібної логіки.

Контролери – це комунікаційна ланка між користувацьким інтерфейсом та бізнес-логікою програмного рішення, котра забезпечує обробку запитів веб-клієнту з створенням об’єктів наповнених даними запиту через механізми

прив’язки даних із застосування механізмів обраної технології, приклад наведено на рисунку 2.21.

Рисунок 2.21 – Прив’язка даних до моделі даних

Також, контролери налаштовано для виконання верифікації стану авторизації користувача, аби унеможливлюючи використання функціоналу в неавторизованому стані, приклад застосування наведено на рисунку 2.22.

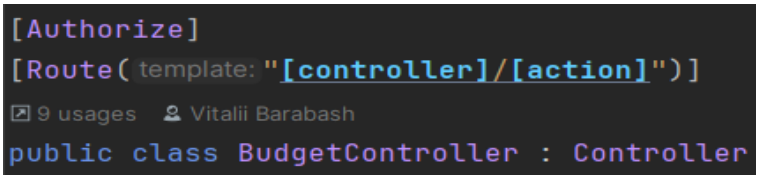

Рисунок 2.22 – Встановлення конфігурації верифікації стану авторизації

В добавок до цих налаштувань, логіка контролерів побудована таким чином, щоб забезпечувати використання раніше імплементованих сервісів для забезпечення роботи із даними. А також, кожен з методів для виконує обробку стану валідності моделі та обробку отриманих виняткових ситуацій, аби представити користувачеві детальний звіт про помилку відповідно до якісних норм обробки винятків з подальшим поверненням відповідного статус коду.

РОЗДІЛ 3. ТЕСТУВАННЯ ПРОГРАМНОГО РІШЕННЯ

3.1 Стратегія тестування

Проєкт розробки програмного забезпечення є процесом неперервним та подекуди нескінченним у часі, адже хороший та якісний проєкт буде підтримуватись та покращуватись протягом усього життєвого циклу програмного продукту. Тестування є невід'ємною складовою цих процесів, що має на меті забезпечення якості та надійності розроблюваного програмного продукту, із залученням перевірених часом підходів, методологій та інструментів.

Стратегія тестування – це є синергічним поєднанням принципів, загальноприйнятих підходів та технологічних інструментів для забезпечення якомога якіснішого та надійнішого способу перевірки коректного функціонування програмного продукту. Правильно обрана стратегія тестування дозволить забезпечити кращий процес розробки і зменшити витрати на подальше утримання та масштабування програмного продукту, нівелюючи ризики виникнення різної складності проблем, від тривіальних неточностей до критичних помилок.

Також, варто зазначити що, тестування веб-додатків не є чимось незвіданим та інноваційним, адже напрацювання в цій сфері уже давно довели свою ефективність, надаючи дуже круті рішення для проведення найрізноманітніших тестувань. Існують різні фреймворки, інструменти для відстеження та імітування запитів, підходи до створення тестів та уже готові стратегії тестування.

Отож, стратегія тестування, розроблюваного в цій кваліфікаційній роботі програмного продукту, буде намагатись застосовувати найкращі загальноприйняті практики, найпродуктивніші інструменти і таке інше.

Головною ідеєю стратегії тестування буде виконання поступової перевірки кожного з шарів розробленого додатку уникаючи повторень уже пройдених тестових варіантів та збільшуючи кількість тестованих компонентів для перевірки їх взаємодій. Тобто, головна ідея стратегії буде заключатись в реалізації, так званої

піраміди тестування – загальноприйнятого підходу, що дозволяє дуже ефективно виконати повноцінне тестування усього продукту з оптимальними затратами ресурсів [13]. Піраміда тестування виділяє такі основні етапи тестування:

- Unit тестування – найнижча ланка, що перевіряє роботу окремо взятого модуля в ізоляції від інших, лише його здатність виконати безпосередньо поставлену задачу;
- Інтеграційне тестування – середня ланка, що перевіряє здатність модулів комунікувати, виконувати завдання відповідно чітко визначеній послідовності кроків;
- Наскрізне тестування – найвища ланка, що перевіряє увесь додаток на виконання його функцій з усіма модулями, компонентами та іншими складовими.

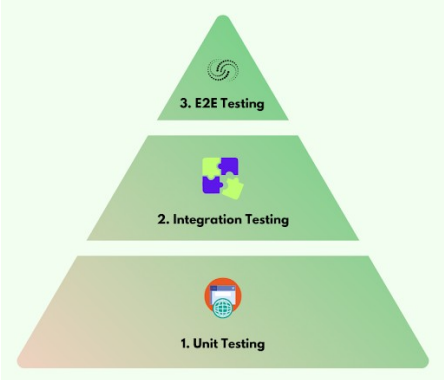

Рисунок 3.1 – Етапи тестування за обраною стратегією

Отож, першочергово тестування програмного рішення включатиме перевірку усіх функціональних можливостей шару доступу до даних. Тобто, перевірятимуться кожен репозиторій на здатність виконувати поставлені задачі з поверненням лише коректних результатів. Тестування виконуватиметься з використанням фреймворку xUnit [14] з базовим підходом до тестування модулів та їх методів.

Наступним етапом тестування, виконуватиметься перевірка комунікації шару доступу до даних із шаром бізнес-логіки. Мета цього етапу виконати

перевірку логіки взаємодії та здатності додатку працювати із переведенням даних до транзитних моделей даних. Етап планується до виконання з використанням фреймворку xUnit [14] та Moq [15] бібліотеки.

Завершальним етапом тестування є виконання тестування у форматі використання програмного рішення з намаганням використати усі наявні функціональні можливості для виявлення помилок, неточностей і таке інше. Етап проводиться вручну для ретельнішої перевірки.

3.2 Розробка та застосування тестування

Тестування шару доступу до даних, як уже було зазначено в заздалегідь розробленій стратегії, виконується з використанням одного з сучасних та ефективних фреймворків. Проєкт рішення для тестування створюється з встановленням необхідних залежностей, що спрощує підготовку та впровадження тестування. Загалом, тестування кожного з репозиторіїв створюється відповідно до укладених в процесі розробки методів. На рисунку 3.2 наведено структуру одного з тестових класів для демонстрації побудови тестів.

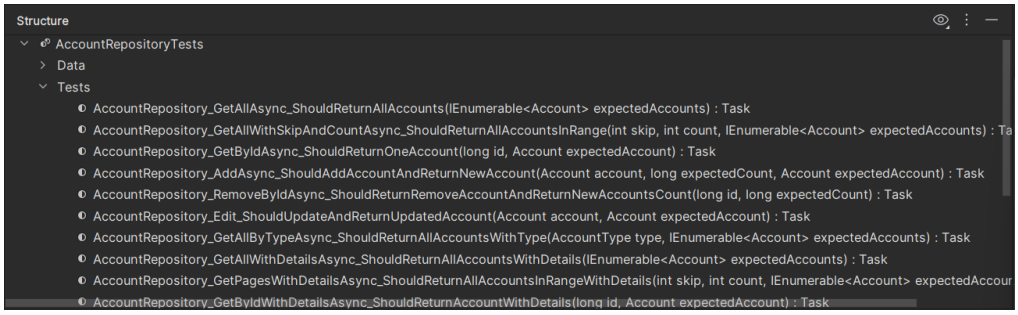

Рисунок 3.2 – Методи тестування репозиторію рахунків

Кожен з методів розробляється з урахуванням відповідності загальноприйнятим якісним практикам тестування. По-перше, сховище даних

наповнюється даними аби симулювати наявність присутності користувацьких даних. Далі, виконуються операції з репозиторієм для отримання чи надсилання даних. В кінці, виконується перевірка на відповідність отриманих результатів до очікуваних, для чого було розроблено з використанням інтерфейсу порівняння класи, що виконують перевірку двох об’єктів визначеного типу. На рисунку 3.3 наведено приклад виконання цієї послідовності в коді.

Рисунок 3.3 – Код тестування одного з методів репозиторію рахунків

Такий шаблон тестового методу в подальшому застосований до кожного методу репозиторію та усіх репозиторіїв, без виключення. Таким чином, результати проведеного тестування, що наведені на рисунку 3.4, свідчитимуть про рівень коректності побудованої логіки взаємодії з сховищем даних.

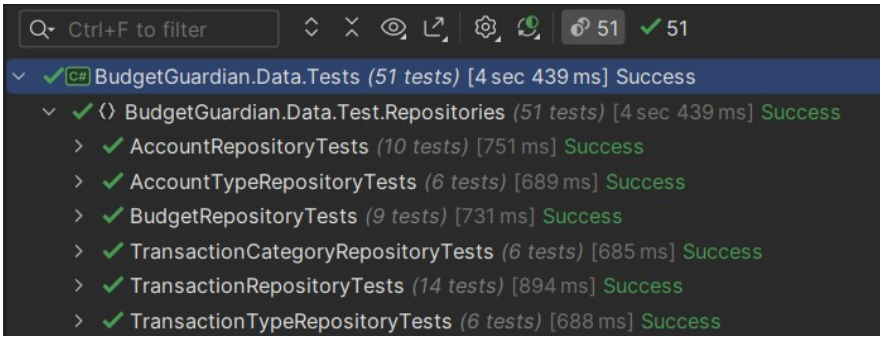

Рисунок 3.4 – Результати тестування репозиторіїв шару доступу до даних

Як видно з рисунку 3.4, тестування репозиторіїв завершилось успіхом, кожен з методів повернув очікуваний результат. Таким чином, шар доступу до даних є придатним до використання і виконує заплановані завдання.

Наступним кроком, виконано тестування шару бізнес-логіки, що є доволі не тривіальною задачею, адже виконує не повинне виконати не банальну перевірку повернутих результатів. Головна мета цього тестування перевірити реалізований код на відповідність встановленій послідовності кроків та коректно виконує конвертацію даних до моделей даних сервісів.

Такий функціонал не є звичним для фреймворку тестування, тому використовуватиметься стороння бібліотека, що дозволятиме широко налаштовувані підробки об’єктів з можливість налаштування методів, їх результатів, перевірки виконання та викликів налаштованих об’єктів.

```
1375    [Theory]
1376    [MemberData(nameof(GetByIdTransactionAsyncTestCases))]
        ◇  Vitalii Barabash
1377    public static async Task TransactionService_GetByIdAsync_ShouldReturnTransaction(
1378        Transaction entity,
1379        long modelId,
1380        TransactionModel expectedModel)
1381    {
1382        // Arrange
1383        Mapper mapper = MapperConfiguringTool.GetConfiguredMapper();
1384        Mock<IUnitOfWork> unitOfWorkMock = new();
1385        unitOfWorkMock.Setup(expression: unitOfWork => unitOfWork.TransactionRepository.GetByIdWithDetailsAsync(It.IsAny<long>()))
1386            .ReturnsAsync(entity);
1387        TransactionService service = new(unitOfWorkMock.Object, mapper);
1388
1389        // Act
1390        TransactionModel actualModel = await service.GetByIdAsync(modelId);
1391
1392        // Assert
1393        unitOfWorkMock.Verify(expression: unitOfWork => unitOfWork.TransactionRepository.GetByIdWithDetailsAsync(
1394                It.Is<long>(match: l :long => l == modelId)),
1395            Times.Exactly(1));
1396
1397        Assert.Multiple(params checks: () =>
1398        {
1399            Assert.NotNull(actualModel);
1400            Assert.Equal(expectedModel, actualModel, comparer: new TransactionComparator());
1401        });
1402    }
```

Рисунок 3.5 – Реалізація тестування сервісу транзакцій шару бізнес-логіки

Як видно з рисунку 3.5, бібліотека Moq [15] надає функціонал для створення об’єктів, що імітують справжній екземпляри об’єктів створених в межах програмного рішення. Тобто, програмним кодом реалізоване налаштування об’єкт доступу до репозиторіїв з подальшою встановленням методу та його

результатів. Таким чином, об’єкт зімітує виклик методу і поверне дані, а в подальшому буде проведено перевірку на відповідність результату наданим даним і на послідовність та кількість викликів методів.

Отож, таким чином буде виконано ретельну перевірку кожного сервісу на виконання ланцюжка викликів і повернення бажаного результату, будь то об’єкт чи виняток. Результати перевірки наведено на рисунку 3.6.

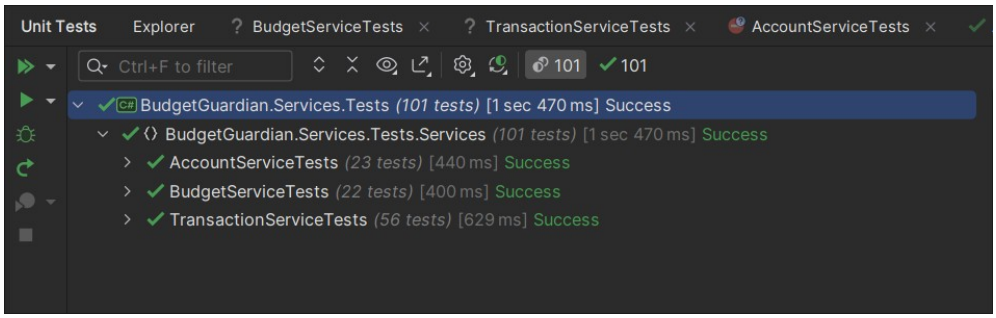

Рисунок 3.6 – Результати тестування шару бізнес-логіки

Отже, шар бізнес-логіки повернув позитивний результат, тим самим підтвердивши коректне виконання логіки сервісів з верифікаціями та викликами потрібних методів. Тож, сервіси виконують поставлену задача, а у комбінації з раніше проведеними тестами шару бізнес-логіки можна зазначити наявність абсолютно функціональної системи, що може отримувати дані від користувача та надсилати дані користувачеві.

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Менеджмент безпеки життєдіяльності

Безпека здоров’я, життя та діяльності людини останні кількасот років є основоположним фактором усіх виробничих процесів у системі “людина-машина-середовище”. Уміння ефективно та раціонально застосовувати передові надбання безпекових наук та освоювати загальнолюдський досвід в цілях досягнення цієї безпеки є менеджментом безпеки життєдіяльності, одним з найважливіших аспектів налагодження виробничих процесів кваліфікованими управлінцями [16].

Основні компоненти менеджменту безпеки життєдіяльності

1. Аналіз ризиків і оцінка небезпек

- Виявлення потенційних загроз.
- Оцінка ймовірності та можливих наслідків цих загроз.
- Розробка стратегії мінімізації ризиків.

2. Розробка політики та процедур безпеки

- Створення внутрішніх документів, що регулюють безпеку на всіх рівнях організації.
- Визначення відповідальних осіб за безпеку.

3. Навчання та підвищення кваліфікації персоналу

- Регулярні тренінги для працівників з питань безпеки.
- Підвищення обізнаності щодо потенційних небезпек та методів їх уникнення.

4. Контроль та аудит

- Постійний моніторинг виконання заходів безпеки.
- Проведення внутрішніх та зовнішніх аудитів для оцінки ефективності системи безпеки.

5. Управління інцидентами

- Розробка планів реагування на надзвичайні ситуації.

- Система оповіщення та дій у разі виникнення інцидентів.
- Аналіз інцидентів та впровадження коригуючих заходів.

Таким чином, менеджер безпеки є працівником, який забезпечує послідовні процеси з виявлення, аналізу та уникнення небезпечних для життя та здоров'я працівників ситуацій застосовуючи увесь наявний ресурс.

В контексті України, створення сприятливої безпекової ситуація у системі "людина-машина-навколишнє середовище" гарантується Конституцією України і включає такі закони України(див. рис. 4.2):

- з охорони праці;
- з охорони навколишнього середовища;
- з дорожнього руху;
- з цивільного захисту;
- з охорони здоров'я [17].

Рисунок 4.2 – Законодавча база з безпеки життєдіяльності

Також, важливими аспектами менеджменту безпеки в контексті України є загальноприйняті в світі норми та специфікації, на яких базуються системи менеджменту охорони праці і промислової безпеки (СМОПіПБ):

- ISO 45001:2018 "Системи управління охороною здоров'я та безпекою праці";
- ISO 31000:2018 "Управління ризиками".

Впровадження СМОПіПБ дає можливості звести до мінімуму аварійні та небезпечні ситуації і зменшити ризики виробництва, раціонально використовувати кошти і підвищити ефективність та якість праці.

Отож, менеджмент безпеки є важливою стратегічною сферою розвитку людства, яка дозволяє вибудувати ефективне та надійне конкурентоспроможне виробництво з мінімальними ризикам та затратами.

4.2 Вимоги ергономіки до організації робочого місця ПК

Ергономіка – це галузь знань, яка комплексно вивчає систему “людина-машина-середовище” з метою забезпечення умов праці, які б максимально відповідали фізичним, психологічним та соціальним можливостям людини, роблячи роботу безпечною та комфортною [18].

Основні завдання ергономіки в сучасних реаліях полягають в:

- дослідження системи “людина-машина-навколишнє середовище”, основних закономірностей взаємодії та співіснування людини-оператора та машини;
- розробка та впровадження норм до проектування робочої зони в межах системи “людина-машина-навколишнє середовище” в найефективнішому та найнадійнішому вигляді;
- розробка та впровадження основних правил взаємодії людини-оператора та машини;
- розробка та впровадження норм до інструментарію органів управління робочого ПК та його програмного забезпечення.

Система “людина-машина-середовище” завжди є поєднанням трьох елементів: предмет праці, засоби праці та суб’єкти праці. Найбільшу зацікавленість вчення ергономіки проявляє до робочого місця та його коректного

облаштування, аби забезпечити найкращі та найбезпечніші умови праці для людини-оператора.

Робоче місце – це зона спеціального призначення, що обладнана та укомплектована різною номенклатурою засобів, де виконується конкретний вид роботи робітником або групою робітників в цілях досягнення поставлених виробничих цілей [18].

Тим самим, робоче місце це основна робоча зона для людини-оператора ПК, тому вимоги повинна відповідати усім ергономічно якісним вимогам, які в контексті України встановлюються законодавчо у формати державних стандартів України. Тож, вимоги до робочого місця людини-оператора встановлюються двома основними стандартами:

- ДСТУ 8604:2015 "Дизайн і ергономіка. Робоче місце для виконання робіт у положенні сидячи. Загальні ергономічні вимоги";
- ДСТУ ISO 9241-5 "Ергономічні вимоги до робочих місць з відеотерміналами".

Відповідно до цих нормативно-правових актів робоче місце людини-оператора ПК для ефективної та комфортної роботи повинне відповідати якісним стандартам вмебльованості та устаткованості.

Вимоги до столу:

- висота столу має регулюватися або відповідати зросту працівника, щоб його передпліччя вільно лежали на столі під кутом 90°, а зап'ястя були прямими (див. рис. 4.2);
- розмір столу повинен бути підлаштований до потреб працівника, стандарти розміру: висота – 725 мм, ширина – 600–1400 мм, глибина – 800–1000 мм [18];

Вимоги до стільця:

- висота сидіння стільця має регулюватися або відповідати зросту працівника, щоб його стопи твердо стояли на підлозі, а коліна були зігнуті під кутом 90° (див. рис. 4.2);

- стілець має мати регульовану спинку, яка забезпечує підтримку поперекового відділу хребта (див. рис. 4.2);
- глибина сидіння стільця має бути достатньою, щоб спина працівника щільно прилягала до спинки стільця (див. рис. 4.2);
- підлокітники стільця мають бути регульованими по висоті та відстані від сидіння.

Вимоги до монітору:

- монітор має бути розташований на відстані 500-750 мм від очей працівника (див. рис. 4.2);
- центр екрану монітора має знаходитися на рівні очей або трохи нижче.
- монітор має бути нахилений під кутом 10-20° назад (див. рис. 4.2);
- яскравість та контрастність монітора мають бути відрегульовані так, щоб зображення було чітким і комфортним для сприйняття.

Вимоги до клавіатура:

- клавіатура має бути розташована на столі таким чином, щоб зап'ястя працівника були прямими, а передпліччя – вільно лежали на столі (див. рис. 4.2);
- клавіатура має мати ухил 5-15° назад;
- клавіші клавіатури мають бути чітко позначені та мати оптимальний розмір і форму.

Вимоги до миші:

- миша має бути розташована на столі таким чином, щоб зап'ястя працівника було прямим, а передпліччя - вільно лежали на столі;
- розмір і форма миші мають відповідати розміру та формі руки працівника.

Вимоги до мікроклімату:

- температура в приміщенні має бути в межах 20-24°С;
- відносна вологість повітря має бути в межах 40-60%;
- швидкість повітря не повинна перевищувати 0,2 м/с;

• освітлення робочого місця має бути достатнім та рівномірним. Рівень освітлення на робочій поверхні столу має бути в межах 300-500 лк.

Вимоги до приміщення:

• на робочому місці має бути достатньо вільного простору для рухів працівника;

• робоче місце має бути чистим і охайним;

• рівень шуму на робочому місці не повинен перевищувати 40 дБ;

• працівник повинен мати можливість робити перерви в роботі протягом дня;

• при розміщенні декількох робочих місць в одному приміщенні площа робочого місця складає не менше 6 кв. М, а об'м – не менше 15 куб.м [18].

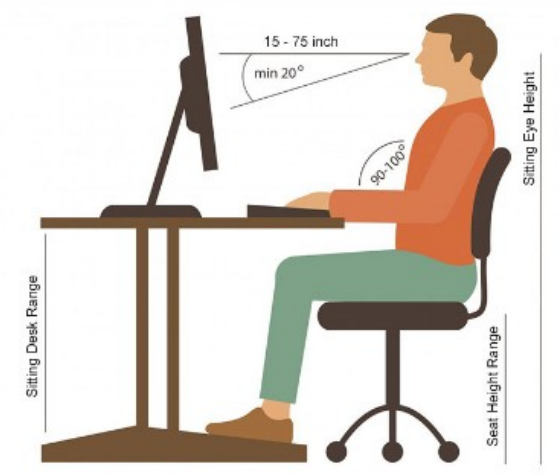

Рисунок 4.2 – Ілюстративне зображення ергономічного робочого місця

Отож, чинні норми та стандарти по облаштуванню робочого місця встановлюють чіткі характеристики облаштування робочого місця людини-оператора ПК для підтримки високих характеристик якості та ефективності роботи.

ВИСНОВКИ

В результаті виконання цієї кваліфікаційної роботи було представлено програмне рішення веб-додатку для оптимізації ведення бюджету із залученням мови програмування C# [1]. Розроблений програмний продукт диджиталізує процеси обліку та планування фінансової активності в мінімалістичному та інтуїтивному веб-додатку, чим суттєво збільшує ефективність та надійність таких задач на фоні традиційних методів бюджетування.

Програмне рішення веб-додатку розроблялось згідно поставленої задачі з залученням до розробки аналітичних процесів дослідження предметної області, перевірка ринку наявних рішень, формування вимог та визначення технологічного стеку. Такий підхід дозволив підготувати важливі для розробки проєкту дані, які можуть бути затребувані в ході проєктування та розробки. Таким чином, вдалось уникнути помилок проєктування, небажаних компонентів та багато інших ресурсоємних затрат.

Основою веб-додатку, згідно обраного стеку технологій, стали мова програмування C# [1] та платформа ASP.NET Core [2]. Потужна зв’язка технологій, що обрана через здатність забезпечити обширним спектром інструментів для розробки багатосторінкового веб-додатку із залученням бази даних. Інструментарій стеку має багато застосування, однак в цій роботі найбільш вагомими стали:

- наявність зручної та ефективної підтримки комунікацій між веб-клієнтом та додатком дозволяючи реалізувати запланований задум;
- розгортання бази даних із залученням об’єктно-реляційної розмітки від Entity Framework Core[6] в цілях уникнення помилок;
- облаштування асинхронного середовища для оптимізації операцій шляхом паралелізації процесів;

- застосування швидких та ефективних послідовностей різних даних для збереження великих об’ємів різної інформації.

Варто також зазначити, що вагому роль в становленні якісного програмного продукту відіграла обрана стратегія побудови архітектури програмного рішення. Трирівнева архітектура із залученням Model-View-Controller [8] та Unit Of Work [12] і Repository шаблонів дозволила створити якісне та масштабоване програмне рішення забезпечивши чудове підґрунтя для вибудовування ефективної комунікації між модулями з уникненням безглуздих дублювань коду, нагромаджень, зайвих залежностей і таке інше.

Також, важливим аспектом в підборі архітектурних шаблонів та проєктуванні програмного рішення відіграли процеси побудови візуальних представлень компонентів системи, тобто UML діаграм. Діаграми дозволили виокремити та структурувати окремі компоненти відзначивши їх основні характеристики і шляхи взаємодії з іншими частинами системи. Тим самим, дозволивши сформувати явне представлення програмного рішення без коду і надалі спростити етапи розробки логіки та користувацького інтерфейсу програмного рішення.

В цілях забезпечення явного підтвердження достатнього рівня функціональних можливостей та відповідності поставленій задачі та вимогам програмне рішення протестовано засобами фреймворку xUnit [5]. Тестування проведене таким способом дозволило верифікувати здатність додатку встановлювати пряму комунікацію з базою даних отримуючи та передаючи дані, й виконати перевірку логіки виконання таких операцій.

Також, було висвітлено основні поняття із менеджменту безпеки життєдіяльності та основ охорони праці, адже в контексті сучасного світу виробничі процеси зв’язані з безпосередніми ризиками. Тому, розуміння основних загроз та правил поводження з ними і знання основ організації зручного комфортного робочого місця дають змогу убезпечити робочі процеси.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A tour of the C# language [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-US/dotnet/csharp/tour-of-csharp/.

2. Overview of ASP.NET Core [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0.

3. Saldo Apps [Електронний ресурс] – Режим доступу до ресурсу: https://saldoapps.com/.

4. Saldo Finance [Електронний ресурс] – Режим доступу до ресурсу: https://saldofinanceapp.com/.

5. About xUnit.net [Електронний ресурс] – Режим доступу до ресурсу: https://xunit.net/#documentation.

6. Entity Framework Core [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/ef/core/.

7. Принципи SOLID. Навіщо вони потрібні програмістам та як з ними працювати [Електронний ресурс] – Режим доступу до ресурсу: https://training.epam.ua/ua/blog/602.

8. Overview of ASP.NET Core MVC [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-8.0.

9. SQL Server: матеріали для завантаження | Microsoft [Електронний ресурс] – Режим доступу до ресурсу: https://www.microsoft.com/uk-ua/sql-server/sql-server-downloads.

10. Git [Електронний ресурс] – Режим доступу до ресурсу: https://git-scm.com/.

11. What is .NET 3-Tier Architecture? [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/what-is-net-3-tier-architecture/.

12. Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10) [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application.

13. What is Testing Pyramid? How Does It Benefit Agile Teams? [Електронний ресурс] – Режим доступу до ресурсу: https://testsigma.com/blog/testing-pyramid/.

14. About xUnit.net [Електронний ресурс] – Режим доступу до ресурсу: https://xunit.net/.

15. devlooped/moq: The most popular and friendly mocking framework for .NET [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devlooped/moq.

16. Безпека життєдіяльності, основи охорони праці – Лекція 7 [Електронний ресурс] — Режим доступу до ресурсу: https://dl.tntu.edu.ua/content.php?cid=349742

17. Стиценко Т.Є., Пронюк Г.В., Сердюк Н.М., Хондак І.І. «Безпека життєдіяльності» навчальний посібник / Т. Є.Стиценко, Г. В. Пронюк, Н. М. Сердюк, І. І. Хондак. – Харків: ХНУРЕ, 2018. – 16 – 96 с.

18. Стиценко Т.Є., Пронюк Г.В., Сердюк Н.М., Хондак І.І. «Безпека життєдіяльності» навчальний посібник / Т. Є.Стиценко, Г. В. Пронюк, Н. М. Сердюк, І. І. Хондак. – Харків: ХНУРЕ, 2018. – 115 – 123 с.

ДОДАТКИ

Лістинг 1 – код BudgetController

```
using System.Diagnostics;
using System.Security.Claims;

using BudgetGuardian.Presentation.Models.AdditionModels;
using BudgetGuardian.Presentation.Models.EditModels;
using BudgetGuardian.Presentation.Models.FullModels;
using BudgetGuardian.Presentation.Models.ViewModels;
using BudgetGuardian.Services.Common.Models.Interfaces;
using BudgetGuardian.Services.Models;

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BudgetGuardian.Presentation.Controllers;

[Authorize]
[Route("[controller]/[action]")]
public class BudgetController : Controller
{
    private readonly IBudgetService _service;
    private readonly ILogger<BudgetController> _logger;

    public BudgetController(IBudgetService service,
ILogger<BudgetController> logger)
    {
        _service = service;
        _logger = logger;
    }

    [HttpGet]
    public async Task<IActionResult> Index(string? current, string?
move)
    {
        int year = DateTime.Now.Year;
        int month = DateTime.Now.Month;
        FullBudgetModel budgetModel = new FullBudgetModel();
        BudgetPageInfo pageInfo;

        string? userId = null;

        try
        {
            userId =
HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (string.IsNullOrEmpty(userId))
```

```
            {
                throw new InvalidOperationException("User was not
found.");
            }

            year = DateTime.Now.Year;
            month = DateTime.Now.Month;

            if (!string.IsNullOrEmpty(move) && !
string.IsNullOrEmpty(current))
            {
                year = int.Parse(current.Split('-')[0]);
                month = int.Parse(current.Split('-')[1]);

                switch (move)
                {
                    case "next":
                        if (month + 1 == 13)
                        {
                            month = 1;
                            year += 1;
                        }
                        else
                        {
                            month += 1;
                        }

                        break;
                    case "previous":
                        if (month - 1 == 0)
                        {
                            month = 12;
                            year -= 1;
                        }
                        else
                        {
                            month -= 1;
                        }

                        break;
                }
            }

            BudgetModel budget = (await
_service.GetByYearMonthAsync(year, month)).First(model =>
model.UserId == userId);
            budgetModel = new FullBudgetModel
            {
                Id = budget.Id,
                UserId = budget.UserId,
                ExpectedIncomeAmount = budget.ExpectedIncomeAmount,
                ExpectedExpenseAmount =
```

```
budget.ExpectedExpenseAmount,
                CurrentIncomeAmount = budget.CurrentIncomeAmount,
                CurrentExpenseAmount = budget.CurrentExpenseAmount,
                MonthYearPeriod = budget.MonthYearPeriod
            };
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(nameof(Index),
                new BudgetViewModel
                {
                    PageInfo = new BudgetPageInfo
                    {
                        Budget = new FullBudgetModel
                        {
                            Id = 0,
                            UserId = userId ?? string.Empty,
                            ExpectedIncomeAmount = 0,
                            ExpectedExpenseAmount = 0,
                            CurrentIncomeAmount = 0,
                            CurrentExpenseAmount = 0,
                            MonthYearPeriod = default
                        },
                        MonthAndYear = DateTime.Now
                    }
                });
        }
        finally
        {

            pageInfo = new BudgetPageInfo
            {
                Budget = budgetModel,
                MonthAndYear = new DateTime(year, month, 01, 00, 00,
00, DateTimeKind.Local)
            };
        }

        return View(new BudgetViewModel { PageInfo = pageInfo });
    }

    [HttpPost]
    public async Task<IActionResult> Add(

[Bind("ExpectedExpenseAmount,ExpectedIncomeAmount,MonthAndYearPeriod
", Prefix = "AddModel")] AddBudgetModel model)
    {
        try
        {
            string? userId =
```

```
HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (string.IsNullOrEmpty(userId))
            {
                throw new InvalidOperationException("User was not
found.");
            }

            if (ModelState.IsValid)
            {
                BudgetModel budgetModel = new BudgetModel
                {
                    UserId = userId,
                    ExpectedIncomeAmount =
model.ExpectedIncomeAmount,
                    ExpectedExpenseAmount =
model.ExpectedExpenseAmount,
                    CurrentIncomeAmount = 0,
                    CurrentExpenseAmount = 0,
                    MonthYearPeriod = model.MonthAndYearPeriod
                };
                await _service.AddAsync(budgetModel);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(nameof(Index),
                new BudgetViewModel
                {
                    PageInfo = new BudgetPageInfo
                    {
                        Budget = new FullBudgetModel
                        {
                            Id = 0,
                            UserId = string.Empty,
                            ExpectedIncomeAmount = 0,
                            ExpectedExpenseAmount = 0,
                            CurrentIncomeAmount = 0,
                            CurrentExpenseAmount = 0,
                            MonthYearPeriod = default
                        },
                        MonthAndYear = DateTime.Now
                    }
                });
        }

        return RedirectToAction(nameof(Index));
    }
```

```
    [HttpPost]
    public async Task<IActionResult> Edit(

[Bind("Id,UserId,ExpectedIncomeAmount,ExpectedExpenseAmount,CurrentI
ncomeAmount,CurrentExpenseModel,MonthAndYearPeriod", Prefix =
"EditModel")]EditBudgetModel model)
    {
        try
        {
            if (ModelState.IsValid)
            {
                BudgetModel budgetModel = new BudgetModel
                {
                    Id = model.Id,
                    UserId = model.UserId,
                    ExpectedIncomeAmount =
model.ExpectedIncomeAmount,
                    ExpectedExpenseAmount =
model.ExpectedExpenseAmount,
                    CurrentIncomeAmount = model.CurrentIncomeAmount,
                    CurrentExpenseAmount =
model.CurrentExpenseAmount,
                    MonthYearPeriod = model.MonthAndYearPeriod
                };
                await _service.UpdateAsync(budgetModel);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(nameof(Index),
                new BudgetViewModel
                {
                    PageInfo = new BudgetPageInfo
                    {
                        Budget = new FullBudgetModel
                        {
                            Id = 0,
                            UserId = string.Empty,
                            ExpectedIncomeAmount = 0,
                            ExpectedExpenseAmount = 0,
                            CurrentIncomeAmount = 0,
                            CurrentExpenseAmount = 0,
                            MonthYearPeriod = default
                        },
                        MonthAndYear = DateTime.Now
                    }
                });
        }
```

```
        return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public async Task<IActionResult> Delete([Bind("modelId", Prefix
= "EditModel")]long modelId)
    {
        try
        {
            if (ModelState.IsValid)
            {
                await _service.DeleteAsync(modelId);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(nameof(Index),
                new BudgetViewModel
                {
                    PageInfo = new BudgetPageInfo
                    {
                        Budget = new FullBudgetModel
                        {
                            Id = 0,
                            ExpectedIncomeAmount = 0,
                            ExpectedExpenseAmount = 0,
                            CurrentIncomeAmount = 0,
                            CurrentExpenseAmount = 0,
                            MonthYearPeriod = default
                        },
                        MonthAndYear = DateTime.Now
                    }
                });
        }

        return RedirectToAction(nameof(Index));
    }

    [ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
    [HttpGet]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

Лістинг 2 – код MoneyAccountController

```
using System.Diagnostics;
using System.Globalization;
using System.Security.Claims;

using BudgetGuardian.Presentation.Models.AdditionModels;
using BudgetGuardian.Presentation.Models.EditModels;
using BudgetGuardian.Presentation.Models.FullModels;
using BudgetGuardian.Presentation.Models.ViewModels;
using BudgetGuardian.Services.Common.Models.Interfaces;
using BudgetGuardian.Services.Models;

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BudgetGuardian.Presentation.Controllers;

[Authorize]
[Route("[controller]/[action]")]
public class MoneyAccountController : Controller
{
    private readonly IAccountService _service;

    private readonly ILogger<MoneyAccountController> _logger;

    public MoneyAccountController(IAccountService service,
ILogger<MoneyAccountController> logger)
    {
        _service = service;
        _logger = logger;
    }

    [HttpGet]
    public async Task<IActionResult> Index()
    {
        AccountPageInfo pageInfo;
        string? userId = null;

        try
        {
            userId =
HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (string.IsNullOrEmpty(userId))
            {
                throw new InvalidOperationException("User was not
found.");
            }

            Task<IEnumerable<AccountTypeModel>> accountTypeModels =
_service.GetAllTypesAsync();
            IDictionary<long, string> accountTypeNameById = new
```

```
Dictionary<long, string>();
            IDictionary<string, IEnumerable<FullAccountModel>>
accountsByType =
                new Dictionary<string,
IEnumerable<FullAccountModel>>();

            foreach (AccountTypeModel model in await
accountTypeModels)
            {
                accountTypeNameById.TryAdd(model.Id, model.Name);

                IEnumerable<FullAccountModel> accountModels = (await
_service.GetAllByTypeAsync(model)).Select(
                    accountModel => new FullAccountModel
                    {
                        Id = accountModel.Id,
                        UserId = accountModel.UserId,
                        Title = accountModel.Title,
                        AccountTypeId = accountModel.AccountTypeId,
                        AccountTypeName =
accountModel.AccountTypeName,
                        AccountIconUrl =
accountModel.AccountIconUrl,
                        InitialAmount = accountModel.InitialAmount,
                        CurrentAmount = accountModel.CurrentAmount
                    }).Where(accountModel => accountModel.UserId ==
userId);

                accountsByType.TryAdd(model.Name, accountModels);
            }

            pageInfo = new AccountPageInfo
            {
                AccountsByType = accountsByType,
                AccountTypeNameById = accountTypeNameById
            };
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(new AccountViewModel
            {
                PageInfo = new AccountPageInfo
                {
                    AccountsByType = new Dictionary<string,
IEnumerable<FullAccountModel>>(),
                    AccountTypeNameById = new Dictionary<long,
string>()
                }
            });
        }
```

```
        return View(new AccountViewModel{PageInfo = pageInfo});
    }

    [HttpPost]
    public async Task<IActionResult>
Add([Bind("Title,AccountTypeId,Amount", Prefix =
"AddModel")]AddAccountModel model)
    {
        string? userId = null;

        try
        {
            userId =
HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (string.IsNullOrEmpty(userId))
            {
                throw new InvalidOperationException("User was not
found.");
            }

            if (ModelState.IsValid)
            {
                AccountModel accountModel = new AccountModel
                {
                    Title = model.Title,
                    UserId = userId,
                    AccountTypeId = model.AccountTypeId,
                    InitialAmount = model.Amount,
                    CurrentAmount = model.Amount
                };
                await _service.AddAsync(accountModel);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException is not null ?
exception.InnerException.Message : exception.Message);
            return View(nameof(Index), new AccountViewModel
            {
                PageInfo = new AccountPageInfo
                {
                    AccountsByType = new Dictionary<string,
IEnumerable<FullAccountModel>>(),
                    AccountTypeNameById = new Dictionary<long,
string>()
                }
            });
        }
```

```
        return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public async Task<IActionResult>
Edit([Bind("Id,UserId,Title,AccountTypeId,InitialAmount,CurrentAmoun
t", Prefix = "EditModel")]EditAccountModel model)
    {
        try
        {
            if (ModelState.IsValid)
            {
                AccountModel accountModel = new AccountModel
                {
                    Id = model.Id,
                    UserId = model.UserId,
                    Title = model.Title,
                    AccountTypeId = model.AccountTypeId,
                    InitialAmount =
decimal.Parse(model.InitialAmount, NumberStyles.Number, new
CultureInfo("en-US")),
                    CurrentAmount =
decimal.Parse(model.CurrentAmount, NumberStyles.Number, new
CultureInfo("en-US"))
                };
                await _service.UpdateAsync(accountModel);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException is not null ?
exception.InnerException.Message : exception.Message);
            return View(nameof(Index), new AccountViewModel
            {
                PageInfo = new AccountPageInfo
                {
                    AccountsByType = new Dictionary<string,
IEnumerable<FullAccountModel>>(),
                    AccountTypeNameById = new Dictionary<long,
string>()
                }
            });
        }

        return RedirectToAction(nameof(Index));
    }

    [HttpPost]
```

```
    public async Task<IActionResult> Delete(long id)
    {
        try
        {
            if (ModelState.IsValid)
            {
                await _service.DeleteAsync(id);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException is not null ?
exception.InnerException.Message : exception.Message);
            return View(nameof(Index), new AccountViewModel
            {
                PageInfo = new AccountPageInfo
                {
                    AccountsByType = new Dictionary<string,
IEnumerable<FullAccountModel>>(),
                    AccountTypeNameById = new Dictionary<long,
string>()
                }
            });
        }

        return RedirectToAction(nameof(Index));
    }

    [ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
    [HttpGet]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

Лістинг 3 – код TransactionController

```
using System.Diagnostics;
using System.Globalization;
using System.Security.Claims;

using BudgetGuardian.Presentation.Models.AdditionModels;
using BudgetGuardian.Presentation.Models.EditModels;
using BudgetGuardian.Presentation.Models.FullModels;
using BudgetGuardian.Presentation.Models.ViewModels;
```

```
using BudgetGuardian.Services.Common.Models.Interfaces;
using BudgetGuardian.Services.Models;

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BudgetGuardian.Presentation.Controllers;

[Authorize]
[Route("[controller]/[action]")]
public class TransactionController : Controller
{
    private readonly ITransactionService _transactionService;
    private readonly IAccountService _accountService;
    private readonly ILogger<TransactionController> _logger;

    public TransactionController(ITransactionService
transactionService, IAccountService accountService,
ILogger<TransactionController> logger)
    {
        _transactionService = transactionService;
        _accountService = accountService;
        _logger = logger;
    }

    [HttpGet]
    public async Task<IActionResult> Index(int? pageNumber)
    {
        TransactionPageInfo pageInfo;
        string? userId = null;

        try
        {
            userId =
HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (string.IsNullOrEmpty(userId))
            {
                throw new InvalidOperationException("User was not
found.");
            }

            List<FullTransactionModel> models = (await
_transactionService.GetAllAsync()).Select(model =>
                    new FullTransactionModel
                    {
                        Id = model.Id,
                        UserId = model.UserId,
                        Description = model.Description,
                        TransactionTypeId = model.TransactionTypeId,
                        TransactionTypeTitle =
model.TransactionTypeTitle,
                        TransactionTypeIconUri =
```

```
model.TransactionTypeIconUri,
                        TransactionCategoryId =
model.TransactionCategoryId,
                        TransactionCategoryTitle =
model.TransactionCategoryTitle,
                        TransactionCategoryIconUri =
model.TransactionCategoryIconUri,
                        AccountId = model.AccountId,
                        AccountTitle = model.AccountTitle,
                        Amount = model.Amount,
                        Note = model.Note,
                        DateTime = model.DateTime
                    })
                .OrderByDescending(model => model.DateTime)
                .ToList();
            Task<IEnumerable<TransactionTypeModel>> transactionTypes
= _transactionService.GetAllTypesAsync();
            Task<IEnumerable<TransactionCategoryModel>>
transactionCategories = _transactionService.GetAllCategoriesAsync();
            Task<IEnumerable<AccountModel>> accounts =
_accountService.GetAllAsync();

            IDictionary<long, string> transactionTypesById = new
Dictionary<long, string>();
            foreach (TransactionTypeModel model in await
transactionTypes)
            {
                transactionTypesById.TryAdd(model.Id, model.Name);
            }

            IDictionary<long, string> transactionCategoriesById =
new Dictionary<long, string>();
            foreach (TransactionCategoryModel model in await
transactionCategories)
            {
                transactionCategoriesById.TryAdd(model.Id,
model.Name);
            }

            IDictionary<long, string?> accountById = new
Dictionary<long, string?>();
            foreach (AccountModel model in await accounts)
            {
                accountById.TryAdd(model.Id, model.Title);
            }

            pageInfo = new TransactionPageInfo
            {
                PaginatedList =
PaginatedList<FullTransactionModel>.CreateAsync(models,
pageNumber ?? 1, 10),
                TransactionTypesById = transactionTypesById,
```

```
                TransactionCategoriesById =
transactionCategoriesById,
                AccountsById = accountById
            };
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(new TransactionViewModel
            {
                PageInfo = new TransactionPageInfo
                {
                    PaginatedList =
PaginatedList<FullTransactionModel>.CreateAsync(new
List<FullTransactionModel>(), pageNumber ?? 1, 10),
                    TransactionTypesById = new Dictionary<long,
string>(),
                    TransactionCategoriesById = new Dictionary<long,
string>(),
                    AccountsById = new Dictionary<long, string?>()
                }
            });
        }

        return View(new TransactionViewModel { PageInfo =
pageInfo });
    }

    [HttpPost]
    public async Task<IActionResult>
Add([Bind("Description,TransactionTypeId,TransactionCategoryId,Accou
ntId,Amount,Note,DateTime", Prefix = "AddModel")]AddTransactionModel
model)
    {
        string? userId = null;

        try
        {
            userId =
HttpContext.User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (string.IsNullOrEmpty(userId))
            {
                throw new InvalidOperationException("User was not
found.");
            }

            if (ModelState.IsValid)
            {
                TransactionModel transactionModel = new
TransactionModel
                {
```

```
                    UserId = userId,
                    Description = model.Description,
                    TransactionTypeId = model.TransactionTypeId,
                    TransactionCategoryId =
model.TransactionCategoryId,
                    AccountId = model.AccountId,
                    Amount = model.Amount,
                    Note = model.Note,
                    DateTime = model.DateTime
                };
                await
_transactionService.AddAsync(transactionModel);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(nameof(Index), new TransactionViewModel
            {
                PageInfo = new TransactionPageInfo
                {
                    PaginatedList =
PaginatedList<FullTransactionModel>.CreateAsync(new
List<FullTransactionModel>(), 1, 10),
                    TransactionTypesById = new Dictionary<long,
string>(),
                    TransactionCategoriesById = new Dictionary<long,
string>(),
                    AccountsById = new Dictionary<long, string?>()
                }
            });
        }

        return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public async Task<IActionResult> Edit(

[Bind("Id,UserId,Description,TransactionTypeId,TransactionCategoryId
,AccountId,Note,DateTime", Prefix =
"EditModel")]EditTransactionModel model)
    {
        try
        {
            if (ModelState.IsValid)
            {
                TransactionModel transactionModel = new
TransactionModel
```

```
                {
                    Id = model.Id,
                    UserId = model.UserId,
                    Description = model.Description,
                    TransactionTypeId = model.TransactionTypeId,
                    TransactionCategoryId =
model.TransactionCategoryId,
                    AccountId = model.AccountId,
                    Amount = decimal.Parse(model.Amount,
NumberStyles.Number, new CultureInfo("en-US")),
                    Note = model.Note,
                    DateTime = model.DateTime
                };
                await
_transactionService.UpdateAsync(transactionModel);

                return RedirectToAction(nameof(Index));
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(nameof(Index), new TransactionViewModel
            {
                PageInfo = new TransactionPageInfo
                {
                    PaginatedList =
PaginatedList<FullTransactionModel>.CreateAsync(new
List<FullTransactionModel>(), 1, 10),
                    TransactionTypesById = new Dictionary<long,
string>(),
                    TransactionCategoriesById = new Dictionary<long,
string>(),
                    AccountsById = new Dictionary<long, string?>()
                }
            });
        }

        return RedirectToAction(nameof(Index));
    }

    [HttpPost]
    public async Task<IActionResult> Delete(long id)
    {
        try
        {
            if (ModelState.IsValid)
            {
                await _transactionService.DeleteAsync(id);

                return RedirectToAction(nameof(Index));
```

```
            }
        }
        catch (Exception exception)
        {
            ModelState.AddModelError(string.Empty,
exception.InnerException?.Message ?? string.Empty);
            return View(nameof(Index), new TransactionViewModel
            {
                PageInfo = new TransactionPageInfo
                {
                    PaginatedList =
PaginatedList<FullTransactionModel>.CreateAsync(new
List<FullTransactionModel>(), 1, 10),
                    TransactionTypesById = new Dictionary<long,
string>(),
                    TransactionCategoriesById = new Dictionary<long,
string>(),
                    AccountsById = new Dictionary<long, string?>()
                }
            });
        }

        return RedirectToAction(nameof(Index));
    }

    [ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
    [HttpGet]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

Лістинг 4 – Код конфігурації та запуску додатку

```
using BudgetGuardian.Data.Common.Interfaces;
using BudgetGuardian.Data.Infrastructure;
using BudgetGuardian.Data.Repositories;
using BudgetGuardian.Identity.Entities;
using BudgetGuardian.Identity.Infrastructure;
using BudgetGuardian.Services.Common;
using BudgetGuardian.Services.Common.Models.Interfaces;
using BudgetGuardian.Services.Services;

using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;

namespace BudgetGuardian.Presentation;
```

```
public static class Program
{
    public static void Main(string[] args)
    {
        WebApplicationBuilder builder =
WebApplication.CreateBuilder(args);

        // Get configuration data.
        string dataContextConnection =
builder.Configuration.GetConnectionString("DataConnection") ??
                                        throw new
InvalidOperationException("'DataConnection' was not found.");

        string identityContextConnection =
builder.Configuration.GetConnectionString("IdentityConnection") ??
                                          throw new
InvalidOperationException("'IdentityConnection' was not found.");

        // Add MVC services to the container.
        builder.Services.AddControllersWithViews();

        // Add razor pages to the container.
        builder.Services.AddRazorPages();

        // Differentiate development and release
        if (builder.Environment.IsDevelopment())
        {
            // Add data context services to the container.
            builder.Services.AddDbContext<BudgetGuardianDbContext>(
                options =>
options.UseInMemoryDatabase(dataContextConnection));

            // Add identity context services to the container.

builder.Services.AddDbContext<BudgetGuardianIdentityDbContext>(
                options =>
options.UseInMemoryDatabase(identityContextConnection));
        }
        else
        {
            // Add data context services to the container.
            builder.Services.AddDbContext<BudgetGuardianDbContext>(
                options =>
options.UseSqlServer(dataContextConnection));

            // Add identity context services to the container.

builder.Services.AddDbContext<BudgetGuardianIdentityDbContext>(
                options =>
options.UseSqlServer(identityContextConnection));
        }
```

```
        builder.Services.AddDefaultIdentity<User>(
                options => options.SignIn.RequireConfirmedAccount =
true)
            .AddEntityFrameworkStores<BudgetGuardianIdentityDbContex
t>();

        // Add data mapper service to the container.
        builder.Services.AddAutoMapper(configuration =>
configuration.AddProfile(new AutomapperProfile()));

        // Add data repositories to the container.
        builder.Services.AddScoped<IAccountRepository,
AccountRepository>();
        builder.Services.AddScoped<IAccountTypeRepository,
AccountTypeRepository>();
        builder.Services.AddScoped<IBudgetRepository,
BudgetRepository>();
        builder.Services.AddScoped<ITransactionCategoryRepository,
TransactionCategoryRepository>();
        builder.Services.AddScoped<ITransactionRepository,
TransactionRepository>();
        builder.Services.AddScoped<ITransactionTypeRepository,
TransactionTypeRepository>();

        // Add unit of work service to the container.
        builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();

        // Add data services to the container.
        builder.Services.AddScoped<IAccountService,
AccountService>();
        builder.Services.AddScoped<IBudgetService, BudgetService>();
        builder.Services.AddScoped<ITransactionService,
TransactionService>();

        // Add razor pages to the container.
        builder.Services.AddRazorPages();

        WebApplication app = builder.Build();

        // Configure the HTTP request pipeline.
        if (!app.Environment.IsDevelopment())
        {
            app.UseExceptionHandler("/Home/Error");
            // The default HSTS value is 30 days.
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();
```

```
            app.UseAuthorization();

            app.MapDefaultControllerRoute();

            app.MapControllerRoute(
                "default",
                "{controller=Home}/{action=Index}/{id?}");

            app.MapRazorPages();

            app.Run();
        }
    }
```

Додаток Б – Диск з роботою