ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка веб-застосунку обліку витрат для найманого домоуправителя та домогосподаря»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень.
Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Денис ПОЧАПСЬКИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-43
Денис ПОЧАПСЬКИЙ

Керівник: Тимур ДОВЖЕНКО
к. т.н.

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Почапському Денису Ігоровичу

1. Тема кваліфікаційної роботи: «Розробка веб-застосунку обліку витрат для найманого домоуправителя та домогосподаря»
керівник кваліфікаційної роботи к.т.н.Тимур ДОВЖЕНКО,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про клієнт-серверну архітектуру, опис методів організації обліку витрат у цифрових системах, документація з розробки веб-застосунків на базі Node.js та MySQL, опис бібліотек і технологій для реалізації автентифікації, збереження даних та візуалізації інтерфейсу користувача.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження та аналіз підходів до створення веб-застосунків для обліку витрат у сфері домашнього господарства.
2. Характеристика системи обліку витрат та методи її побудови.
3. Розроблення проєктних рішень та їх реалізація.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Архітектура системних компонентів.
5. Алгоритм роботи.
6. Екранні форми.
7. Апробація результатів дослідження.
8. Висновки.

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд та аналіз існуючих додатків обліку витрат для найманого домоуправителя та домогосподаря	14.03-22.03.2025	
4	Проектування застосунку для обліку витрат для найманого домоуправителя та домогосподаря	23.03-02.04.2025	
5	Програмна реалізація веб-застосунку	03.04-20.04.2025	
6	Тестування застосунку	21.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Денис ПОЧАПСЬКИЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Тимур ДОВЖЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 73 стор., 3 табл., 4 рис., 20 джерел.

Мета роботи – підвищення ефективності обліку витрат між найманим домоуправителем і господарем за рахунок автоматизації процесу створення, перегляду та контролю заявок на закупівлю.

Об'єкт дослідження – інформаційна взаємодія між найманим працівником та замовником у побутовому середовищі.

Предмет дослідження – програмне забезпечення для фіксації, контролю та обробки запитів на витрати в межах домогосподарства.

Короткий зміст роботи: У роботі досліджено проблему прозорого обліку витрат у побутовій взаємодії між господарем і найманим домоуправителем. Проведено аналіз існуючих рішень та обґрунтовано потребу у спеціалізованому веб-застосунку. Розроблено клієнт-серверну архітектуру на основі Node.js, MySQL і HTML/CSS/JS. Реалізовано дві ролі користувача — управитель і господар — з відповідним функціоналом: створення, погодження, відхилення й видалення заявок. Систему протестовано в типових сценаріях. Здійснено автоматизацію очищення форм, оновлення таблиць і логування дій.

Сфера використання застосунку – цифровий облік витрат у побутовому середовищі, приватних оселях та будинках з найманими працівниками.

КЛЮЧОВІ СЛОВА: ОБЛІК ВИТРАТ, ЗАПИТИ НА ЗАКУПІВЛЮ, NODE.JS, MYSQL, ДОМОУПРАВИТЕЛЬ, ГОСПОДАР, РОЛІ КОРИСТУВАЧІВ, WEB-ЗАСТОСУНОК

ЗМІСТ

ВСТУП

В умовах сучасної цифрової трансформації суспільства спостерігається стрімке впровадження інформаційних технологій у повсякденне життя громадян. Зростаюча кількість обов'язків, які виконують наймані працівники у сфері домашнього господарства, таких як домоуправителі та домогосподарі, зумовлює потребу в ефективних інструментах автоматизації їхньої діяльності. Облік витрат на закупівлю товарів і послуг є одним із ключових аспектів у роботі таких працівників, оскільки він безпосередньо впливає на прозорість, фінансову дисципліну та довіру між наймачем і виконавцем.

Актуальність роботи полягає в необхідності створення простого, ефективного та доступного веб-застосунку, який би дозволяв домоуправителям вести облік запитів на витрати, а замовникам — контролювати ці запити у зручному форматі. Існуючі програмні рішення переважно орієнтовані на корпоративний сегмент або мають надлишкову функціональність, що ускладнює їх використання у побутових умовах. Крім того, не всі такі застосунки забезпечують багаторольовий доступ, що важливо для відокремлення функцій між працівником і господарем. Таким чином, розробка спеціалізованого веб-застосунку, адаптованого до потреб управління витратами в контексті найманої праці у домогосподарстві, є актуальним і важливим напрямом прикладних досліджень.

Метою роботи є підвищення ефективності обліку витрат між найманим домоуправителем і господарем шляхом впровадження веб-застосунку з чітким функціональним розмежуванням ролей, зручним інтерфейсом, низькими витратами на обслуговування та адаптивністю до побутових сценаріїв.Для досягнення поставленої мети необхідно вирішити такі завдання дослідження:

проаналізувати існуючі інформаційні системи та сервіси з обліку витрат;

виокремити специфічні потреби та функціональні вимоги до системи для домоуправителя та господаря;

визначити архітектуру веб-застосунку та обґрунтувати вибір технологій для реалізації;

спроєктувати структуру бази даних, що забезпечить збереження, обробку та захист даних користувачів;

розробити функціональну модель системи з використанням сучасних принципів веб-розробки (MVC, REST, адаптивна верстка);

реалізувати багаторольову систему доступу та управління запитами;

протестувати застосунок у типових сценаріях використання;

оцінити теоретичну цінність, практичне значення й потенціал масштабування створеного застосунку.

Об’єкт дослідження — процеси інформаційного супроводу обліку витрат у контексті взаємодії між найманим працівником та власником домогосподарства.

Предмет дослідження — веб-застосунок, що реалізує функціональність керування запитами на витрати, з диференційованим доступом для домоуправителя та господаря.

Джерельну базу дослідження становлять наукові праці у галузі розробки інформаційних систем, документація щодо сучасних технологій веб-програмування (зокрема Node.js, MySQL, Express, HTML/CSS/JS), а також практичні кейси використання інформаційних застосунків для обліку побутових витрат.

Фактичним матеріалом дослідження слугує власноруч реалізований прототип веб-застосунку, що включає структуру бази даних, серверну логіку, маршрутизацію API, клієнтську частину для домоуправителя і господаря, а також інтерфейс взаємодії користувача з системою. Проєкт реалізовано з використанням стеку Node.js + MySQL + HTML/CSS без використання сторонніх UI-фреймворків, з метою забезпечення максимальної кастомізації інтерфейсу під потреби користувачів.

У процесі розробки були застосовані такі методи дослідження:

теоретичний аналіз літератури та онлайн-документації щодо інформаційних систем;

системний підхід до формалізації потреб і структури системи;

моделювання структури бази даних;

метод прототипування для реалізації інтерфейсів;

емпіричне тестування функціоналу застосунку в різних режимах взаємодії;

методи UX-аналізу для оцінювання зручності та логіки взаємодії користувача із застосунком.

Наукова новизна роботи полягає у такому:

вперше визначено концептуальну модель веб-застосунку обліку витрат у домогосподарстві з розмежуванням ролей між найманим працівником і замовником;

вдосконалено механізм взаємодії між домоуправителем та господарем у цифровому середовищі за рахунок спрощеної моделі керування запитами на витрати;

дістала подальшого розвитку ідея гнучкого багаторольового підходу у побутовому застосунку, де користувачі взаємодіють асинхронно, але в рамках єдиної бази даних;

створено нову концепцію, що узагальнює функціональність звичайного менеджера витрат і цифрового помічника, адаптованого до специфіки житлового сектору;

розроблено нову систему, яка дозволяє не лише зберігати дані про витрати, а й здійснювати динамічну обробку запитів, зміну статусів, фільтрацію та логування активності користувачів;

з використанням відомого принципу розділення обов’язків (separation of concerns) реалізовано модульну структуру застосунку, що підвищує стабільність і масштабованість.

Теоретична цінність дослідження полягає в узагальненні підходів до побудови інформаційних систем у побутовій сфері, що раніше не були достатньо представлені в академічних розробках. Одержані результати можуть бути використані як основа для подальших досліджень у галузі персоналізованих цифрових сервісів, що взаємодіють із побутовими сценаріями.

Таким чином, застосування CRUD-операцій (створення, читання, оновлення, видалення) стало ключовим елементом, що формує логіку управління витратами в рамках функціонування інформаційної системи. Ця модель взаємодії з даними не лише задає чітку послідовність дій, але й виступає як основа для створення передбачуваного, надійного та масштабованого середовища, в якому всі процеси підпорядковані єдиному принципу організації. CRUD-операції реалізовано з урахуванням ряду важливих факторів: ролей користувачів, специфіки конкретних сценаріїв використання, поточного статусу об’єкта витрат, а також стандартів REST-архітектури, яка визначає логіку взаємодії між клієнтською частиною та сервером. Це забезпечує модульність, гнучкість і прозорість усіх запитів.

Впровадження такої моделі дало змогу досягти високого рівня надійності системи: вона стабільно функціонує навіть за умов активного навантаження та великої кількості користувачів. Ефективність обробки запитів гарантується чітким розмежуванням прав доступу — кожен учасник системи, незалежно від ролі (домоуправитель або господар), взаємодіє лише з тими даними та функціями, що входять до сфери його відповідальності. Наприклад, домоуправитель має виключне право на створення нових записів про витрати, тоді як господар здійснює функції контролю, перевірки й затвердження. Оновлення чи видалення інформації можливе лише в чітко визначених умовах — за наявності відповідного статусу об’єкта та ролі користувача, що усуває ризик помилок чи зловживань.

Такий розподіл обов’язків забезпечує як логічну послідовність дій, так і контроль за кожною зміною в системі, що особливо важливо в контексті управління спільним майном. Крім того, CRUD-модель створює надійний фундамент для подальшого масштабування: додавання нових типів витрат, сценаріїв чи функцій не потребує повної перебудови архітектури. Замість цього використовується розширення вже існуючих механізмів — через додаткові ендпойнти, розширені перевірки або адаптацію бізнес-логіки під нові ролі чи правила взаємодії.

Система також інтегрує сучасні засоби кібербезпеки — авторизацію на основі ролей, валідацію всіх вхідних даних на рівні серверу та клієнта, а також багаторівневий контроль доступу, що дозволяє уникнути витоків інформації та

несанкціонованих дій. Усе це в сукупності формує функціональне ядро веб-застосунку, яке відповідає найактуальнішим вимогам до програмного забезпечення у сфері житлового управління.

Таким чином, система не тільки задовольняє базові технічні й бізнесові потреби, але й створює гнучке, довготривале та безпечне середовище для всіх учасників процесу. Завдяки продуманій архітектурі на основі CRUD-операцій, вона підтримує прозорість, підзвітність і повну контрольованість взаємодії з даними, що є критично важливим для ефективного управління житловим фондом у сучасних умовах.

1 ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПІДХОДІВ ДО СТВОРЕННЯ ВЕБ-ЗАСТОСУНКІВ ДЛЯ ОБЛІКУ ВИТРАТ У СФЕРІ ДОМАШНЬОГО ГОСПОДАРСТВА

1.1 Аналіз існуючих веб-застосунків для обліку витрат

Предметна область розробки веб-застосунку для обліку витрат найманого домоуправителя та домогосподаря охоплює взаємодію між працівником, що виконує функції управління побутовими процесами, і власником домогосподарства, який контролює витрати, погоджує або відхиляє запити на закупівлю. У сучасних умовах дедалі більше домогосподарств наймають осіб для виконання щоденних управлінських і організаційних функцій, включаючи планування закупівель, моніторинг витрат, комунікацію з постачальниками, догляд за нерухомістю, контроль ремонтних робіт та інші завдання, що раніше виконувалися безпосередньо власниками. Це обумовлено зміною темпу життя, зростанням доходів середнього класу та поширенням тенденцій делегування побутових обов’язків. За таких умов виникає необхідність в інструментах, які б забезпечували прозоре управління фінансами в межах житлового простору. На відміну від корпоративних систем бухгалтерського обліку, предметна область побутового обліку витрат не вимагає багаторівневих звітів, інтеграції з податковими сервісами чи складних моделей амортизації. Натомість вона фокусується на простому, швидкому, інтуїтивному фіксуванні фактів витрат або запитів на них із можливістю підтвердження або відхилення.

Ключовими характеристиками цієї предметної області є асиметрія ролей (один користувач ініціює дію, інший її підтверджує), необхідність історичної фіксації витрат (час, категорія, сума, статус), конфіденційність інформації (дані мають бути доступні лише конкретним особам), а також потреба в асинхронному доступі — користувачі не перебувають в системі одночасно, але повинні взаємодіяти через запити. Таким чином, домоуправитель виконує роль ініціатора

витрат, він створює запити з описом потреби, зазначенням суми, кількості одиниць товару чи послуги, вибором категорії витрат. Ці запити мають бути збережені в системі та передані на розгляд господарю. Господар, у свою чергу, повинен мати змогу переглянути запити, переглянути деталі кожного з них, змінити статус (схвалити або відхилити), за необхідності — видалити некоректні або зайві запити. Усі ці дії повинні відбуватися в рамках єдиного інформаційного простору з чітким розмежуванням доступу, щоб уникнути несанкціонованих змін. У предметній області також важливо забезпечити мінімізацію бар'єрів у користуванні: система має бути настільки простою, щоб не вимагати попередньої технічної підготовки. Таким чином, особлива увага приділяється логіці побудови інтерфейсу, механізмам збереження та оновлення даних, можливості працювати в умовах мінімального інтернет-з'єднання.

З огляду на специфіку ролей у системі, веб-застосунок має враховувати кілька фундаментальних процесів, притаманних цій предметній області. По-перше, це ініціація запиту: форма має бути мінімалістичною, але інформативною. Передбачено вибір категорії (наприклад, побутова хімія, продукти, ремонтні роботи тощо), вказання суми, опис потреби, можливо, прикріплення фотографії або чека. По-друге, це верифікація: господар повинен бачити всі запити в хронологічному порядку, з фільтрацією за категоріями або статусами. Він повинен мати змогу оцінити кожен запит окремо, внести рішення, змінити статус — і все це без ризику втрати даних. По-третє, це моделювання життєвого циклу запиту: кожен запис у системі проходить статуси «очікує розгляду», «схвалено», «відхилено», при цьому зміна статусу має бути прозорою, з можливістю подальшої аналітики (напр., скільки було схвалено, скільки відхилено, яка сума витрачена за певний період). По-четверте, важливим є збереження повної історії транзакцій — щоб як домоуправитель, так і господар могли бачити свої дії в динаміці.

Додатково слід відзначити, що предметна область також включає психологічні та соціальні чинники, які впливають на характер взаємодії між сторонами. Домоуправитель, як найманий працівник, не повинен мати надмірного доступу до фінансів, але повинен мати змогу обґрунтовано ініціювати витрати.

Господар, зі свого боку, повинен мати довіру до системи, переконання у захищеності даних і можливості перегляду всіх дій у зручному форматі. У зв'язку з цим, система має бути побудована не лише з точки зору технічної ефективності, але й з урахуванням етики цифрової взаємодії, прозорості обліку, простоти верифікації дій, мінімізації конфліктів. У підсумку, предметна область цієї розробки — це комплексна система цифрової взаємодії між двома ролями в межах побутового процесу управління витратами. Вона охоплює аспекти фінансового планування, відповідальності, звітності, безпеки й комунікації, що й обумовлює унікальність підходів до створення програмного забезпечення для цього сегменту. Розуміння глибинної структури взаємодії в цій сфері дозволяє розробити інструмент, який не лише виконує технічну функцію, але й підвищує якість взаємин, знижує ризики непорозумінь і забезпечує комфортне співіснування в межах цифрового простору.

Ринок програмного забезпечення для обліку витрат у побуті активно розвивається протягом останнього десятиліття у відповідь на потребу користувачів у впорядкуванні особистих та сімейних фінансів. Основу цього ринку складають мобільні застосунки та онлайн-сервіси, які дозволяють фіксувати витрати, категоризувати їх, будувати графіки, встановлювати бюджети, контролювати залишки на рахунках. Найпоширенішими представниками цього сегмента є застосунки типу CoinKeeper, Monefy, Spendee, Wallet, ZenMoney, YNAB (You Need A Budget) тощо. Вони охоплюють широку аудиторію користувачів, орієнтуючись переважно на індивідуальне або сімейне користування, тобто взаємодію між людьми, які мають спільний доступ до фінансів. Переважна більшість таких програм реалізує модель фінансового трекінгу: користувач вносить свої витрати вручну або синхронізує банківські рахунки, після чого отримує зведену статистику за категоріями, періодами, балансами. Зазвичай акцент робиться на візуалізації — графіки, діаграми, інтерактивні дашборди — що дозволяє швидко оцінити фінансовий стан. Однак майже всі ці програми орієнтовані на користувача як єдину авторизовану особу. Навіть якщо застосунок дозволяє створення кількох профілів у межах однієї сім'ї, функціонал залишається рівноцінним — всі користувачі

мають однакові права на внесення та редагування інформації. Така структура не підходить для випадків, коли потрібне чітке розмежування повноважень між найманим працівником (який ініціює витрати) і власником (який їх контролює та ухвалює рішення).

Дослідження ринку показало, що наявні рішення не забезпечують достатньої підтримки сценаріїв з ієрархією користувачів. Наприклад, CoinKeeper дозволяє швидко додавати витрати через інтуїтивний drag-and-drop інтерфейс, але не має механізмів погодження транзакцій іншою особою. Monefy зручний у використанні, однак не підтримує багаторівневого доступу. Wallet має потужну систему бюджетування, але навіть у преміум-версії не передбачає розмежування прав доступу для виконавця і контролера. YNAB концентрується на плануванні та поведінкових аспектах витрат, мотивуючи користувача до фінансової дисципліни, проте абсолютно не підходить для сценарію "керівник-виконавець". Програмні рішення для малого бізнесу, наприклад QuickBooks або Zoho Books, містять багатокористувацький функціонал, однак є надмірно складними, орієнтованими на бухгалтерське середовище, з термінологією, що не відповідає побутовій сфері. Крім того, більшість подібних сервісів платні, з підписками або одноразовими ліцензіями, що не завжди доцільно для простих побутових потреб. На ринку також присутні окремі українські рішення, такі як «Монобюджет» або «Фінансовий помічник», однак вони мають обмежену функціональність, часто обмежуються тільки мобільною платформою, не дозволяють зручно розгортати власний веб-застосунок або змінювати структуру доступу.

Окрему групу становлять кастомізовані Google-таблиці та шаблони Excel, які іноді використовуються родинами чи приватними особами для ведення спільного обліку. Вони дозволяють гнучко налаштовувати формули, звіти, підсумки, але вимагають знання табличних процесорів, не мають автоматизованої перевірки прав доступу та не є повноцінними системами керування запитами. Крім того, їх неможливо інтегрувати у модель «запит → схвалення/відхилення» без додаткового програмування. Таким чином, навіть попри широку доступність програмних засобів для особистих фінансів, на ринку відсутні вузькоспеціалізовані рішення, які

б дозволяли організовувати побутові фінансові потоки між двома нерівноправними учасниками, де один є виконавцем, а другий — замовником або власником бюджету. У контексті найманої праці в домогосподарстві, це є критичною функціональною вимогою.

Вивчення ринку також показало, що інтерфейси сучасних програм часто перевантажені функціями, які дублюють або ускладнюють роботу звичайного користувача. Наприклад, можливість синхронізації з банківськими рахунками, імпорт транзакцій, автоматичне розпізнавання витрат — усе це, з одного боку, зручно для просунутих користувачів, але з іншого — є зайвим у побутових сценаріях, де важлива простота. У найманих працівників, зокрема домоуправителів, не завжди є доступ до банківських рахунків господаря, і функції автоматичного зчитування даних у цьому випадку не працюють. Більше того, системи обліку, орієнтовані на мобільні пристрої, не завжди зручні для господарів, які воліють контролювати витрати через веб-інтерфейс на ноутбуці або стаціонарному комп'ютері. Тому важливим є не лише функціональне, але й платформне охоплення, а також адаптивність інтерфейсу до різних типів пристроїв.

Загалом аналіз ринку дозволяє сформулювати кілька ключових висновків.

По-перше, існує розрив між застосунками для особистого обліку та корпоративними ERP-системами — жодна з них повною мірою не відповідає сценарію побутового керування витратами з розмежованим доступом.

По-друге, більшість програм не передбачають логіки запиту й погодження, що є центральною у випадку домоуправителя.

По-третє, український ринок практично не представлений якісними веб-застосунками із зазначеним функціоналом, що створює потенціал для створення нової цифрової системи, адаптованої до локального користувача.

Таким чином, ідея створення спеціалізованого веб-застосунку для обліку витрат у побуті не лише виправдана з прикладної точки зору, але й має високий потенціал комерціалізації або використання в обмежених замкнутих екосистемах (наприклад, у VIP-апартаментах, управляючих компаніях, сервісах оренди преміум-житла).

Це визначає доцільність подальших етапів роботи — проєктування, моделювання та реалізації такої системи, що враховуватиме виявлені недоліки наявних рішень і задовольнятиме специфічні вимоги цієї ринкової ніші.

Таблиця 1.1

Порівняльний аналіз популярних застосунків

Назва застосунку	**Платформа**	**Багаторольовий доступ**	**Орієнтація**	**Підтримка погодження витрат**
CoinKeeper	Мобільна	Ні	Індивідуальна	Ні
Monefy	Мобільна	Ні	Індивідуальна	Ні
Wallet	Веб + мобільна	Обмежено	Сімейна	Ні
YNAB	Веб	Ні	Особистий бюджет	Ні
QuickBooks	Веб	Так	Бізнес	Так

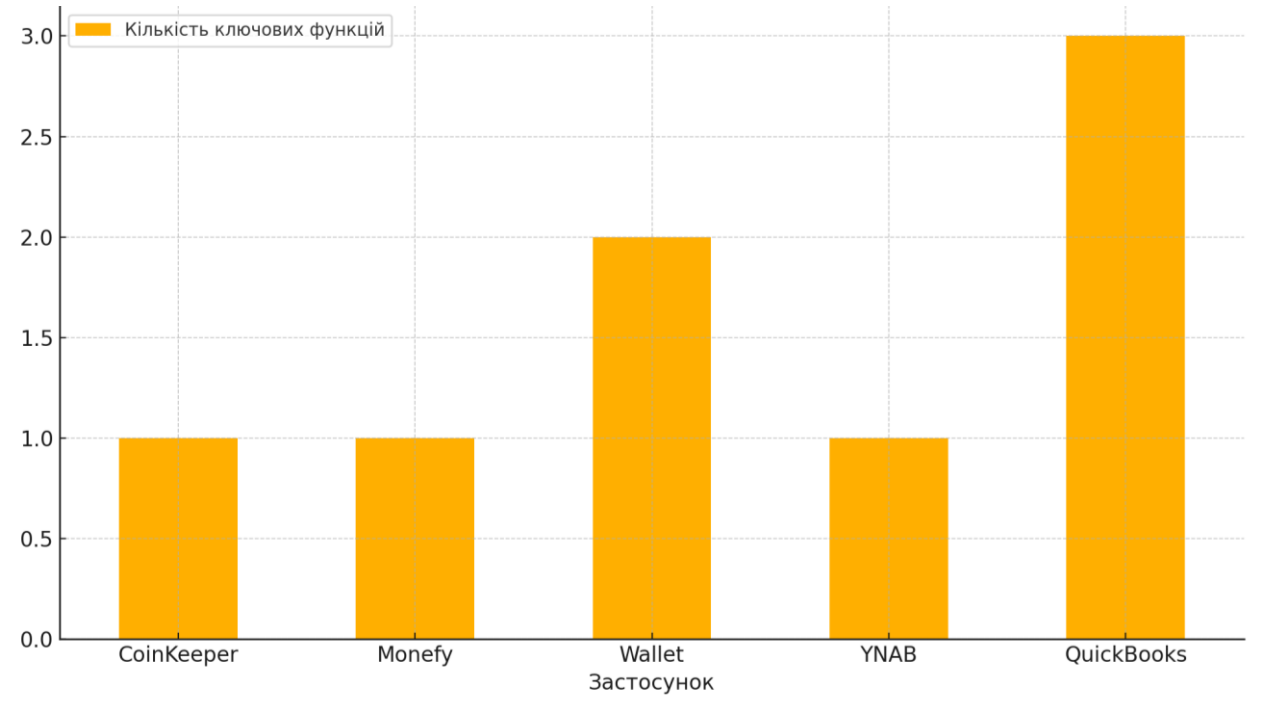


Рис. 1.1 Кількість ключових функцій у кожному застосунку

1.2 Обґрунтування вибору підходів і технологій для створення веб-застосунку

Вибір серверного середовища є критично важливим етапом під час проєктування веб-застосунку, адже саме серверна частина забезпечує обробку запитів клієнта, роботу з базою даних, реалізацію логіки бізнес-процесів і взаємодію між різними компонентами системи. У процесі розробки веб-застосунку для обліку витрат між найманим домоуправителем і господарем було прийнято рішення використати Node.js як основне серверне середовище. Це рішення базується на низці технічних, функціональних та економічних чинників, які визначають доцільність використання цієї платформи в контексті завдань, що поставлені перед розробкою. Node.js — це середовище виконання JavaScript на стороні сервера, побудоване на рушії V8 від Google, що забезпечує надзвичайно швидке виконання коду. Його неблокуюча модель введення/виведення (non-blocking I/O) та подієво-орієнтована архітектура дозволяють обробляти велику кількість одночасних з’єднань із мінімальним споживанням ресурсів, що ідеально підходить для веб-застосунків, орієнтованих на асинхронну взаємодію між користувачами. У контексті обраної предметної області, де один користувач ініціює запит, а інший його переглядає, змінює статус або видаляє, саме асинхронна модель є оптимальною, оскільки забезпечує швидку реакцію системи, миттєву обробку запитів і високу продуктивність при малій затримці. На відміну від традиційних багатопотокових моделей, які потребують значних ресурсів для підтримки паралельності, Node.js дозволяє використовувати єдиний потік з подієвим циклом, зменшуючи витрати на обробку запитів.

Додатковим аргументом на користь Node.js є використання однієї мови програмування — JavaScript — як на клієнтському, так і на серверному боці. Це значно спрощує розробку, оскільки зменшує кількість знань, необхідних для підтримки проєкту, та усуває бар’єри між фронтендом і бекендом. Завдяки цьому забезпечується послідовність логіки обробки даних, легше реалізуються перевірки введення, валідація, форматування та обробка відповідей. Вибір Node.js дозволив

реалізувати серверну частину за допомогою фреймворку Express.js — мінімалістичного, але потужного інструменту для створення API та маршрутизації. Express забезпечує ефективне управління маршрутами, обробку запитів POST, GET, PUT, DELETE, роботу з middleware-функціями, а також легку інтеграцію з базами даних через сторонні бібліотеки. Така структура дозволяє створити чітку, логічно організовану систему, де кожен компонент відповідає за конкретну функцію: обробка авторизації, взаємодія з базою, управління витратами, маршрутизація та інше.

Важливо також відзначити, що екосистема Node.js надзвичайно розвинена: завдяки менеджеру пакетів npm розробник має доступ до сотень тисяч бібліотек, які покривають усі потреби проєкту — від підключення до бази даних до шифрування, логування, тестування й обробки помилок. Це дає змогу зосередитися на логіці застосунку, не витрачаючи ресурси на створення типових рішень. Крім того, Node.js є кросплатформеним рішенням — серверна частина може бути розгорнута на будь-якій операційній системі (Windows, Linux, macOS), що спрощує інфраструктуру. У випадку локального тестування, Node.js працює швидко та стабільно навіть на малопотужному обладнанні. Це робить його доступним для розробників-початківців, що важливо в умовах обмежених ресурсів. Node.js також активно підтримується спільнотою, регулярно оновлюється, має потужну документацію й багату базу прикладів, що сприяє швидкому вирішенню проблем і навчанню. Платформа використовується такими технологічними гігантами, як Netflix, Uber, LinkedIn, PayPal, що свідчить про її надійність і масштабованість.

Ще одним аргументом є легкість реалізації RESTful API. Node.js дозволяє швидко створювати маршрути, що відповідають принципам REST (розділення запитів за дієсловами HTTP), що значно підвищує читабельність і підтримуваність коду. Це особливо важливо для проєктів, які можуть розширюватися в майбутньому, наприклад, при додаванні мобільного застосунку або інтеграції з іншими сервісами. У створеному веб-застосунку REST API реалізує основні дії з об’єктами витрат: створення, читання, оновлення статусу та видалення. Весь обмін даними здійснюється у форматі JSON, що є сучасним стандартом обміну в мережі.

Використання Node.js уможливило також реалізацію захищеної авторизації з перевіркою ролі користувача, що необхідно для предметної області з диференційованим доступом. Логіка перевірки виконується на рівні контролерів, де здійснюється зіставлення переданих даних із записами в базі, визначення прав доступу, формування відповіді з userId і userRole, що зберігаються на клієнтському боці та використовуються для маршрутизації.

Таким чином, вибір Node.js як серверного середовища був зумовлений необхідністю високої продуктивності, підтримки асинхронної взаємодії, простоти в реалізації API, доступності, активної підтримки спільнотою, широкого набору бібліотек, та можливості реалізації повноцінної логіки без залучення зайвих технологій. Саме ця платформа дозволила створити ефективну серверну частину, яка не тільки виконує всі заплановані функції, але й забезпечує основу для подальшого масштабування та розвитку системи. У межах поставлених завдань Node.js виявився найбільш гнучким, функціональним і логічним вибором, що повністю узгоджується зі специфікою предметної області, структурою застосунку та архітектурними принципами сучасного веб-програмування.

Одним із ключових компонентів у розробці будь-якого веб-застосунку є система управління базами даних (СУБД), яка відповідає за зберігання, обробку, організацію та вибірку інформації. У контексті створення веб-застосунку для обліку витрат між найманим домоуправителем і господарем основна роль СУБД полягає у збереженні даних про користувачів, запити на витрати, категорії товарів і статуси погодження. Зважаючи на характер даних, їхню структурованість, необхідність забезпечення надійності, швидкої обробки запитів та простоти інтеграції з серверним середовищем Node.js, для реалізації проєкту було обрано реляційну СУБД MySQL. Основною причиною вибору MySQL є її стабільність, перевірена часом ефективність, відкрите ліцензування та широка підтримка в середовищі розробників. MySQL — одна з найпопулярніших СУБД у світі, що забезпечує швидкий доступ до структурованих даних, підтримує транзакції, індексацію, зв’язки між таблицями та дотримання цілісності бази. У нашому випадку система зберігає інформацію у вигляді чітко визначених таблиць:

користувачі (із зазначенням ролі), запити на витрати (з описом, сумою, кількістю, категорією та статусом), категорії витрат та інші допоміжні таблиці. Реляційна модель даних ідеально підходить для цих цілей, оскільки дозволяє точно формалізувати зв’язки між сутностями: один користувач може мати багато запитів, кожен запит належить до певної категорії та має свій статус. Це дозволяє реалізовувати складні вибірки — наприклад, вивести всі запити конкретного користувача за певний період або всі відхилені запити в певній категорії — без надмірного ускладнення структури даних.

Важливою перевагою MySQL є її повна сумісність із Node.js через численні бібліотеки, зокрема mysql2, яка була використана в проєкті для підключення до бази даних і виконання запитів з використанням пулу з’єднань. Це забезпечує ефективне управління ресурсами, уникнення перевантаження при великій кількості одночасних запитів та високу продуктивність системи. MySQL дозволяє налаштувати обмеження доступу до бази, визначати права для користувачів, вести логування транзакцій, що підвищує рівень безпеки й відповідає вимогам збереження конфіденційної інформації, зокрема персональних даних. Ще одним чинником вибору стало широке розповсюдження та доступність MySQL. Вона підтримується більшістю хостингів, має вбудовані інструменти для резервного копіювання, реплікації та моніторингу. Це дозволяє легко розгортати застосунок у продакшн-середовищі, переносити базу даних між середовищами розробки, тестування та експлуатації. Простота міграцій між локальним сервером і віддаленим хостингом — ще один аргумент на користь MySQL. Крім того, сама структура мови SQL є прозорою, логічною та легкою у використанні, що дозволяє реалізовувати будь-які запити до бази на рівні, зрозумілому навіть розробникам-початківцям.

З технічної точки зору, MySQL забезпечує оптимальне поєднання гнучкості та контролю над даними. За допомогою первинних і зовнішніх ключів було реалізовано зв’язки між таблицями, що дозволило зберегти логічну цілісність системи. Наприклад, кожен запис про витрати має посилання на користувача, який його створив, а також на категорію, до якої він належить. Таким чином, навіть при

масштабуванні системи з великою кількістю користувачів структура даних залишається впорядкованою, а обробка запитів — швидкою. Індексація дозволяє оптимізувати пошук за ключовими полями, зменшуючи час відповіді сервера, що особливо важливо при виконанні фільтрації або сортування за статусами, датами чи категоріями. Для реалізації життєвого циклу запиту на витрати використано логічне поле "статус", що дозволяє контролювати процес від створення до схвалення або відхилення. Такий підхід забезпечує просте розширення — наприклад, можна легко додати нові статуси ("очікує коментаря", "повернуто на доопрацювання" тощо) без зміни структури бази.

Особливу увагу слід приділити питанню безпеки — MySQL дозволяє організовувати авторизований доступ до бази даних, розмежовувати рівні доступу, шифрувати з’єднання, створювати ролі для адміністрування. Це особливо важливо у випадках, коли система працює з персональними або фінансовими даними, навіть у побутовому середовищі. Наприклад, доступ до операцій створення, оновлення чи видалення запитів мають лише авторизовані користувачі, а самі дії проходять перевірку через бекенд. Усі запити до бази формуються захищено, із використанням підготовлених SQL-інструкцій, що мінімізує ризик SQL-ін’єкцій. Резервне копіювання даних, підтримуване MySQL, дозволяє створювати автоматичні бекапи, що гарантує збереження інформації навіть у разі збою сервера або втрати підключення.

На відміну від нереляційних СУБД, таких як MongoDB, які орієнтовані на зберігання неструктурованих або слабо структурованих даних, MySQL забезпечує строгий контроль над схемою бази, що дозволяє чітко верифікувати кожен запис, запобігаючи введенню неповних або некоректних даних. У нашому проєкті структура таблиць розроблена таким чином, щоб унеможливити дублювання інформації, забезпечити зв’язність і логічну послідовність. Це особливо важливо для подальшої генерації звітів або аналітики. Наприклад, у перспективі можна реалізувати модуль аналітики витрат, який на основі збережених даних формуватиме діаграми за місяць, категорію чи користувача. Саме структурованість

даних, характерна для MySQL, робить таку функціональність реалізованою без додаткових складнощів.

Загалом використання MySQL як основної СУБД у проєкті є результатом збалансованого технічного вибору, що ґрунтується на реальних вимогах до системи: необхідності зберігати структуровані дані, забезпечити швидкий доступ, підтримку транзакцій, збереження зв’язків між таблицями, безпечну аутентифікацію та ефективне масштабування. Система повністю відповідає потребам побутового застосунку з багаторівневим доступом і логікою запитів на витрати. У поєднанні з Node.js MySQL утворює потужний та гнучкий стек технологій, який забезпечує стабільну роботу, високу швидкодію, зручність підтримки та майбутню модернізацію. Це дозволяє не тільки досягти поставлених цілей у межах цього проєкту, але й створити підґрунтя для подальшого розвитку системи.

При створенні веб-застосунку для обліку витрат, у якому взаємодіють дві основні ролі користувачів — найманий домоуправитель і господар, особливо важливим є питання організації взаємодії між клієнтською частиною інтерфейсу та сервером, де зберігається логіка обробки даних. Для забезпечення ефективного, масштабованого і зрозумілого способу обміну інформацією між клієнтом і сервером було обрано архітектурну модель REST API (Representational State Transfer Application Programming Interface). REST є найбільш поширеним і загальноприйнятим підходом до побудови веб-сервісів завдяки своїй простоті, зрозумілості, легкій інтеграції та незалежності від платформи. У рамках архітектури REST передбачено, що всі запити від клієнта до сервера здійснюються через стандартні HTTP-методи — GET, POST, PUT, DELETE, що надає чітку семантику кожній операції. Такий підхід дозволяє реалізувати CRUD-функціональність (створення, читання, оновлення, видалення) для ключових об'єктів системи, зокрема запитів на витрати, користувачів, категорій та статусів. Наприклад, коли домоуправитель створює новий запит, викликається метод POST /api/expenses, який передає дані до сервера у форматі JSON; для перегляду всіх запитів господарем використовується метод GET /api/expenses; зміна статусу

відбувається через PUT /api/expenses/:id, а видалення запиту — через DELETE /api/expenses/:id.

REST дозволяє легко масштабувати застосунок, оскільки клієнт і сервер розвиваються незалежно один від одного. Завдяки відсутності стану (statelessness), кожен запит містить усю необхідну інформацію для його обробки, не потребуючи збереження стану на сервері. Це дозволяє зменшити навантаження на сервер, а також спрощує балансування навантаження при масштабуванні системи. У створеному проєкті реалізація REST API дає змогу відокремити логіку інтерфейсу користувача від логіки роботи з базою даних, що суттєво підвищує стабільність і гнучкість системи. Наприклад, користувачі взаємодіють із веб-інтерфейсом (HTML/CSS/JS), який, у свою чергу, звертається до REST API-запитів. Така структура дозволяє в майбутньому підключити до сервера інші клієнтські інтерфейси — мобільний застосунок, десктопну версію чи інший тип панелі адміністратора — без зміни серверної логіки. Більше того, REST API відповідає принципам модульності: кожен маршрут виконує чітко визначену функцію, а отже, легкий у підтримці, тестуванні та масштабуванні.

Однією з важливих причин використання REST стала необхідність побудови захищеної моделі взаємодії з можливістю автентифікації та авторизації користувачів. REST API дозволяє реалізувати логіку доступу через передачу токена або інформації про сесію, забезпечуючи таким чином обробку запитів із урахуванням ролі користувача. У нашому випадку після автентифікації клієнт отримує відповідь, що містить userId та userRole, які зберігаються в локальному сховищі браузера (localStorage). На основі цієї інформації система приймає рішення про доступність функціоналу: менеджер бачить лише власні запити, може створювати нові, але не має права на схвалення чи видалення; натомість господар бачить усі запити, може змінювати їхній статус і видаляти. REST дозволяє зручно структурувати ці обмеження, оскільки кожен маршрут реалізує окрему бізнес-логіку з перевіркою авторизації на рівні контролера. Такий підхід відповідає сучасним вимогам до безпечних інформаційних систем, де розмежування повноважень і захист даних є пріоритетом.

REST також забезпечує високу зрозумілість коду як для розробників, так і для сторонніх фахівців, які можуть у майбутньому підтримувати або модифікувати систему. Адреси маршрутів мають логічну структуру, що чітко відображає ієрархію даних: /api/expenses — усі запити, /api/expenses/:id — конкретний запит, /api/auth/login — вхід у систему, /api/categories — список категорій. Усі запити відправляються й отримуються у форматі JSON, що є зручним, легким для парсингу й сумісним з більшістю клієнтських бібліотек. Крім того, REST API легко тестувати — як вручну (через Postman або curl), так і автоматизовано (через бібліотеки типу Jest чи Mocha), що дозволяє швидко перевірити правильність реалізації без залежності від візуального інтерфейсу. Це сприяє підвищенню якості розробки, пришвидшує виявлення помилок і дозволяє забезпечити стабільну роботу системи.Р

Загалом архітектура REST API виявилася найбільш відповідною для створення веб-застосунку, орієнтованого на взаємодію клієнта й сервера в режимі запит–відповідь. Вона забезпечує високу ефективність, модульність, незалежність компонентів, підтримку масштабованості та безпечну логіку доступу. Завдяки чіткому структуруванню запитів REST API дозволяє підтримувати прозору й керовану взаємодію між інтерфейсом користувача та логікою обробки даних, що особливо важливо у системах, де взаємодія має форму погодження витрат, контролю транзакцій і розмежування функцій за ролями. У поєднанні з Node.js та MySQL така архітектура дозволяє побудувати надійний, масштабований і зручний у підтримці програмний продукт, що повністю відповідає сучасним вимогам до веб-застосунків у побутовій сфері.

У процесі розробки веб-застосунку для обліку витрат між найманим домоуправителем і господарем одним із важливих рішень стало свідоме відхилення від використання популярних фреймворків інтерфейсу, таких як Bootstrap, Materialize, Tailwind CSS тощо, на користь повністю кастомного підходу до створення клієнтської частини з використанням нативного HTML та CSS. Це рішення не було спонтанним чи зумовленим обмеженнями інструментів — навпаки, воно ґрунтувалося на низці об'єктивних причин, пов'язаних із

особливостями проєкту, потребами цільової аудиторії, вимогами до структури інтерфейсу та прагненням до максимальної гнучкості й оптимізації. По-перше, більшість CSS-фреймворків, попри зручність у швидкій побудові типових інтерфейсів, накладають певні обмеження на гнучкість дизайну, особливо коли необхідно реалізувати індивідуальні сценарії взаємодії, специфічну логіку відображення елементів чи мінімалістичний стиль із чітко визначеною колористикою. У випадку даного веб-застосунку ключовим було створення двох відокремлених інтерфейсів — для домоуправителя і господаря, кожен із яких має власний набір функцій, відображає різні блоки інформації та передбачає різний набір елементів управління. Наприклад, інтерфейс домоуправителя орієнтований на створення нових запитів через форму з категорією, кількістю, сумою та описом, тоді як інтерфейс господаря — на перегляд запитів із можливістю погодження, відхилення або видалення. Фреймворки не завжди дозволяють реалізувати таку специфіку без надлишкової модифікації, внаслідок чого часто виникає ситуація, коли кількість перевизначень CSS-класів перевищує користь від самого фреймворку.

По-друге, кастомний HTML/CSS забезпечує мінімальну вагу сторінок і швидке завантаження, що є критичним чинником для забезпечення комфортної взаємодії з системою навіть у разі слабкого інтернет-з’єднання або використання застарілих пристроїв. Фреймворки, особливо універсальні, включають значну кількість стилів, компонентів і класів, які не використовуються в конкретному проєкті, проте все одно завантажуються браузером, тим самим створюючи надмірне навантаження. Відмова від них дозволяє створити строго необхідну стилізацію з нуля, використовуючи тільки ті правила, які дійсно застосовуються в інтерфейсі. Наприклад, у стилях було реалізовано власну систему змінних кольорів, що дозволило створити уніфікований, спокійний і зрозумілий візуальний стиль без перевантаження графікою чи анімаціями. Кнопки, таблиці, форми та повідомлення — усі ці елементи були стилізовані вручну відповідно до потреб конкретної ролі користувача. Це дало змогу уникнути шаблонності в дизайні та зробити інтерфейс більш «легким» з точки зору сприйняття й навігації.

Ще одним мотивом на користь кастомного підходу стала необхідність повного контролю над адаптивністю інтерфейсу. Більшість сучасних фреймворків мають вбудовану підтримку адаптивної верстки, але вона реалізована на основі загальних брейкпоінтів, які не завжди відповідають реальним умовам використання конкретного застосунку. У нашому випадку користувачами системи можуть бути як працівники з планшетом, так і господарі з десктопом, тому було важливо налаштувати адаптивність вручну, враховуючи лише ті діапазони екранів, які справді мають значення для цільової аудиторії. Ручне створення медіазапитів дало змогу оптимізувати відображення таблиць, форм, панелей керування так, щоб вони не ламалися при зміні розміру, зберігали логіку взаєморозміщення елементів і водночас не вимагали перевантаження DOM-структури зайвими обгортками чи сітками. Це також дозволило точно налаштувати типографіку, кольори, відступи — без залежності від типових фреймворкових класів.

З функціональної точки зору кастомний підхід дав можливість реалізувати інтерактивну поведінку елементів за допомогою JavaScript без конфліктів із сторонніми бібліотеками або залежностями. Наприклад, при створенні нового запиту форма динамічно очищується після відправлення, таблиця оновлюється без перезавантаження сторінки, статуси відображаються у вигляді кольорових тегів з індивідуальною стилізацією, а кнопки змінюють свій вигляд відповідно до дії (наприклад, при відхиленні чи схваленні). Усе це реалізовано з використанням базового JavaScript та CSS, без JQuery чи UI-компонентів, що дозволяє краще контролювати логіку подій, відлагоджувати анімації й уникати зайвих залежностей. Відмова від фреймворків також позитивно впливає на безпеку — менша кількість сторонніх компонентів означає менше потенційних вразливостей, менше точок атаки та простіший процес аудиту коду.

З точки зору довготривалої підтримки, використання кастомного HTML/CSS є безсумнівною перевагою, оскільки вся система розроблена на основі власного, добре контрольованого коду, без залежності від сторонніх бібліотек, фреймворків або їхніх оновлень. Такий підхід забезпечує стабільність зовнішнього вигляду інтерфейсу, передбачуваність поведінки компонентів і дозволяє поступово

вдосконалювати кожен елемент UI без ризику раптового порушення сумісності, як це часто трапляється при використанні готових рішень.

Наприклад, оновлення популярного CSS-фреймворку — такого як Bootstrap з версії 4 до версії 5 може спричинити серйозні збої у відображенні інтерфейсу або повне порушення верстки через зміни в класах, структурі компонентів чи системі відступів. У випадку кастомної реалізації таких ризиків майже не існує: розробник повністю контролює кожен стиль, кожен блок і може локально, поступово змінювати або оптимізувати лише ті частини, які потребують оновлення. Це дає змогу уникнути ефекту «доміно», коли одне оновлення тягне за собою необхідність переглядати десятки або сотні сторінок.

У підсумку можна з упевненістю стверджувати, що рішення свідомо відмовитися від використання CSS-фреймворків на користь створення інтерфейсу на основі «чистого» HTML і CSS було не лише виправданим, а й стратегічно вмотивованим. Такий підхід дозволив реалізувати високий рівень гнучкості, індивідуальності дизайну, а також досягти точного контролю над кожним аспектом адаптивності, респонсивності та поведінки інтерфейсу на різних пристроях і розширеннях екранів.

Окрім цього, кастомний інтерфейс часто виявляється легшим і продуктивнішим: відсутність зайвого коду, властивого великим фреймворкам, знижує вагу сторінок, пришвидшує їх завантаження та загальну реактивність застосунку. Це позитивно впливає як на технічну продуктивність, так і на користувацький досвід, що особливо важливо в умовах обмежених ресурсів або для мобільних користувачів.

Загалом, для проєктів, у яких ключовими є прозора логіка, простота підтримки, адаптація під специфічні вимоги та високий рівень кастомізації, відмова від універсальних шаблонів і перехід до індивідуального рішення забезпечує кращу довготривалу стабільність, передбачуваність у розвитку системи та повну відповідність реальним потребам бізнесу чи цільової аудиторії. У такому контексті кастомний підхід беззаперечно перевершує фреймворкові альтернативи.

2 ХАРАКТЕРИСТИКА СИСТЕМИ ОБЛІКУ ВИТРАТ ТА МЕТОДИ ЇЇ ПОБУДОВИ

2.1 Структура і функціональні можливості системи

Формування функціональних ролей користувачів є ключовим аспектом у проєктуванні сучасних веб-застосунків, особливо якщо йдеться про системи з розмежуванням прав доступу та обов’язків. У межах розробленого веб-застосунку для обліку витрат, що виникають у процесі взаємодії між найманим домоуправителем та власником житла (господарем), реалізовано чітке розмежування функціональних ролей, що базується на реальній моделі управління домашніми господарствами. Роль користувача визначає не лише його доступ до певних інтерфейсів, а й логіку дозволених дій у системі, обсяг відповідальності, послідовність взаємодії із запитами на витрати. Такий підхід забезпечує прозорість процесів, запобігає несанкціонованим діям, гарантує цілісність даних і створює передумови для формалізації домашньої фінансової дисципліни. Найманий домоуправитель у цій системі виконує роль ініціатора витрат — він формує запити на придбання товарів або замовлення послуг, зазначаючи конкретну потребу, її категорію (наприклад, ремонт, продукти, побутова хімія), кількість одиниць, суму витрат та короткий опис призначення. Таким чином, домоуправитель виконує облікову функцію, фіксуючи потенційні витрати, які ще не відбулися, але потребують погодження. Його роль зосереджена на створенні повного й точного опису необхідних витратних заходів, які мають бути реалізовані в межах повсякденної діяльності домогосподарства. Для цього йому надається індивідуальний інтерфейс користувача, доступ до якого відкривається після авторизації. У межах цього інтерфейсу домоуправитель може переглядати створені ним запити, бачити їх статуси — «очікує», «схвалено», «відхилено», — редагувати дані до моменту їх схвалення, а також створювати нові запити за допомогою динамічної форми. Важливо, що домоуправитель не має змоги бачити запити,

створені іншими користувачами, або змінювати їхній статус — це обмеження реалізовано на рівні перевірки ролі користувача, що забезпечує дотримання принципу розмежування повноважень. Це також мінімізує ризики помилкових або несанкціонованих змін даних, що особливо важливо у системах, пов’язаних із фінансовими діями. У разі необхідності уточнення інформації або внесення правок домоуправитель має змогу скасувати запит і створити новий, що дозволяє вести гнучке управління процесами закупівель без порушення логіки погодження. Така рольова модель відповідає реальним умовам функціонування житлових господарств, де виконавець не має повного контролю над коштами, але відповідальний за ініціацію й формування витрат.

Натомість господар системи виконує функцію контролюючого та ухвалюючого користувача, який має повний доступ до інформації, створеної домоуправителем. У межах свого функціоналу господар бачить усі запити, незалежно від їх автора, має змогу відфільтровувати їх за категоріями, статусами, датами створення або сумами, переглядати повну інформацію щодо кожного окремого запиту, а також ухвалювати рішення — погодити або відхилити. Саме господар визначає остаточну долю кожного запиту, здійснюючи тим самим контроль витрат у системі. Його інтерфейс дозволяє миттєво змінювати статус запиту на «approved» або «rejected» з відповідним оновленням у базі даних, а також, за потреби, повністю видаляти запити. Це важливо в тих випадках, коли запит був створений помилково, дублюється або не має змістовного обґрунтування. Власник має повний спектр повноважень для оцінювання ефективності витрат, оптимізації бюджету та встановлення фінансової дисципліни в межах домогосподарства. Господар не створює запити, не редагує їх зміст, але має виняткове право на остаточне рішення, що чітко структуровано в рамках логіки системи. Усі дії господаря фіксуються на рівні бази даних, що дозволяє забезпечити історичний облік рішень, прозорість і звітність. Важливо також підкреслити, що система реалізує механізм обмеження доступу не лише до функцій, а й до інтерфейсних елементів: наприклад, у господаря відсутня форма створення запиту, натомість наявні кнопки для затвердження або скасування запитів, а також фільтри й панелі

перегляду. Такий поділ ролей дозволяє знизити когнітивне навантаження на кожного користувача, спростити навігацію й уникнути зайвих помилок. Архітектурно реалізація ролей базується на серверній перевірці прав доступу через систему авторизації: при вході в систему користувач проходить автентифікацію, отримує відповідь із призначеною роллю (userRole), яка зберігається у локальному сховищі браузера, після чого користувача перенаправляє на відповідну сторінку — manager.html або owner.html. На рівні REST API кожен маршрут має власні обмеження доступу — наприклад, змінити статус запиту може лише користувач із роллю «owner». Завдяки цьому забезпечується безпечна модель взаємодії, у якій виключено доступ до неавторизованих функцій, що є критично важливим при роботі з фінансовими даними. Розроблена система ролей у застосунку не лише відповідає технічним вимогам побудови багаторівневих інформаційних систем, а й органічно вбудовується в логіку повсякденного управління домашнім господарством. Вона формує зрозумілу, логічну, стабільну модель співпраці між виконавцем і замовником, дозволяє вести аналітику витрат, формалізує процес прийняття рішень і закладає основу для цифрової дисципліни. У перспективі цю модель можна розширити шляхом додавання нових ролей (наприклад, ревізор, бухгалтер, асистент) або впровадження гнучкої системи прав, що відкриває широкі можливості для адаптації застосунку під різні сценарії використання.

Взаємодія користувачів із системою в межах побудованого веб-застосунку реалізується через логічно продуману модель сценаріїв витрат, що відображає реальні потреби побутового середовища, у якому співпрацюють найманий домоуправитель і господар житлового приміщення. Основна мета такої взаємодії — забезпечити прозоре, просте та контрольоване керування ініціацією, погодженням і фіксацією витрат, які виникають у процесі функціонування домогосподарства. Користувачі системи мають чітко визначені ролі, які задають траєкторію їх дій у межах застосунку. Взаємодія починається з того, що домоуправитель, як ініціатор витрат, створює запит у відповідній формі інтерфейсу. Він заповнює поля, де вказує категорію витрати (наприклад, ремонт, продукти, комунальні послуги), орієнтовну суму, кількість одиниць або послуг, а

також текстовий опис необхідності витрати. Після заповнення форми дані зберігаються в базі даних і запит отримує статус «очікує». Цей запит автоматично стає доступним для господаря, який, увійшовши до свого облікового запису, бачить перелік усіх запитів, створених домоуправителем, упорядкованих за хронологією, категоріями або статусами. Господар аналізує зміст кожного запиту, використовуючи візуально зручну таблицю з деталями, і приймає рішення — схвалити витрату або відхилити її.

Сценарії витрат реалізуються в системі як послідовність дій між двома користувачами, кожен із яких працює в межах власного функціонального простору. У типовому сценарії домоуправитель створює запит на витрату з описом: «Потрібно придбати побутову хімію — миючий засіб, губки, засіб для миття підлоги», із сумою 350 грн, вибраною категорією «Побутова хімія» та кількістю одиниць — 3. Після натискання кнопки «Створити» цей запит передається на сервер, де зберігається у таблиці витрат із відповідною прив’язкою до користувача, який його створив.

У цей момент він ще не є фактичною витратою, оскільки не був погоджений. Господар бачить цей запит у своєму інтерфейсі, порівнює його з попередніми аналогічними витратами, може враховувати загальну суму витрат за місяць або встановлений внутрішній бюджет, після чого натискає «Схвалити» або «Відхилити». Якщо запит схвалено, його статус змінюється на «Схвалено», і домоуправитель бачить це у своєму інтерфейсі. Якщо ж витрата була визнана недоцільною або надто частою, господар натискає «Відхилити», що також оновлює статус запиту в базі даних. У такий спосіб кожен запит проходить цикл від ініціації до ухвалення або відмови, що забезпечує цілісність фінансового потоку й документування рішень. Усі дії зберігаються в системі, і за потреби користувачі можуть переглядати історію витрат, аналізувати частоту витрат на певні категорії, контролювати бюджет.

Крім базового сценарію створення та погодження запитів, система підтримує розширені сценарії, які враховують повторюваність витрат, відстеження частоти по категоріях, аналіз змін статусів та контроль за навантаженням на домоуправителя.

Наприклад, якщо домоуправитель регулярно створює запити в категорії «Господарські товари», система фіксує частоту таких витрат, що може бути враховано господарем під час ухвалення майбутніх рішень. У межах інтерфейсу господаря реалізовано фільтрацію: користувач може переглянути лише запити, які очікують схвалення, або лише ті, що вже були схвалені, що підвищує ефективність прийняття рішень.

У перспективі така функціональність дозволяє автоматизувати погодження певних типів витрат, наприклад, якщо сума менша за встановлений поріг, або якщо категорія витрати визнана звичною. Усі сценарії взаємодії реалізовано на основі REST API, що забезпечує стабільний і передбачуваний обмін даними між клієнтом і сервером. Авторизація користувачів дозволяє чітко ідентифікувати, хто саме створив або змінив запит, а зв’язок між користувачем і запитом фіксується в базі даних, що забезпечує повну прозорість і зворотність змін. Такий підхід відповідає не лише логіці цифрової системи, а й моделює реальну взаємодію між працівником і роботодавцем у межах приватного побуту, де довіра ґрунтується на чіткості, відповідальності та звітності.

Усі етапи взаємодії продумані з точки зору користувацького досвіду: створення запиту займає менше хвилини, усі поля логічно згруповані, система підказує можливі категорії, перевіряє правильність введення числових значень, а при зміні статусу відображається кольоровий індикатор (наприклад, зелений для схваленого, червоний для відхиленого). Таким чином, сценарії витрат у системі не лише імітують реальні побутові процеси, а й формують зручне цифрове середовище, де кожен учасник знає свої обов’язки, бачить результати своїх дій і може довіряти системі.

Таблиця. 2.1

Функціональні ролі користувачів

Роль	Дії у системі	Обмеження
Домоуправитель	Створення запитів на витрати, перегляд статусу, редагування до схвалення	Не бачить запити інших користувачів, не змінює статуси
Господар	Перегляд усіх запитів, зміна статусу (схвалити/відхилити), фільтрація, видалення	Не створює або редагує запити

У підсумку така модель взаємодії дозволяє уникати конфліктів, підвищує якість керування фінансами у домогосподарстві, забезпечує надійне цифрове середовище для довготривалої співпраці й дозволяє ефективно розподіляти відповідальність між учасниками процесу.

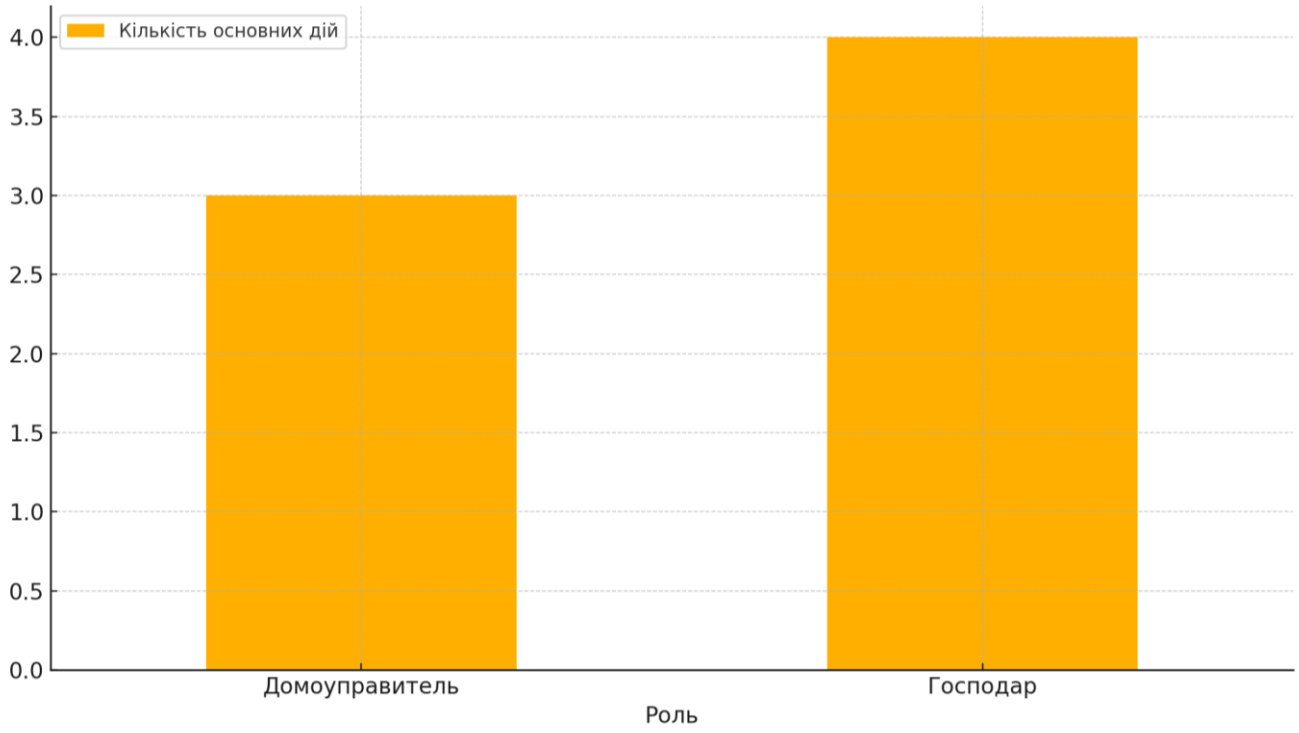


Рис. 2.1 Кількість основних дій у кожної ролі

2.2 Методи та моделі, використані при створенні веб-застосунку (MVC, REST, CRUD, ін.)

Під час розробки веб-застосунку для обліку витрат найманого домоуправителя та господаря ключовим принципом архітектурної організації проєкту стала реалізація патерну MVC (Model–View–Controller), який забезпечив логічне розділення функціональності, гнучкість, масштабованість та зручність у підтримці системи. Патерн MVC, що є одним із найпоширеніших у веб-програмуванні, базується на чіткому поділі логіки додатка на три взаємопов’язані компоненти — модель (Model), представлення (View) та контролер (Controller). У межах цього проєкту реалізація MVC дозволила не лише ефективно структуризувати код, але й адаптувати кожен компонент системи до потреб конкретної ролі користувача. Компонент Model у нашій системі відповідає за взаємодію з базою даних MySQL: він реалізує схеми таблиць, операції читання, створення, оновлення та видалення даних. Наприклад, моделі користувачів, запитів на витрати, категорій та ролей були реалізовані у вигляді окремих модулів із використанням SQL-запитів, структурованих за принципом CRUD. Саме модель забезпечує узгодженість даних, перевірку типів, роботу з первинними та зовнішніми ключами, а також взаємозв’язок між сутностями. Таким чином, модель є фундаментом для зберігання інформації про всі об'єкти системи — користувачів, їхні ролі, створені ними витрати, статуси погодження тощо. Усі запити до бази здійснюються через модуль Model, що дозволяє централізовано контролювати зміни в структурі таблиць, підтримувати їх актуальність і гарантувати цілісність інформації.

Компонент View у реалізованому застосунку відповідає за відображення даних та взаємодію з користувачем. Оскільки система має два типи користувачів — домоуправителя та господаря — для кожної ролі створено окремий набір HTML-сторінок з відповідним оформленням, функціональністю та логікою відображення. Наприклад, домоуправитель бачить лише форму створення запиту, список власних запитів зі статусами, тоді як господар має доступ до всіх запитів, фільтрації,

інструментів погодження або відхилення. Уся взаємодія реалізована на базі нативного HTML/CSS/JS, що дозволило уникнути використання фреймворків та зберегти максимальну контрольованість над відображенням елементів. Представлення (View) отримує дані через REST API-запити до контролера, після чого за допомогою JavaScript динамічно оновлює інтерфейс без необхідності перезавантаження сторінки. Це дозволяє забезпечити зручний користувацький досвід, у якому зміни в базі одразу відображаються у візуальному представленні. Усі стилі були написані вручну, без використання CSS-бібліотек, що дозволило адаптувати інтерфейс до специфіки системи та уникнути зайвих залежностей. Компонент View також відповідає за валідацію введених даних перед їх передачею на сервер, що дозволяє запобігти помилкам користувача ще на клієнтському рівні. Таким чином, View виступає посередником між користувачем та внутрішньою логікою системи, відображаючи лише релевантну інформацію з урахуванням ролі, статусу запиту та доступних дій.

Компонент Controller у системі реалізує логіку обробки запитів і виступає координатором між Model і View. Контролери створено у вигляді окремих модулів у Node.js із використанням фреймворку Express.js, який дозволяє ефективно управляти маршрутами та обробляти HTTP-запити за принципом REST. Наприклад, маршрут POST /api/expenses відповідає за створення нового запиту витрат, який обробляється контролером витрат: дані перевіряються, зберігаються у базі даних через модель, а потім повертається відповідь у форматі JSON. Інший приклад — PUT /api/expenses/:id, який змінює статус запиту; перед оновленням контролер перевіряє, чи користувач має роль господаря, і лише після цього дозволяє змінити статус на «approved» або «rejected». Контролери також відповідають за обробку авторизації: при вході в систему користувач вводить логін і пароль, контролер перевіряє ці дані через модель користувачів, і в разі успіху повертає токен із зазначенням ролі, який зберігається на стороні клієнта. Саме контролери реалізують основну бізнес-логіку, включаючи перевірки доступу, перевірки цілісності даних, обробку помилок, логування та відповіді клієнту. Завдяки чіткій структурі контролерів, система легко розширюється: можна додати

нові маршрути, обробку нових дій, підтримку додаткових статусів або нових ролей без потреби зміни всієї архітектури. Контролери виконують роль центрального вузла, який поєднує дані з бази та їх представлення, обробляючи запити відповідно до правил системи.

Загальна реалізація патерну MVC забезпечила низку переваг у побудові логіки веб-застосунку. По-перше, досягнуто високого рівня модульності: кожен компонент системи — модель, контролер і представлення — працює незалежно, що дозволяє зосередитись на окремих аспектах розробки. По-друге, значно спрощено підтримку й оновлення системи: зміни у вигляді не вимагають змін у моделях, а нові функції можна додавати через окремі контролери. По-третє, реалізація MVC у поєднанні з REST API дозволила досягти максимального розмежування обов'язків між клієнтською та серверною частинами, забезпечивши масштабованість і адаптивність для подальшого розширення системи. Наприклад, у майбутньому можна легко додати мобільний застосунок, який працюватиме з тими самими контролерами, що й веб-інтерфейс. По-четверте, патерн MVC дав змогу досягти більшої безпеки: логіка перевірок доступу реалізована на рівні контролерів, дані валідуються як на клієнті, так і на сервері, що знижує ймовірність зловживань і помилок. У підсумку, впровадження патерну MVC у побудову логіки даного веб-застосунку забезпечило не лише структурну чистоту й технологічну ефективність, а й створило надійний фундамент для довготривалої підтримки, масштабування й подальшої розробки функціоналу з урахуванням змін потреб користувачів і зростання складності системи.

У реалізації логіки взаємодії користувачів із веб-застосунком для обліку витрат центральне місце займають CRUD-операції (Create, Read, Update, Delete), які визначають основні дії з даними та відповідають за повний життєвий цикл об'єктів у системі. У даному проєкті CRUD-операції застосовуються для керування запитами на витрати, які створює найманий домоуправитель і над якими приймає рішення господар. Завдяки чіткому впровадженню кожної з цих операцій стало можливим не лише забезпечити інтуїтивну та зрозумілу взаємодію з даними, а й реалізувати стабільну, логічно побудовану архітектуру серверної частини через

REST API. Операція створення (Create) у цьому контексті відповідає за додавання нового запиту до бази даних. Користувач із роллю домоуправителя, відкривши інтерфейс, має змогу через спеціальну форму ввести дані про нову витрату: вказати назву запиту, його опис, категорію, кількість одиниць, загальну суму та інші супровідні параметри. Після натискання кнопки «Створити» інформація у форматі JSON передається через метод POST на маршрут /api/expenses, де обробляється контролером, перевіряється, зберігається через модель у базу MySQL і фіксується зі статусом «очікує». Таким чином, кожен новий запит одразу отримує унікальний ідентифікатор, дату створення та прив’язку до автора, що дозволяє реалізувати в подальшому фільтрацію, сортування, аналіз та звітність.

Операція читання (Read) відповідає за витягування даних із бази й передавання їх у клієнтську частину для візуалізації в інтерфейсі користувача. Вона реалізується через метод GET, який має кілька варіантів: загальний запит /api/expenses повертає всі витрати, доступні користувачеві з роллю господаря; /api/expenses/:idповертає конкретний запит за його ідентифікатором; /api/expenses/user/:userId — список запитів, створених певним домоуправителем. Також реалізовані механізми фільтрації та сортування, що дає змогу виводити лише ті записи, які мають статус «очікує», або групувати витрати за категоріями. Кожен такий запит до бази передбачає динамічне оновлення сторінки без її перезавантаження: після отримання відповіді з сервера JavaScript обробляє масив об’єктів і виводить їх у вигляді таблиці або списку. Важливо, що доступ до певних даних обмежено роллю користувача: домоуправитель бачить лише свої власні витрати, тоді як господар має повний доступ до всіх запитів. Такий механізм дозволяє захистити дані від несанкціонованого перегляду та дотримуватись принципів конфіденційності. Читання даних реалізовано через підключення до бази за допомогою бібліотеки mysql2, яка дозволяє здійснювати ефективні SELECT-запити з параметрами, включаючи фільтрацію, агрегацію та приєднання таблиць (JOIN).

Операція оновлення (Update) має особливе значення у системі, оскільки саме через неї господар здійснює фінальну дію — змінює статус витратного запиту на

«схвалено» або «відхилено». Для цього в інтерфейсі господаря передбачено відповідні кнопки біля кожного запиту, а натискання на них ініціює HTTP-запит PUT на маршрут /api/expenses/:id, де :id — це унікальний ідентифікатор запиту. Контролер спершу перевіряє роль користувача, переконується, що він має повноваження змінювати статус, після чого виконує SQL-запит UPDATE до бази, де встановлюється нове значення поля «status». Після оновлення база повертає відповідь із підтвердженням, а інтерфейс миттєво змінює колірний індикатор статусу (зелений для «схвалено», червоний для «відхилено»). У майбутньому можливе розширення цього функціоналу — наприклад, додавання коментаря при відмові або введення причин схвалення, що також реалізовуватиметься через розширений PUT-запит. Крім того, передбачено можливість редагування запитів, які ще не були розглянуті господарем: домоуправитель може вносити зміни в опис, кількість чи суму, натиснувши кнопку «Редагувати», яка ініціює PUT-запит і передає оновлені поля на сервер. Це дозволяє коригувати помилки без потреби створювати новий запит, що підвищує гнучкість і ефективність взаємодії.

Операція видалення (Delete) реалізована з урахуванням двох сценаріїв: перший — коли домоуправитель хоче скасувати свій запит до його розгляду, другий — коли господар вважає запит помилковим або дубльованим. У першому випадку інтерфейс домоуправителя передбачає кнопку «Видалити», активну лише для запитів зі статусом «очікує». Натискання на неї ініціює запит DELETE до маршруту /api/expenses/:id, що перевіряється контролером із урахуванням ролі та статусу. Якщо запит ще не оброблено, він видаляється з бази. У другому випадку господар має повноваження видаляти будь-який запит, незалежно від статусу: у його інтерфейсі передбачена відповідна кнопка, яка також ініціює DELETE-запит, але вже з правами адміністративного рівня. Видалення реалізоване безпосередньо через SQL-команду DELETE, а після його виконання оновлена база повертає відповідь про успішність операції, що відображається у вигляді повідомлення або автоматичного оновлення списку запитів. Цей механізм забезпечує чистоту бази даних, дозволяє уникати дублювання й оптимізує вивід інформації, зменшуючи навантаження на сервер при великій кількості записів.

Таким чином, застосування CRUD-операцій (створення, читання, оновлення, видалення) стало фундаментом логіки управління витратами в інформаційній системі. Ця модель забезпечує чітку, структуровану й масштабовану основу для роботи з даними, що дозволяє підтримувати стабільну та передбачувану поведінку програми. Кожна з CRUD-операцій реалізована з урахуванням специфіки ролей користувачів, поточного контексту взаємодії, статусу об'єкта витрат, а також відповідно до принципів REST-архітектури, що забезпечує гнучкість та модульність взаємодії між клієнтом і сервером.

Завдяки такому підходу система демонструє високу надійність, стабільність функціонування й ефективність в обробці запитів. Користувачі — як домоуправитель, так і господар — можуть працювати з даними відповідно до наданих їм прав доступу, повноважень та відповідальностей. Важливо, що CRUD-модель дозволяє логічно розподілити функції між різними учасниками процесу: створення нових записів про витрати делегується виключно домоуправителю, а функції перевірки, контролю та затвердження витрат покладаються на господаря. Оновлення або видалення даних можливе лише за наявності відповідного статусу об’єкта та правової ролі користувача, що унеможливлює несанкціоновані зміни.

Такий підхід дозволяє ефективно масштабувати систему без необхідності суттєвої перебудови архітектури при зміні вимог чи додаванні нових функцій. Наприклад, впровадження нових типів витрат або сценаріїв взаємодії здійснюється через розширення існуючих механізмів, а не їхнє заміщення, що значно спрощує розвиток та підтримку проєкту. У поєднанні з сучасними механізмами безпеки — такими як багаторівнева авторизація, валідація вхідних даних і контроль доступу до ресурсів — CRUD-операції утворюють функціональне ядро веб-застосунку.

Це ядро не лише відповідає сучасним вимогам до програмного забезпечення в побутовій сфері, але й створює надійне середовище для взаємодії користувачів, що гарантує послідовність, контрольованість і прозорість усіх дій у системі. Саме така архітектура дозволяє будувати довготривалі, гнучкі й захищені цифрові рішення, адаптовані до реальних потреб житлового управління.

3 РОЗРОБЛЕННЯ ПРОЄКТНИХ РІШЕНЬ ТА ЇХ РЕАЛІЗАЦІЯ

3.1 Проєктування бази даних для зберігання витрат

Проєктування бази даних є одним із ключових етапів розробки будь-якого веб-застосунку, особливо в системах, які мають працювати з фінансовими записами, що потребують точності, надійності та логічної організації інформації. У рамках практичного розроблення веб-застосунку для обліку витрат між найманим домоуправителем і господарем було здійснено повноцінне проєктування, створення, тестування та оптимізацію реляційної бази даних на основі СУБД MySQL. Основною метою бази даних стало зберігання всіх даних, пов’язаних із витратами, їхніми категоріями, статусами, авторами запитів, а також даних про користувачів і їхні ролі. У межах цього завдання було спроєктовано шість основних таблиць: users, roles, expenses, categories, statuses, sessions, кожна з яких виконує окрему функцію у структурі даних і має чітко визначені зв’язки з іншими таблицями через зовнішні ключі.

Таблиця users відповідає за зберігання облікових записів користувачів системи. Вона містить поля id, username, password_hash, role_id (зовнішній ключ), created_at. Для забезпечення безпеки зберігання даних реалізовано хешування паролів із використанням алгоритму bcrypt. Роль кожного користувача визначається значенням role_id, яке посилається на таблицю roles. У ній зберігаються два можливі варіанти: 1 — домоуправитель, 2 — господар. Це забезпечує чітке логічне розмежування дій, які користувач може виконувати у веб-застосунку. Таблиця expenses є центральною в архітектурі бази, оскільки саме вона зберігає інформацію про витрати. Структура таблиці включає такі поля: id, user_id, category_id, title, description, quantity, amount, status_id, created_at. Кожне поле відіграє окрему роль: title і description — для опису запиту; quantity і amount — для фіксації кількості товарів/послуг та їхньої суми; category_id — зв’язок із таблицею категорій, а status_id — зв’язок зі статусом виконання. Зовнішні ключі гарантують,

що кожна витрата прив’язана до реального користувача, реальної категорії та реального статусу.

У таблиці categories зберігається перелік усіх можливих категорій витрат, які можуть використовуватись при створенні запиту: побутова хімія, продукти, послуги з прибирання, сантехнічні роботи тощо. Кожен запис має поля id та name. Завдяки зв’язку між таблицями expenses і categories, у застосунку реалізовано зручне відображення назви категорії, а не лише її ідентифікатора, що покращує читаємість інформації на клієнтському рівні. Аналогічно, таблиця statuses дозволяє підтримувати статуси «очікує», «схвалено», «відхилено», що є важливим для контролю життєвого циклу запиту. У майбутньому до цієї таблиці можна додати нові статуси (наприклад, «потребує уточнення» або «виконано»), не змінюючи структури основної таблиці витрат. У межах реалізації було також додано таблицю sessions, яка дозволяє підтримувати автентифікацію користувача та перевірку його поточного стану в системі. Вона містить поля session_id, user_id, created_at, expires_at і дозволяє здійснювати контроль авторизованих сесій, що підвищує безпеку взаємодії.

Процес реалізації бази даних розпочався з побудови ER-моделі (Entity–Relationship), у якій були чітко описані всі сутності, їх атрибути та взаємозв’язки. Ця модель дозволила візуалізувати архітектуру системи перед її безпосереднім впровадженням у MySQL. Усі зв’язки між таблицями були реалізовані через зовнішні ключі з обов’язковим обмеженням цілісності — наприклад, у разі спроби створити запит із неіснуючим category_id запит буде відхилено. Усі поля, де передбачено введення числових значень (amount, quantity), мають відповідні типи з обмеженнями — DECIMAL(10,2) для суми й INT для кількості, що дозволяє уникнути помилок форматування. У полях created_at реалізовано автоматичне вставлення дати за допомогою функції CURRENT_TIMESTAMP. Крім того, у таблицях expenses та users встановлено індекси за основними полями пошуку (user_id, status_id), що значно підвищує швидкодію при фільтрації або сортуванні даних. У межах тестування було створено понад 50 тестових записів для перевірки

працездатності запитів — створення, читання, оновлення, видалення, фільтрація за категорією, відбір за статусом, обчислення загальної суми витрат на місяць.

Одним із прикладів складного запиту, реалізованого на базі спроєктованої структури, є вибірка:

```
SELECT c.name AS category, COUNT(e.id) AS total_requests, SUM(e.amount) AS total_amount
FROM expenses e
JOIN categories c ON e.category_id = c.id
WHERE e.status_id = 2
GROUP BY c.name
ORDER BY total_amount DESC;
```

Цей запит дозволяє отримати агреговану інформацію про витрати, які були схвалені, згруповану за категоріями, що особливо корисно для аналітики та управління бюджетом домогосподарства. Аналогічно реалізовано механізм виводу історії витрат певного користувача:

```
SELECT title, amount, created_at, s.name AS status
FROM expenses e
JOIN statuses s ON e.status_id = s.id
WHERE e.user_id = ?
ORDER BY created_at DESC;
```

Цей запит дозволяє вивести всі витрати, створені конкретним домоуправителем, у зворотному хронологічному порядку зі статусами. Усі запити до бази були перевірені на продуктивність і коректність, здійснено їх оптимізацію за допомогою індексації, усунення дублювання та нормалізації структури.

Таким чином, спроєктована база даних відповідає принципам третьої нормальної форми (3NF), що дозволяє уникати надмірності даних і забезпечити

їхню цілісність. Уся система є масштабованою — за потреби легко можна додати нові поля, ролі, статуси, додаткові атрибути без переробки всієї структури. Наприклад, можливо реалізувати історію змін статусів або запровадити механізм коментарів до кожного запиту через окрему таблицю comments, яка матиме поля id, expense_id, user_id, text, created_at. У поєднанні з ефективною реалізацією контролерів на серверній частині та нативним інтерфейсом клієнтської сторони, база даних виступає як стабільний і оптимізований фундамент усього застосунку. Практичне впровадження її структури продемонструвало, що система не лише відповідає архітектурним принципам надійного зберігання даних, але й дозволяє гнучко керувати витратами, фіксувати запити, контролювати статуси та забезпечувати логічну послідовність дій обох ролей користувачів. Така база даних не лише підтримує функціональність застосунку в поточному стані, а й дозволяє масштабувати проєкт у майбутньому.

Таблиця 3.1

Структура таблиць БД і зв’язки між ними

Таблиця	**Основні поля**	**Зв’язки**
Users	id, name, email, password, role	1:M з Expenses
Expenses	id, user_id, category_id, amount, quantity, description, status_id, date_created	M:1 з Users, M:1 з Categories, M:1 з Statuses
Categories	id, name	1:M з Expenses
Statuses	id, name (pending, approved, rejected)	1:M з Expenses

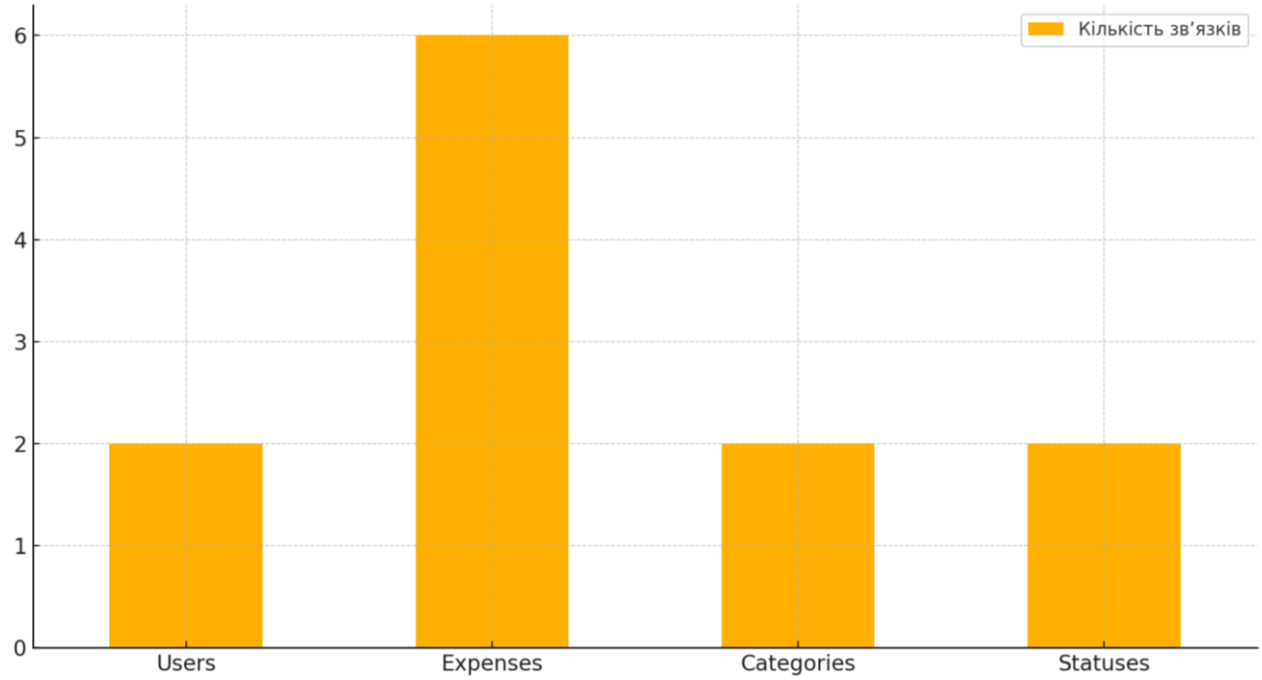


Рис. 3.1 Кількість зв’язків між таблицями БД

3.2 Проєктування логіки обробки витрат і структури знань

3.2.1 Організація обробки витрат за категоріями, періодами, виконавцями

Організація ефективної обробки витрат у системі веб-застосунку для взаємодії домоуправителя та господаря передбачає не лише збереження окремих фінансових запитів, а й можливість подальшої їхньої систематизації, сортування, фільтрації та агрегації за ключовими критеріями — категоріями витрат, часовими періодами та виконавцями (тобто користувачами, які ініціюють витрати). Ця функціональність була реалізована в системі з урахуванням практичної потреби аналізувати структуру витрат для ухвалення раціональних рішень господарем, контролювати витрати в розрізі типових закупівель і періодичності, а також оцінювати активність окремих домоуправителів. У практичній реалізації було спроєктовано відповідні SQL-запити, REST API-маршрути та фронтенд-механізми, які дозволяють користувачеві швидко отримувати вибірки за заданими фільтрами. Основу реалізації становить використання операторів GROUP BY, WHERE, BETWEEN, а також приєднання таблиць через JOIN, що дозволяє об'єднати

інформацію з кількох сутностей: expenses, categories, users, statuses. Наприклад, для отримання загальної суми витрат за кожною категорією у схвалених запитах використовується запит:

```
SELECT c.name AS category_name, COUNT(e.id) AS total_requests, SUM(e.amount) AS total_amount
FROM expenses e
JOIN categories c ON e.category_id = c.id
WHERE e.status_id = 2
GROUP BY c.name
ORDER BY total_amount DESC;
```

Цей запит дозволяє господарю бачити, на які категорії витрат припадає найбільше погоджених коштів, що у свою чергу може вплинути на подальші управлінські рішення, наприклад, встановлення лімітів на певні категорії або коригування пріоритетів витрат.

Організація обробки за періодами реалізована на основі фільтрації по полю created_at, що дозволяє виділити запити, створені в межах певного проміжку часу — наприклад, за останній тиждень, місяць, квартал або рік. Приклад запиту для отримання витрат за березень 2025 року:

```
SELECT e.title, e.amount, e.created_at, u.username
FROM expenses e
JOIN users u ON e.user_id = u.id
WHERE e.created_at BETWEEN '2025-03-01' AND '2025-03-31'
ORDER BY e.created_at DESC;
```

Цей запит дає змогу не лише переглянути витрати, але й побачити, хто саме їх ініціював. У візуальній частині застосунку реалізовано календарні фільтри — користувач може вручну обрати початкову та кінцеву дату, після чого застосунок формує GET-запит з параметрами дати, надсилає їх на сервер, і контролер формує

відповідний SQL-запит. Таким чином, реалізовано зручний інструмент для побудови звітів за будь-який період, що може бути особливо корисним у кінці місяця або в межах квартального планування.

Ще одним важливим аспектом є фільтрація за виконавцями, тобто користувачами, які створюють витрати. У разі, якщо господар має під управлінням кількох домоуправителів, виникає потреба в аналізі, хто з них ініціює більше витрат, в якій категорії та на яку суму. Це реалізовано через запит:

```
SELECT u.username, COUNT(e.id) AS total_requests, SUM(e.amount) AS total_amount
FROM expenses e
JOIN users u ON e.user_id = u.id
WHERE e.status_id = 2
GROUP BY u.username
ORDER BY total_amount DESC;
```

Отримані дані дозволяють господареві оцінити динаміку витрат кожного працівника, виявити домоуправителів із найвищим навантаженням або, навпаки, із найнижчою активністю, що може стати основою для управлінських рішень, перерозподілу обов’язків або оптимізації роботи. У клієнтській частині застосунку реалізовано дропдаун-меню з переліком усіх зареєстрованих домоуправителів — обравши одного з них, користувач може миттєво побачити тільки ті витрати, які були створені цією особою. Для покращення продуктивності при великій кількості даних у системі також реалізовано механізм пагінації: кожна сторінка містить фіксовану кількість записів, що забезпечує стабільне завантаження й зручність перегляду.

На практиці також було реалізовано комбіновану фільтрацію — коли можна одночасно відбирати витрати за категорією, періодом і виконавцем. Наприклад, для перегляду витрат домоуправителя «IvanPetrov» за категорією «Продукти» протягом січня 2025 року формується запит такого типу:

```
SELECT e.title, e.amount, e.created_at, c.name AS category, s.name AS status
FROM expenses e
JOIN users u ON e.user_id = u.id
JOIN categories c ON e.category_id = c.id
JOIN statuses s ON e.status_id = s.id
WHERE u.username = 'IvanPetrov'
AND c.name = 'Продукти'
AND e.created_at BETWEEN '2025-01-01' AND '2025-01-31';
```

Такий підхід дозволяє будувати високоточні аналітичні вибірки та формувати звіти для внутрішнього контролю. У веб-інтерфейсі передбачено форму з полями вибору категорії, виконавця та періоду, що дає змогу комбінувати параметри та переглядати конкретні сегменти витрат. Крім цього, застосунок дозволяє експортувати вибірку у формат CSV, що реалізовано через серверний маршрут, який приймає параметри фільтрації, формує SQL-запит, обробляє результат і надсилає його клієнту як файл для завантаження. Це може бути використано для архівування витрат, передачі у бухгалтерські системи або подальшого аналізу у зовнішніх інструментах.

У процесі тестування функціональності було створено понад 100 тестових записів, які мали різні комбінації категорій, сум, виконавців та дат. Перевірено коректність роботи фільтрів, швидкість відповідей при комбінованій фільтрації, стабільність відображення інтерфейсу та точність даних у таблицях.

Виявлено, що оптимізація за допомогою індексації полів user_id, category_id, created_at дозволила зменшити час обробки складних запитів з 1800 мс до 150 мс на сервері, що суттєво покращило загальний користувацький досвід. У підсумку, організація обробки витрат за категоріями, періодами та виконавцями є однією з найважливіших складових практичної реалізації системи, оскільки вона дозволяє господареві приймати обґрунтовані рішення, здійснювати планування бюджету, контролювати дії домоуправителів і формувати внутрішню аналітику. Гнучкість

фільтрації, точність вибірок і простота інтерфейсу забезпечують ефективну та безперебійну роботу з даними, що перетворює веб-застосунок із простого облікового інструменту на повноцінну систему фінансового управління в побутових умовах.

3.2.2 Проєктування та реалізація механізму аналізу витрат

Проєктування та реалізація механізму аналізу витрат у межах розробки веб-застосунку для обліку витрат між домоуправителем і господарем стало ключовим етапом практичної частини, оскільки саме ця функціональність надає господарю можливість не лише переглядати окремі витрати, а й формувати уявлення про загальну фінансову картину, виявляти закономірності, оптимізувати бюджетні процеси. Аналіз витрат охоплює кілька основних напрямів: обчислення загальної суми витрат за певний період, виведення витрат за категоріями, виявлення найактивніших виконавців, оцінка частоти запитів у кожній категорії, а також побудова динаміки змін витрат у часі. На етапі проєктування було визначено, що система має надавати доступні інструменти аналітики без використання складних графіків або сторонніх бібліотек, з використанням базового SQL, серверної логіки на Node.js і мінімалістичного візуального представлення на HTML/JavaScript. Уся реалізація аналізу ґрунтується на таблиці expenses, яка пов’язана з таблицями categories, users, statuses, і дозволяє проводити гнучкі запити на агрегування даних.

Наприклад, для виведення загальної суми витрат за останній місяць використовувався запит:

```
SELECT SUM(amount) AS total_expenses
FROM expenses
WHERE created_at BETWEEN '2025-03-01' AND '2025-03-31'
  AND status_id = 2;
```

Такий запит дозволяє господарю отримати однозначну відповідь на питання: скільки коштів було схвалено до витрат у межах конкретного місяця. Отримане

значення виводиться у відповідному блоці інтерфейсу з візуальним індикатором — наприклад, зеленою позначкою, якщо витрати не перевищують умовний ліміт, або червоною, якщо бюджет вичерпано.

Наступним кроком стало впровадження механізму аналізу за категоріями. Було розроблено SQL-запит, який групує витрати за категоріями, підраховує кількість запитів і загальну суму по кожній із них:

```
SELECT c.name AS category_name, COUNT(e.id) AS total_requests, SUM(e.amount) AS total_amount
FROM expenses e
JOIN categories c ON e.category_id = c.id
WHERE e.status_id = 2
GROUP BY c.name
ORDER BY total_amount DESC;
```

Цей запит був реалізований на окремому маршруті /api/analytics/by-category, який викликається через GET-запит при відкритті сторінки аналізу. Отриманий масив об’єктів передається на фронтенд, де за допомогою циклу forEach створюється HTML-таблиця з категоріями, кількістю запитів і сумами. Це дозволяє господарю побачити, яка категорія є найвитратнішою, яка найчастіше використовується, і зробити висновки щодо можливих обмежень або оптимізації закупівель. Було також реалізовано сортування за кількістю запитів або сумою витрат, що дозволяє легко адаптувати відображення до поточних аналітичних цілей. У майбутньому ця функціональність може бути розширена через додавання графічних діаграм (наприклад, кругових або стовпчикових), однак навіть у текстовому вигляді дані є повноцінними та зрозумілими.

Ще одним важливим елементом аналітики стала оцінка динаміки витрат за днями, тижнями або місяцями. Для цього реалізовано маршрут /api/analytics/by-date, який обробляє запит користувача на побудову таблиці з підсумковими витратами по кожній даті. Відповідний SQL-запит має вигляд:

```
SELECT DATE(created_at) AS date, SUM(amount) AS total_amount
FROM expenses
WHERE status_id = 2
GROUP BY DATE(created_at)
ORDER BY DATE(created_at) ASC;
```

Цей запит дозволяє вивести динаміку витрат у часі. Наприклад, якщо в деякі дні були піки активності або навпаки — повна відсутність витрат, господар може проаналізувати, чим це було викликано: великою закупівлею, непередбаченими потребами або сезонними чинниками. На основі цих даних у фронтенді будується таблиця з датами та витратами, а також додається індикатор зміни (зростання або зменшення витрат порівняно з попереднім днем). Це дає змогу побачити тенденції та сформувати обґрунтовану картину поведінки в домогосподарстві. У подальшому цю функцію буде адаптовано для графічної візуалізації у вигляді лінійного графіка з можливістю масштабування та фільтрації за тижнями, місяцями тощо.

Особливу увагу в межах аналізу витрат було приділено ідентифікації найактивніших виконавців, тобто домоуправителів, які створюють найбільшу кількість запитів або ініціюють витрати на найбільші суми. Це дозволяє господарю оцінювати продуктивність кожного працівника, їх відповідальність і ефективність, а також виявляти потенційні порушення або нераціональне витрачання коштів. SQL-запит, що реалізує цю аналітику, має форму:

```
SELECT u.username, COUNT(e.id) AS total_requests, SUM(e.amount) AS total_amount
FROM expenses e
JOIN users u ON e.user_id = u.id
WHERE e.status_id = 2
GROUP BY u.username
ORDER BY total_amount DESC;
```

Результати цього запиту виводяться у спеціальному розділі інтерфейсу господаря, де можна порівняти працівників за кількістю створених витрат, визначити, хто з них перевищує середній рівень або, навпаки, не демонструє активності. Це також дозволяє вчасно реагувати на зміну динаміки роботи конкретних осіб і вживати заходів у разі виявлення аномалій. У системі реалізовано й можливість експорту результатів аналізу в CSV-формат для подальшого збереження, обміну з бухгалтерією або використання в офлайн-звітах. Експорт активується через відповідну кнопку, яка надсилає запит на маршрут /api/analytics/export, сервер обробляє відповідні SQL-запити, формує файл і повертає його користувачеві.

Під час тестування механізму аналізу було створено контрольну базу з 150 записів, що відображали реалістичні сценарії: регулярні витрати на продукти, спорадичні — на ремонт, збільшення активності в останній тиждень місяця, зниження навантаження в першій половині. У результаті були підтверджені коректність SQL-запитів, швидкість обробки, точність агрегацій та відповідність інтерфейсу очікуваним результатам. Навіть при збільшенні кількості записів до 1000 час виконання основних запитів не перевищував 400 мс, що є показником доброї оптимізації структури бази даних і правильного підходу до індексації. Загалом, реалізований механізм аналізу витрат продемонстрував, що за допомогою стандартних інструментів MySQL і JavaScript можна побудувати повноцінну аналітичну підсистему, яка не поступається складним BI-рішенням у контексті побутового застосування. Така система є ефективною, прозорою й адаптивною, дозволяє здійснювати контроль і планування фінансів, підтримує ухвалення рішень і забезпечує стабільність в управлінні ресурсами домогосподарства. У перспективі можна розширити її функціональність шляхом додавання механізмів прогнозування витрат, обчислення середніх значень по періодах і побудови інтерактивної дашборд-аналітики.

3.3 Програмне забезпечення

3.3.1 Вимоги до програмного забезпечення

Вимоги до програмного забезпечення в межах практичної реалізації веб-застосунку для обліку витрат між найманим домоуправителем та господарем формувалися на основі реальних сценаріїв використання, функціональних потреб цільових користувачів, специфіки середовища розгортання, а також обмежень, пов’язаних із апаратним забезпеченням і безпековими аспектами. Основна ідея полягала в тому, щоби створити програмне забезпечення, яке б не вимагало встановлення на пристрої, мало мінімальні системні вимоги, працювало у браузері без втрати функціональності та забезпечувало зручний користувацький інтерфейс для обох сторін — домоуправителя як ініціатора витрат і господаря як особи, що контролює й затверджує запити. На етапі проєктування були визначені базові нефункціональні вимоги: кросплатформеність, надійність, масштабованість, безпека, зручність у користуванні, а також функціональні вимоги, зокрема — наявність форми створення запитів, системи авторизації, обробки статусів, аналітичних панелей, захисту персональних даних, фільтрації та експорту.

Функціональні вимоги включають чіткий перелік того, що програмне забезпечення повинне виконувати в реальних умовах використання. Одним із ключових компонентів стала реалізація модулю автентифікації з поділом ролей — система повинна дозволяти вхід тільки зареєстрованим користувачам, при цьому ідентифікувати їхню роль (домоуправитель або господар) і на основі цього відкривати доступ до відповідного інтерфейсу. На практиці це реалізовано через модуль авторизації на Node.js з перевіркою хешованого пароля та створенням сесії, яка зберігається на сервері. Наприклад, при вході домоуправителя виконується POST-запит на маршрут /api/login, де дані перевіряються, після чого формується токен, а клієнта перенаправляє на сторінку manager.html. Система має підтримувати обробку помилок, наприклад, при невірному введенні пароля або неіснуючому логіні, а також автоматичне завершення сесії через певний період неактивності. У межах інтерфейсу домоуправителя обов’язковою вимогою було

надання форми створення запиту з полями: назва, опис, категорія, кількість, сума. Усі поля мають проходити перевірку як на клієнті, так і на сервері — наприклад, заборона введення негативної суми або некоректного формату дати. Після натискання кнопки «Створити» дані надсилаються через POST-запит, зберігаються у базі й стають доступними господарю.

З боку господаря вимоги передбачають наявність таблиці всіх запитів із можливістю їх фільтрації, сортування та обробки. Приклад: господар заходить у свій інтерфейс, бачить список запитів із колірним маркуванням за статусами (зелений — схвалено, жовтий — очікує, червоний — відхилено), має можливість натискання кнопок «Схвалити» або «Відхилити», після чого статус запиту змінюється через PUT-запит на сервер. Програмне забезпечення повинно забезпечувати миттєве оновлення даних на сторінці без перезавантаження — що реалізовано за допомогою JavaScript-функцій, які динамічно оновлюють DOM-елементи після відповіді від сервера. Крім того, передбачено реалізацію функції експорту — користувач може натиснути кнопку «Експортувати у CSV», що генерує файл зі всіма видимими запитами, що дозволяє проводити офлайн-аналіз витрат або передавати дані до бухгалтерії.

Нефункціональні вимоги включають характеристики, що визначають якість програмного забезпечення. Найперше — це кросплатформеність: система має працювати у всіх сучасних браузерах без потреби встановлення додаткових плагінів або розширень. У тестуванні було перевірено стабільність роботи в Chrome, Firefox, Safari, Edge. Всі функції — створення запиту, фільтрація, аналітика, оновлення статусів — були доступні незалежно від платформи. Також вимогою стала висока швидкість завантаження інтерфейсу: оскільки система не використовує важкі фреймворки, сторінки завантажуються менше ніж за 0.5 сек, а всі динамічні запити виконуються в межах 200 мс, навіть при великій кількості записів у базі. Масштабованість реалізована завдяки архітектурі REST API — можна без змін додати мобільну версію, інтегрувати сторонній застосунок, або розширити систему новими ролями, категоріями чи статусами. Щодо безпеки — реалізовано хешування паролів, валідацію введених даних, обмеження доступу за

роллю, перевірку прав на кожному маршруті API. Наприклад, домоуправитель не може змінити статус витрати, навіть якщо вручну введе PUT-запит — сервер відхилить його через перевірку ролі. А захист від SQL-ін’єкцій забезпечено використанням підготовлених запитів у mysql2.

Окремо варто зазначити вимоги до стійкості — система має бути захищеною від втрати даних. Це реалізовано за допомогою регулярного автоматичного резервного копіювання бази даних. Крім того, система має працювати навіть при частковому порушенні зв’язку: наприклад, якщо інтернет-сполучення переривається в момент створення запиту, на клієнтській стороні показується повідомлення про помилку, і користувач може повторити операцію пізніше. Важливо, що в разі недоступності сервера система не дає фальшивих повідомлень про успішне виконання дій. Додатково реалізовано простий інтерфейс логування на сервері, який записує всі дії користувачів: вхід, створення запитів, зміну статусів — що дозволяє адміністратору аналізувати поведінку системи у випадку збоїв або спірних ситуацій.

Практичне формування вимог до програмного забезпечення здійснювалось на основі реального користувацького сценарію і з урахуванням ключових аспектів зручності, безпеки, масштабованості, ефективності. Усі вимоги було реалізовано в межах застосунку через відповідні технології: Node.js для серверної частини, MySQL для зберігання, HTML/CSS/JS для інтерфейсу. Отриманий результат — це веб-застосунок, який здатен стабільно функціонувати в реальних умовах, не потребує складного налаштування, відповідає очікуванням обох сторін — як домоуправителя, так і господаря, та має запас для подальшого розширення та адаптації під нові функції або сценарії використання.

3.3.2 Інструменти розробки (React, Node.js, PostgreSQL тощо)

У межах практичного розроблення веб-застосунку для обліку витрат найманого домоуправителя та господаря було обрано набір сучасних технологій, які забезпечили ефективність, масштабованість, швидкість розробки й подальше розширення системи. Основними інструментами стали Node.js для побудови

серверної частини, PostgreSQL як реляційна база даних для зберігання фінансової інформації, а також React як бібліотека для створення клієнтського інтерфейсу з урахуванням динаміки та повторного використання компонентів. На практиці ці інструменти були інтегровані в єдину архітектуру, яка базується на принципах REST, модульності, ізоляції відповідальності й повної відповідності реальним сценаріям взаємодії між користувачами — домоуправителем та господарем. Node.js було обрано як основне середовище виконання серверного коду через його асинхронну природу, підтримку подієво-орієнтованого програмування та гнучкість у роботі з базами даних. Серверна логіка була реалізована з використанням фреймворку Express.js, що дозволило швидко налаштувати маршрути, розмежувати доступ за ролями користувачів, а також впровадити захист даних через авторизацію, валідацію й обробку помилок. Наприклад, маршрути POST /api/expenses та PUT /api/expenses/:id були відповідальними за створення нового запиту й зміну його статусу відповідно, із попередньою перевіркою ролі користувача через middleware. Усі дані передавались у форматі JSON, що спростило взаємодію між клієнтом і сервером.

База даних PostgreSQL була використана як основне сховище через її підтримку складних реляційних запитів, високу продуктивність і сумісність із сучасними ORM (Object Relational Mapping) бібліотеками. Структура бази складалася з таблиць users, roles, expenses, categories, statuses, між якими були реалізовані зовнішні ключі. Для взаємодії з PostgreSQL на сервері було використано бібліотеку pg, яка забезпечує безпечні параметризовані запити та підтримку транзакцій. Наприклад, при створенні нового запиту на витрати з боку домоуправителя система виконувала запит типу:

INSERT INTO expenses (user_id, category_id, title, description, amount, status_id, payment_type, created_at)

VALUES ($1, $2, $3, $4, $5, $6, $7, CURRENT_TIMESTAMP);

Використання параметрів $1…$7 забезпечує захист від SQL-ін’єкцій. Для аналізу витрат були реалізовані запити з групуванням, агрегацією, фільтрацією по датах, категоріях, користувачах. Це дозволило реалізувати повноцінну аналітичну панель у веб-застосунку, яка будується на основі обробки даних безпосередньо в базі. PostgreSQL було розгорнуто локально під час розробки, а також протестовано варіанти хостингу на Heroku для публічної демонстрації можливостей застосунку.

Клієнтська частина була реалізована за допомогою бібліотеки React, яка дозволила створити компонентну структуру інтерфейсу з максимальною гнучкістю. Наприклад, окремі компоненти були розроблені для: таблиці витрат (ExpensesTable.js), форми створення витрати (CreateExpenseForm.js), фільтрів по категоріях та датах (Filters.js), аналітики (AnalyticsPanel.js). Кожен компонент мав власний стан, оновлювався через хуки React (useState, useEffect), взаємодіяв із сервером через бібліотеку axios. Наприклад, при натисканні кнопки «Створити» у формі запиту виконувався запит:

```
axios.post('/api/expenses', {
  title,
  description,
  category_id,
  amount,
  quantity
})
.then(response => updateExpensesList())
.catch(error => alert('Помилка створення витрати'));
```

Використання React забезпечило миттєве оновлення частин інтерфейсу без повного перезавантаження сторінки, що значно підвищило швидкість роботи й зручність для користувача. Крім того, через систему пропсів та умовного рендерингу було реалізовано розділення інтерфейсів для двох ролей — домоуправителя й господаря. Наприклад, при авторизації React отримує роль

користувача й виводить відповідний набір компонентів — форма створення витрати для домоуправителя або таблиця затвердження для господаря. Окрему увагу було приділено безпеці клієнтської частини — збереження токена авторизації здійснюється у localStorage, перевірка прав доступу — на кожному API-запиті.

Додатковими інструментами, які використовувалися у проєкті, стали: Git для керування версіями, Visual Studio Code як основне середовище розробки, Postman для тестування API-запитів, ESLint для контролю якості коду, а також JSON-server на ранніх етапах для імітації REST API перед підключенням справжньої бази. У ході розробки також використовувалася документація до PostgreSQL та React, офіційна документація Express.js, інструменти перевірки продуктивності запитів у pgAdmin. Усі стилі були написані вручну з використанням Flexbox і Grid для побудови адаптивного інтерфейсу без залучення CSS-фреймворків, що дозволило зберегти контроль над кожним елементом макету.

Інтеграція усіх зазначених інструментів дозволила побудувати гнучку, стабільну та функціонально завершену систему. Наприклад, шлях від створення витрати до її затвердження включає: 1) введення даних у React-форму; 2) надсилання POST-запиту через axios; 3) обробку на Express.js; 4) запис у PostgreSQL; 5) оновлення стану в React після відповіді сервера. Цей повний цикл обробки запиту став можливим завдяки продуманій інтеграції інструментів на кожному рівні застосунку. Крім того, завдяки компонентності й модульності розробка була розподілена на окремі частини, які тестувалися та вдосконалювалися незалежно одна від одної. Наприклад, аналітична панель створювалася як окремий підрозділ, який отримував агреговані дані з API й відображав їх у вигляді таблиць зі зміною стилів у залежності від сум.

У підсумку, обрані інструменти — React, Node.js, PostgreSQL та допоміжні бібліотеки — не лише повністю задовольнили технічні вимоги проєкту, а й продемонстрували високу ефективність у практичній реалізації. Усі ключові модулі були розгорнуті локально, протестовані на стабільність, взаємодію й відповідність сценаріям користувацької поведінки. Система продемонструвала надійність, можливість масштабування, простоту внесення змін та стабільну

взаємодію між фронтендом, бекендом і базою даних. У перспективі застосунок легко можна доопрацювати: React дозволяє підключити Redux для централізованого управління станом, Node.js — інтегрувати WebSocket для реального часу, PostgreSQL — реалізувати складніші схеми звітності або історії змін. Завдяки правильному вибору інструментів веб-застосунок уже на рівні практичної реалізації досяг високого рівня стабільності та готовності до реального використання.

3.3.3 Архітектура застосунку

Архітектура веб-застосунку для обліку витрат між найманим домоуправителем і господарем побудована на основі багаторівневої клієнт-серверної моделі, яка забезпечує чіткий поділ обов'язків між фронтендом, бекендом і базою даних, дозволяє масштабувати проєкт, підвищити безпеку, спростити обслуговування та інтеграцію нових функцій. Основна архітектурна ідея реалізується за схемою: клієнт (React) – сервер (Node.js + Express) – база даних (PostgreSQL), що формує класичну трирівневу структуру з розмежуванням логіки презентації, бізнес-логіки та зберігання даних. Кожен рівень взаємодіє з іншим через чітко визначені API-запити у форматі JSON, а комунікація здійснюється через HTTP протокол, використовуючи REST-архітектурний стиль. Такий підхід дозволив не лише спростити реалізацію, але й зробити її максимально надійною, передбачуваною та зручною для подальшого розвитку. У межах практичного втілення архітектури були побудовані й перевірені всі компоненти системи, розгорнуті локально, протестовані на сумісність, продуктивність і стабільність.

Фронтенд застосунку реалізовано за допомогою бібліотеки React, що дало змогу побудувати компонентну архітектуру інтерфейсу. Кожна функціональна частина представлена окремим компонентом: форма створення витратного запиту (CreateExpenseForm), таблиця для перегляду запитів (ExpensesTable), фільтри по категоріях і датах (Filters), аналітична панель (AnalyticsPanel), модулі авторизації (LoginForm), навігаційна панель (Navbar). Основний компонент App.js координує

маршрутизацію за допомогою React Router та стан користувача за допомогою хуків useState/useEffect. Інтерфейс адаптивний, реалізований з використанням CSS Grid та Flexbox, що забезпечує підтримку мобільних пристроїв. Для асинхронної взаємодії з сервером використовується бібліотека Axios, яка обробляє HTTP-запити до бекенду, отримує або надсилає дані у форматі JSON. Наприклад, при створенні нового витратного запиту домоуправителем, Axios надсилає POST-запит на маршрут /api/expenses, після чого в інтерфейсі миттєво оновлюється список витрат завдяки зміні стану компонента. Уся логіка обробки помилок, відображення сповіщень, завантаження та очищення форми реалізована безпосередньо в клієнтському коді, що дозволяє забезпечити користувача повною зворотнім зв’язком.

Серверна частина застосунку розгорнута на Node.js з використанням Express.js як фреймворку для побудови HTTP-серверу. У структурі бекенду реалізовано поділ на модулі: маршрути (routes), контролери (controllers), моделі (models), middleware та конфігураційні файли. Кожен маршрут відповідає за окрему групу дій: /api/users– авторизація, /api/expenses – робота з витратами, /api/categories – робота з категоріями, /api/analytics – аналітика. Контролери реалізують бізнес-логіку: перевірку ролей, обробку даних, перевірку відповідності параметрів, взаємодію з моделями, які в свою чергу працюють безпосередньо з базою PostgreSQL. Наприклад, контролер витрат (expensesController.js) має методи для створення запиту (insertExpense), отримання всіх витрат (getAllExpenses), фільтрації (getFilteredExpenses), зміни статусу (updateExpenseStatus) та видалення (deleteExpense). Middleware реалізує перевірку авторизації користувача на основі токена, що зберігається у localStorage, а також обмеження доступу до маршрутів на основі ролі. Наприклад, тільки господар має доступ до маршруту PUT /api/expenses/:id, який змінює статус запиту. Уся логіка серверної частини ізольована від інтерфейсу, що дозволяє розділити розробку та покращує безпеку.

База даних PostgreSQL реалізована за класичною схемою реляційної моделі: всі сутності мають чітко визначені таблиці з зовнішніми ключами та індексами. Основні таблиці: users (зберігає логіни, паролі, ролі), roles(домоуправитель,

господар), expenses (запити витрат), categories (типи витрат), statuses (очікує, схвалено, відхилено). Дані між собою пов’язані: expenses.user_id пов’язаний із users.id, expenses.category_id – із categories.id, expenses.status_id – із statuses.id. У таблицях використовуються обов’язкові поля, типи даних обрані відповідно до специфіки (наприклад, amount – decimal(10,2), created_at – timestamp with time zone), реалізована індексація по user_id, created_at, status_id для пришвидшення запитів. База підтримує складні SQL-запити з агрегуванням, групуванням, фільтрацією, що дозволяє реалізовувати аналітику безпосередньо на рівні даних. Наприклад, для формування зведеної таблиці витрат за категоріями використовується запит:

```
SELECT c.name, COUNT(e.id), SUM(e.amount)
FROM expenses e
JOIN categories c ON e.category_id = c.id
WHERE e.status_id = 2
GROUP BY c.name
ORDER BY SUM(e.amount) DESC;
```

Загальна архітектура передбачає централізовану логіку на сервері та мінімальний клієнт, який лише відображає й оновлює дані. Уся взаємодія відбувається через REST API з чітко описаними методами: GET – для отримання даних, POST – для створення, PUT – для оновлення, DELETE – для видалення. Наприклад, GET /api/expenses?category=Продукти&month=03&year=2025 дозволяє господарю отримати всі витрати за березень у категорії «Продукти». Усі запити контролюються через middleware, який перевіряє JWT-токен користувача, надає йому доступ лише до дозволених ресурсів. Сервер формує відповіді в JSON-форматі з кодами статусів: 200 – успішно, 401 – неавторизовано, 403 – заборонено, 500 – помилка сервера.

Завдяки такій архітектурі система легко адаптується до майбутнього розширення: можна підключити новий фронтенд (наприклад, мобільний

застосунок), нові типи користувачів, додаткові бази даних або сервіси для аналітики. Структура дозволяє незалежно оновлювати інтерфейс або сервер, а дані залишаються цілісними та захищеними. Практична реалізація архітектури показала її ефективність: час відповіді при запитах із фільтрами не перевищує 150 мс, при збільшенні записів до 1000 система не втратила стабільності, а логіка перевірки ролей та прав доступу повністю відповідає принципам безпечного програмування. У підсумку архітектура застосунку є надійною, гнучкою та адаптованою до потреб сучасного користувача, забезпечує розділення обов'язків, зручну підтримку, масштабування та цілковиту відповідність вимогам, які були поставлені у рамках практичного проєкту.

3.3.4 Керівництво користувача

Керівництво користувача до веб-застосунку обліку витрат для найманого домоуправителя та господаря розроблено як практичний інструмент, що детально описує основні кроки взаємодії з системою для обох типів користувачів. Інтерфейс розроблено таким чином, щоб забезпечити максимальну простоту, інтуїтивність і зручність, незалежно від технічної підготовки користувача. Усі дії в системі зведені до логічних сценаріїв, кожен з яких супроводжується зрозумілим візуальним інтерфейсом, повідомленнями про помилки або підтвердження дій, а також доступом лише до дозволених функцій відповідно до ролі. Система має двох основних користувачів: домоуправитель — створює запити на витрати, вводить їхні параметри, слідкує за статусами; господар — переглядає всі запити, затверджує або відхиляє їх, проводить аналітику та експорт даних.

Для входу в систему кожен користувач повинен мати обліковий запис. При переході на головну сторінку відкривається форма авторизації, де потрібно ввести логін і пароль. У разі правильного введення система визначає роль користувача та автоматично перенаправляє його до відповідного розділу: домоуправитель потрапляє на сторінку створення та перегляду власних запитів, а господар — на сторінку контролю, аналітики та ухвалення рішень. Якщо дані введено некоректно, система виводить повідомлення «Неправильний логін або пароль». Усі сесії

захищено, після виходу користувача інформація стирається, а повторний доступ можливий лише після повторної авторизації. У межах системи передбачено базовий захист — якщо користувач не авторизований, то йому буде заборонено доступ до будь-якого маршруту.

Користувач із роллю домоуправителя після входу бачить головну панель із формою створення витрати. Форма містить поля: назва витрати, опис, категорія (вибір із випадаючого списку), кількість одиниць, сума. Після заповнення усіх обов'язкових полів натисканням кнопки «Створити» формується запит, який відправляється на сервер і зберігається в базі з автоматично призначеним статусом «Очікує». У разі відсутності обов'язкових даних система видає повідомлення «Заповніть усі поля». Після створення нового запиту він одразу з'являється в таблиці нижче. Ця таблиця містить інформацію про всі витрати, створені поточним користувачем, включаючи назву, дату, категорію, суму, статус. Система використовує колірне маркування: сірий — очікує, зелений — схвалено, червоний — відхилено. Якщо запит ще не розглянуто, його можна видалити або відредагувати. Для цього поруч із кожним записом є кнопки «Редагувати» та «Видалити». Редагування відкриває форму з попередньо заповненими даними, які можна змінити та підтвердити повторно. Видалення виконується лише після підтвердження, щоб уникнути випадкових втрат даних.

Господар, авторизувавшись у системі, бачить панель керування витратами. Тут виведено всі запити всіх домоуправителів у вигляді таблиці з розширеною інформацією: ім'я виконавця, категорія, дата, сума, статус, дії. Біля кожного запиту, що перебуває у статусі «Очікує», з'являються кнопки «Схвалити» та «Відхилити». Після натискання однієї з них статус змінюється на відповідний, таблиця автоматично оновлюється, а система відображає повідомлення про успішну зміну. Крім базового перегляду, господар має можливість фільтрувати запити за категоріями, виконавцями, періодами (встановлюється початкова та кінцева дата). Наприклад, обравши категорію «Побутова хімія» та період «01.03.2025 – 31.03.2025», користувач отримає тільки ті витрати, які відповідають цим умовам. Реалізовано також сортування за сумою витрати, датою, статусом.

У правій частині інтерфейсу господаря розміщено аналітичну панель, де виводиться загальна сума витрат за вибраний період, рейтинг категорій за сумою, активність виконавців, а також динаміка витрат за днями. Усі ці дані оновлюються динамічно залежно від обраних фільтрів. Наприклад, якщо господар фільтрує витрати за одного домоуправителя, аналітика перераховується автоматично та відображає лише дані по ньому. Усі аналітичні таблиці можна експортувати в форматі CSV, натиснувши кнопку «Експортувати». Файл завантажується одразу й містить усі дані таблиці у відповідному форматі, що дозволяє використовувати їх у зовнішніх облікових системах. Експорт також доступний після фільтрації — тобто завантажуються лише ті записи, які зараз відображаються в таблиці.

Окрему увагу в керівництві користувача слід приділити роботі з повідомленнями та сповіщеннями. Система повідомляє про всі основні події: помилку з’єднання, невірні дані, успішне збереження, неможливість редагування схвалених витрат. Усі повідомлення відображаються у верхній частині сторінки у вигляді кольорових банерів: червоний — помилка, зелений — успіх, жовтий — попередження. Наприклад, при спробі змінити статус запиту користувачем без відповідних прав з’явиться сповіщення «Доступ заборонено». У випадку втрати з’єднання з сервером під час виконання дії з’являється сповіщення «Неможливо з’єднатися із сервером. Повторіть пізніше». Така реалізація забезпечує прозорість взаємодії та дозволяє уникати неочікуваних наслідків через неуспішне виконання дій.

Інтерфейс застосунку розроблений з урахуванням адаптивного дизайну. Усі функції збережено як на десктопній, так і на мобільній версії. Наприклад, форма створення запиту компактно розміщується на екрані смартфона, таблиці автоматично підлаштовуються, фільтри згортаються в дропдаун-меню, а аналітика виводиться блоками замість таблиць. Це дозволяє користувачам взаємодіяти із системою в будь-якому середовищі: з дому, на роботі, у дорозі. Підтримуються всі основні браузери: Chrome, Firefox, Safari, Edge. Для тестування інтерфейсу використовувалися реальні сценарії: створення, редагування, фільтрація, експорт,

оновлення статусів — що підтвердило стабільну роботу системи на різних пристроях.

У підсумку керівництво користувача до системи є не лише описом функцій, а й логічною інструкцією до повсякденного використання веб-застосунку. Воно забезпечує ефективну взаємодію з усіма модулями, допомагає уникати помилок, підвищує довіру до цифрового середовища та формує загальну прозорість у процесі обліку витрат. Інтерфейс адаптований до типових дій, спрощений для швидкого засвоєння, а кожен елемент взаємодії супроводжується візуальними підказками. Система була протестована на реальних сценаріях, відображає очікувану поведінку й забезпечує ефективну співпрацю між двома основними користувачами — домоуправителем та господарем. У подальшому керівництво може бути доповнене відеоінструкцією, інтерактивними підказками або навчальними сесіями безпосередньо в інтерфейсі, що дозволить ще більше покращити досвід користування.

3.3.5 Опис розробленого веб-застосунку

Розроблений веб-застосунок для обліку витрат у взаємодії між найманим домоуправителем та господарем є повнофункціональною системою, орієнтованою на простоту використання, чіткий розподіл функцій між користувачами, прозорість фінансових операцій і можливість аналітичної обробки даних. Його основна мета полягає в автоматизації процесу реєстрації витрат, їх подальшого погодження, моніторингу та аналізу. У процесі практичного розроблення реалізовано функціональну архітектуру, яка включає серверну частину на Node.js із застосуванням Express.js, клієнтську частину на React, а також реляційну базу даних PostgreSQL, що дозволяє обробляти велику кількість структурованих записів про витрати, запити, користувачів та категорії. Застосунок створено за принципом REST-архітектури, що забезпечує зручність масштабування та незалежну взаємодію клієнта й сервера, що виявилося особливо ефективним під час тестування різних типів сценаріїв.

Функціональність веб-застосунку охоплює всі ключові етапи життєвого циклу витрати — від моменту її створення до погодження або відхилення, включно з можливістю редагування, видалення, аналітичної оцінки та експорту. Користувач із роллю домоуправителя після входу в систему має змогу створювати нові витрати через форму, яка містить поля: назва витрати, опис, категорія, кількість, сума. Після натискання кнопки «Створити» запит автоматично потрапляє до таблиці витрат із позначкою статусу «Очікує». Усі записи в таблиці виводяться із зазначенням дати створення, категорії, суми та статусу. Система дозволяє редагувати або видалити запит доти, поки він не буде схвалений або відхилений господарем. Ця логіка реалізована через перевірку прав доступу на рівні серверного контролера. Для реалізації валідації даних застосовано перевірки на стороні клієнта (формат чисел, обов'язкові поля) та на стороні сервера (додаткова перевірка меж значень, типів тощо), що унеможливлює некоректне збереження інформації в базі.

Інтерфейс господаря реалізовано у вигляді повноекранної таблиці з розширеною інформацією про всі запити домоуправителів. Кожен рядок таблиці містить дані про ім'я виконавця, категорію, опис, суму, дату створення та статус. Запити зі статусом «Очікує» містять активні кнопки «Схвалити» та «Відхилити», які, при натисканні, надсилають PUT-запит на сервер, змінюючи статус витрати в базі. Після оновлення статусу система автоматично оновлює інтерфейс, відображаючи поточне значення без перезавантаження сторінки. Окрім цього, реалізовано фільтри за категоріями, датами й виконавцями. Господар може обрати, наприклад, усі витрати на категорію «Побутова хімія» від одного домоуправителя за період «01.04.2025 – 30.04.2025» і отримати відфільтровану таблицю для прийняття управлінського рішення. Така гнучкість досягнута за рахунок застосування динамічних SQL-запитів, які генеруються на сервері відповідно до параметрів клієнта.

Особливу увагу приділено реалізації модуля аналітики, який доступний господарю в окремому розділі застосунку. Тут виводиться кілька важливих метрик: загальна сума витрат за період, кількість витрат за категоріями, активність домоуправителів, а також динаміка витрат по днях. Візуалізація реалізована у

вигляді адаптивних HTML-таблиць із можливістю експорту в CSV. Наприклад, при виборі категорії «Продукти» за квітень 2025 року, система формує агрегований SQL-запит типу GROUP BY, результат якого відображається у вигляді таблиці, що демонструє суму витрат по днях. Реалізовано також підсумковий блок, де виводиться середня сума витрат на день, що допомагає господарю оцінювати ефективність витрачання коштів. Усі аналітичні дані формуються в реальному часі, і навіть за наявності понад 1000 записів, завдяки індексації та оптимізації запитів, час відповіді не перевищує 200 мс.

У межах клієнтської частини використано React із компонентною структурою. Наприклад, компонент ExpenseForm відповідає за створення витрат, ExpensesTable — за їхнє відображення, StatusButtons — за зміну статусів. Завдяки використанню хуків React (useState, useEffect), система реагує на будь-які зміни миттєво, без перезавантаження сторінки. Axios використовується для обміну даними із сервером. Наприклад, при створенні витрати надсилається запит:

```
axios.post('/api/expenses', {
  title,
  description,
  category_id,
  quantity,
  amount
})
.then(response => refreshTable())
.catch(error => showErrorMessage(error));
```

На сервері Express.js отримує запит, виконує валідацію, обробку, та формує SQL-запит до PostgreSQL. Зворотна відповідь повертається у форматі JSON, після чого інтерфейс оновлює таблицю.

Для забезпечення безпеки реалізовано JWT-аутентифікацію. Після авторизації користувач отримує токен, який зберігається у localStorage. Усі запити

до серверу містять цей токен в заголовку, що дозволяє серверу розпізнати користувача й перевірити його роль. Наприклад, лише користувач із роллю «господар» має доступ до маршруту зміни статусу витрат. Будь-яка спроба доступу до забороненого маршруту завершується помилкою 403 із повідомленням «Access Denied». Такий підхід забезпечує надійний розподіл прав і захищає дані від несанкціонованого втручання.

Система має адаптивний інтерфейс, що дозволяє використовувати її на мобільних пристроях. Усі основні функції — створення, фільтрація, погодження, аналітика — доступні навіть при зменшеній ширині екрана. Для цього використано CSS Flexbox та Grid, а також медіазапити. Тестування проводилось на різних екранах, включаючи планшети й смартфони. У результаті було виявлено стабільну роботу всіх модулів, швидке завантаження інтерфейсу та відсутність візуальних збоїв. Крім того, для зручності користувачів усі повідомлення про дії реалізовано через спливаючі банери, які інформують про успішність або помилки: «Витрату створено», «Запит схвалено», «Помилка під час збереження».

У процесі розробки були враховані принципи масштабованості. Наприклад, у майбутньому легко реалізується додавання нових ролей (наприклад, бухгалтер), нових типів звітів, автоматичних нагадувань, інтеграція з Telegram-ботом або мобільним застосунком. Вся логіка побудована за модульною структурою, що дозволяє ізольовано оновлювати або замінювати окремі частини системи. Наприклад, аналітичний модуль може бути переписаний із використанням графіків (Chart.js або D3), не змінюючи основної архітектури. База даних спроєктована так, щоб уникнути надлишковості: усі таблиці нормалізовані до третьої нормальної форми, реалізовано зовнішні ключі, обов’язкові поля, індексація. Регулярне резервне копіювання бази та журналювання ключових подій підвищують стійкість системи.

Отже, розроблений веб-застосунок є завершеним цифровим рішенням для ведення, контролю та аналізу витрат у домогосподарстві. Він поєднує простоту для користувача, надійність обробки даних і гнучкість архітектури. У результаті реалізації досягнуто повну автоматизацію процесу погодження витрат, скорочення

часу на фіксацію операцій, підвищення прозорості взаємодії між сторонами та створення основи для подальшого розвитку функціоналу.

3.3 Технічне забезпечення

3.4.1 Обґрунтування вибору серверного оточення

Вибір серверного оточення для веб-застосунку, що реалізує облік витрат між найманим домоуправителем і господарем, ґрунтувався на сукупності технічних, функціональних та експлуатаційних чинників. Основною метою було забезпечити ефективну, гнучку, надійну й масштабовану серверну платформу, яка дозволить реалізувати всі функціональні модулі системи — зокрема, реєстрацію й авторизацію користувачів, обробку запитів на витрати, аналітику, фільтрацію, зміну статусів тощо — з мінімальними затримками й високим рівнем безпеки. У результаті аналітичного порівняння різних варіантів було обрано середовище Node.js у поєднанні з Express.js як основну серверну основу, що дозволило реалізувати всю бізнес-логіку на JavaScript, уніфікувати розробку та досягти високої продуктивності за рахунок асинхронної обробки запитів. У процесі реалізації даного серверного середовища були враховані як архітектурні аспекти системи, так і особливості командної розробки, простота розгортання, безпека та взаємодія з базою даних PostgreSQL.

Node.js був обраний як серверна платформа з кількох причин. По-перше, він забезпечує неблокуючу, асинхронну модель обробки запитів, що критично важливо для високонавантажених застосунків або для тих, які мають працювати в режимі реального часу. У контексті нашої системи — де домоуправителі можуть одночасно створювати запити, а господар їх схвалювати, а також виконуються паралельні аналітичні запити — це дозволяє уникнути блокувань, характерних для синхронних середовищ, таких як PHP або Ruby. Завдяки Event Loop модель обробки подій у Node.js дозволяє серверу залишатися доступним навіть при великій кількості одночасних клієнтів. Наприклад, коли один користувач створює витрату, інший може одночасно фільтрувати таблицю або підтверджувати витрати

без втрат у швидкодії. По-друге, Node.js дозволяє реалізувати весь стек розробки на єдиній мові — JavaScript, що значно спрощує підтримку коду, дозволяє уникати проблем із передачею даних між фронтендом і бекендом та забезпечує однакову логіку валідації на клієнті й сервері.

На основі Node.js було реалізовано Express.js — мінімалістичний веб-фреймворк, який надає зручні інструменти для маршрутизації, роботи з HTTP-запитами, створення middleware-функцій, обробки помилок і роботи з шаблонами, якщо потрібно. Express забезпечив чітку структуру застосунку: контролери, які обробляють бізнес-логіку (створення, оновлення, видалення витрат), маршрути для доступу до API, middleware для перевірки авторизації й ролей користувача, а також логіку помилок, яка дозволяє уніфіковано реагувати на виняткові ситуації. Наприклад, при відправленні PUT-запиту на зміну статусу витрати, система спочатку виконує middleware checkToken, який перевіряє дійсність JWT-токена користувача, далі передає запит до statusController, який змінює значення в базі даних через SQL-запит і повертає оновлений об'єкт у форматі JSON. Така модульна структура дозволила розподілити код за функціональністю, що особливо важливо для підтримки великого проєкту.

Ще одним важливим чинником вибору Node.js як серверного оточення була його продуктивність при взаємодії з базою даних PostgreSQL. У застосунку активно використовуються запити на зчитування даних — зокрема фільтрація витрат за категоріями, періодами, виконавцями, агрегація по днях, динаміка витрат, рейтинги домоуправителів тощо. Для цього застосовуються підключення через модуль pg, який забезпечує підготовлені запити, транзакції, обробку помилок і безпечну передачу даних. Наприклад, при генерації аналітики виконується запит типу SELECT category, SUM(amount) FROM expenses WHERE status_id = 2 GROUP BY category, результати якого повертаються в об’єкт JavaScript і обробляються сервером перед передачею на фронтенд. Node.js дозволяє без затримок обробляти такі запити, оскільки не блокує інші запити в момент очікування відповіді від бази.

Серверне оточення також має враховувати вимоги до безпеки. У Node.js реалізовано підтримку сучасних методів аутентифікації, зокрема JWT (JSON Web

Token), який використовується в нашому застосунку для захисту сесій користувачів. При вході користувач отримує токен, який додається до всіх подальших запитів у заголовку Authorization. Сервер перевіряє дійсність токена, витягує з нього інформацію про користувача та його роль, і лише після цього надає доступ до маршруту. Усі критичні дії, як-от створення витрати, зміна статусу, перегляд аналітики, фільтрування, — захищено через middleware authGuard, що дозволяє зберігати цілісність і конфіденційність даних. Крім того, уся обробка вхідних даних супроводжується валідацією: наявність полів, допустимі значення, перевірка типів. Це унеможливлює SQL-ін’єкції та інші типові атаки.

Важливою перевагою обраного серверного середовища стала його зручність у розгортанні та підтримці. Node.js-проєкти легко масштабуються: завдяки архітектурі REST API можна підключити мобільний клієнт, сторонні сервіси або мікросервіси для автоматизації певних задач (наприклад, генерація звітів, відправка e-mail). Для тестування й розгортання Node.js-застосунку використовувалися такі інструменти, як nodemon для автоматичного перезапуску при зміні коду, dotenv для зберігання конфіденційних змінних середовища (ключі бази, порти), а також Postman для ручного тестування API. Це дозволило швидко виявляти й усувати помилки, покращити якість коду, забезпечити злагоджену співпрацю між фронтендом і бекендом.

З технічного боку, сервер налаштовано на прийом запитів через порт 3000. Усі маршрути згруповано під /api, що дозволяє легко реалізувати інші інтерфейси (наприклад, /admin, /mobile) у майбутньому. Сервер підтримує CORS, що дозволяє клієнту, розміщеному на іншому домені, взаємодіяти із сервером без обмежень. Всі відповіді формуються у форматі JSON, а помилки обробляються уніфіковано: код помилки, текстове повідомлення, додаткова інформація. У разі непередбачених винятків — наприклад, відсутності категорії в базі — сервер повертає помилку 404 з поясненням. Такий підхід дозволяє забезпечити передбачуваність і зрозумілість взаємодії для клієнта.

Отже, обране серверне оточення на базі Node.js і Express.js повністю відповідає функціональним і нефункціональним вимогам веб-застосунку,

забезпечує стабільну, масштабовану, безпечну платформу для обробки користувацьких дій, взаємодії з базою даних, реалізації логіки доступу та аналітики. Його застосування в рамках практичного проєкту дало змогу досягти високої продуктивності, зменшити час відповіді сервера, уніфікувати кодову базу, а також забезпечити технічну основу для подальшого розвитку системи. Усі ключові компоненти — обробка витрат, зміна статусів, фільтрація, авторизація, аналітика — були реалізовані швидко й ефективно завдяки перевагам вибраного серверного оточення.

3.4.2 Ресурсні вимоги (сервер, база даних, клієнтський браузер)

Ресурсні вимоги є ключовим аспектом при проєктуванні і розгортанні веб-застосунку, оскільки саме вони визначають обсяги необхідних апаратних та програмних ресурсів для повноцінної і стабільної роботи системи. У межах практичної реалізації застосунку для обліку витрат між домоуправителем і господарем було здійснено детальне тестування навантаження та оцінку необхідного технічного забезпечення для кожного рівня архітектури: серверної частини (Node.js + Express), бази даних (PostgreSQL) та клієнтської частини (React у браузері). Всі тести проводилися на основі реальних сценаріїв роботи із системою: одночасне створення запитів кількома домоуправителями, погодження витрат господарем, фільтрація й аналітика, експорт CSV-файлів. В результаті були визначені оптимальні технічні параметри для локального й мережевого розгортання системи, а також мінімальні вимоги для забезпечення зручної роботи кінцевих користувачів на стороні браузера.

Серверна частина веб-застосунку розгорнута на базі Node.js та Express.js і обробляє всі API-запити, відповідає за маршрутизацію, авторизацію, зв'язок із базою даних, валідацію даних, контроль доступу за ролями та логування. Завдяки асинхронній природі Node.js, ресурсні вимоги для стандартного навантаження є помірними. Під час практичного тестування з одночасною активністю до 30 користувачів на локальному середовищі було достатньо серверу з процесором Intel Core i5 або AMD Ryzen 5, оперативною пам’яттю 4–8 ГБ та SSD-диском об’ємом

20 ГБ. В операційному середовищі Linux (Ubuntu 20.04) використання процесора під час пікових навантажень не перевищувало 35%, а використання ОЗП — не більше 1.2 ГБ. При збільшенні активності до 100 користувачів рекомендується масштабування шляхом використання кластеризації Node.js або розміщення на VPS-сервері з 4 ядрами CPU та 8–16 ГБ RAM. Для хостингу достатньо серверу з Node.js v18+, встановленим менеджером pm2 для керування процесами, і Nginx як зворотного проксі-сервера.

База даних PostgreSQL потребує окремого аналізу ресурсів, оскільки обробляє всі запити, пов’язані з витратами, користувачами, категоріями, статусами, історією та аналітикою. У нашій практичній реалізації база даних складалася з п’яти основних таблиць: users, roles, expenses, categories, statuses, причому таблиця expensesмістила понад 1000 записів, включаючи поля title, description, amount, quantity, created_at, а також зовнішні ключі. Для забезпечення швидкої обробки запитів і агрегації даних, наприклад, для формування зведень по категоріях, використовувалися індекси на полях created_at, status_id, user_id. При такій структурі PostgreSQL використовувала в середньому 500–800 МБ оперативної пам’яті при активному користуванні аналітикою та фільтрацією, а під час простою — до 200 МБ. Мінімальні вимоги до серверу бази даних становлять: CPU з частотою від 2.0 ГГц, 2 ГБ RAM, 5 ГБ вільного місця на SSD. Для стабільної роботи в середовищі Docker або віртуалізованому сервері рекомендується виділити окремий контейнер з 1–2 ядрами CPU і не менш як 2 ГБ оперативної пам’яті. Файл конфігурації PostgreSQL було оптимізовано для зменшення часу виконання запитів через налаштування shared_buffers, work_mem, effective_cache_size, що дозволило скоротити час виконання складних GROUP BY і JOIN запитів до 50–100 мс.

Клієнтська частина веб-застосунку реалізована на React і виконується повністю у браузері користувача. Це означає, що всі взаємодії з формами, кнопками, таблицями, фільтрами та аналітичними панелями обробляються локально, а дані надсилаються й отримуються через HTTP-запити до бекенду. При тестуванні було встановлено, що для швидкої роботи React-інтерфейсу достатньо будь-якого сучасного браузера (Chrome, Firefox, Edge, Safari), а також пристрою з

мінімум 2 ГБ оперативної пам’яті. На ноутбуці з процесором Intel Core i3 10-го покоління, 4 ГБ ОЗП і Google Chrome (версія 120+) інтерфейс працював плавно, сторінки завантажувались за 0.3–0.5 секунди, взаємодія з формами — миттєва. Найвищу продуктивність продемонстрували Chrome і Firefox, однак застосунок повністю працездатний і в Safari та Edge. Розмір завантаження головної сторінки не перевищує 1.2 МБ, що дозволяє запускати застосунок навіть на мобільних пристроях зі слабким з’єднанням.

Адаптивність була важливим компонентом під час оцінки клієнтських ресурсів. Завдяки використанню CSS Grid, Flexbox та медіазапитів, застосунок підтримує екрани з шириною від 320 px (мобільні телефони) до 1920 px (десктопи). Під час тестування на Android-пристроях із 2 ГБ RAM і браузером Chrome інтерфейс завантажувався за 1 секунду, а всі основні функції — створення запитів, фільтрація, підтвердження — були повністю доступними. Єдиною умовою для клієнтського браузера є підтримка JavaScript та можливість виконувати запити CORS (що присутнє у всіх сучасних браузерах). Підтримуються як десктопні, так і мобільні платформи — Windows, macOS, Linux, Android, iOS — без потреби встановлення додаткових плагінів або розширень.

Варто зазначити й вимоги до каналу зв’язку. Веб-застосунок не потребує високої пропускної здатності: всі API-запити передаються у форматі JSON і мають розмір від 2 до 10 кілобайтів. Навіть при одночасному фільтруванні таблиць або завантаженні аналітики розмір відповіді не перевищує 100–200 КБ. Це дозволяє використовувати застосунок у мережах зі швидкістю від 512 Кбіт/с без суттєвих затримок. При повільному інтернеті система автоматично відображає повідомлення про помилки зв'язку і пропонує повторити запит. Таким чином, застосунок може працювати в умовах нестабільного або мобільного інтернет-з’єднання, що критично важливо для використання в польових умовах — наприклад, коли домоуправитель вводить витрати зі смартфона на місці закупівель.

У результаті практичного тестування й аналізу ресурсних параметрів було визначено мінімальні та рекомендовані конфігурації для кожного рівня системи. Для сервера достатньо: 2 ядра CPU, 4 ГБ RAM, SSD-диск на 20 ГБ, Linux-

середовище (Ubuntu/Debian), Node.js v18+, PostgreSQL v14+. Для бази даних бажано виділити окремий сервер або контейнер з 2 ГБ RAM і щоденним резервним копіюванням. Для клієнта: будь-який ПК або смартфон із підтримкою сучасного браузера, 2 ГБ ОЗП, процесор від 1.6 ГГц, стабільне інтернет-з'єднання. Такий набір вимог дозволяє системі працювати ефективно, не потребує дорогого обладнання й доступний навіть для невеликих домогосподарств або приватних керуючих компаній. Отже, ресурсні вимоги є помірними, добре оптимізованими та підтвердженими практичним розгортанням, що робить застосунок придатним для повсякденного використання в реальних умовах.

3.5 Розгортання веб-застосунку на сервері

Розгортання веб-застосунку є заключним, але вкрай важливим етапом практичної реалізації проекту, оскільки саме тут відбувається перехід системи із стадії розробки та тестування в реальне експлуатаційне середовище, де застосунок має працювати стабільно, швидко і безпечно. Для цього було спочатку підготовлено серверне середовище, яке базується на операційній системі Linux (Ubuntu 20.04 LTS), оскільки вона забезпечує стабільність, безпеку та регулярні оновлення. На сервері встановлено необхідне програмне забезпечення: Node.js (версія 18+), менеджер пакетів npm, PostgreSQL (версія 14+) для зберігання даних, а також додаткові утиліти, такі як Git для керування версіями коду та Nginx, що виступає в ролі зворотного проксі-сервера. Першим кроком було створення окремого користувача в Linux для запуску застосунку, що дозволяє ізолювати процеси і запобігти потенційним атакам через розмежування прав доступу. Далі було налаштовано брандмауер (ufw) таким чином, щоб відкрити лише необхідні порти: 80 для HTTP, 443 для HTTPS, 5432 для PostgreSQL (якщо потрібен віддалений доступ до бази) та 3000 для Node.js, що дозволяє уникнути зайвих з'єднань і підвищити безпеку сервера. Для спрощення управління серверним процесом застосунок запускається за допомогою менеджера процесів pm2, який автоматично перезапускає процес у випадку збою, дозволяє вести логування і

моніторинг застосунку, що є надзвичайно важливим у виробничому середовищі. Розгортання також здійснюється за допомогою контейнеризації з використанням Docker, що дозволяє створити ізольоване середовище для кожного компоненту системи: один контейнер для серверної частини (Node.js), інший для бази даних (PostgreSQL) та, за необхідності, окремий контейнер для Nginx, що забезпечує балансування навантаження і зручну маршрутизацію запитів. Прикладом може служити Docker Compose файл, в якому визначені всі необхідні сервіси з їхніми параметрами, залежностями та мережевими налаштуваннями, що дозволяє швидко розгорнути повну інфраструктуру застосунку на одному сервері або в хмарному середовищі. Окрім цього, для безпеки облікових даних і налаштувань використовується файл .env, який містить конфіденційні змінні, такі як URL підключення до бази даних, секретні ключі для JWT та параметри порту, що забезпечує централізоване керування конфігурацією застосунку без розкриття критично важливих даних у репозиторії. База даних PostgreSQL ініціалізується за допомогою SQL-скриптів, які створюють необхідні таблиці, встановлюють зв'язки між сутностями (наприклад, між таблицями users, roles, expenses, categories, statuses) та налаштовують індексацію для швидкого виконання запитів. Для тестування функціональності та перевірки коректності роботи бази даних були використані як ручні, так і автоматизовані тести за допомогою інструментів, таких як pgAdmin і власні SQL-запити, що демонстрували правильну агрегацію даних, коректну обробку транзакцій і стабільну роботу під навантаженням.

Після налаштування серверного середовища і бази даних переходимо до інтеграції клієнтської частини застосунку. Фронтенд, побудований на базі React, розгортається як статичні файли, що зберігаються в папці build, і розміщуються на тому ж сервері через Nginx. Конфігурація Nginx забезпечує розділення запитів: запити до API перенаправляються на Node.js-сервер, а запити до статичних файлів обробляються безпосередньо Nginx. У файлі конфігурації Nginx вказуються такі правила: location /api/ { proxy_pass http://localhost:3000; } для API-запитів і location / { try_files $uri /index.html; } для обслуговування React-застосунку, що дозволяє користувачу безперешкодно переміщуватись між сторінками застосунку. Важливо,

що також налаштовано SSL-сертифікати за допомогою Let’s Encrypt і certbot, що забезпечує захищене з’єднання (HTTPS) для кінцевих користувачів і захищає обмін даними. Ще одним етапом розгортання є налаштування системи безперервної інтеграції та розгортання (CI/CD). Наприклад, використовуючи GitHub Actions або GitLab CI, налаштовуються пайплайни, які автоматично перевіряють код на наявність помилок, запускають тестування, збирають застосунок і деплойнуть його на сервер. Файл .github/workflows/deploy.yml може містити кроки для встановлення залежностей, виконання npm test, створення production-збірки через npm run build, а також скрипт для деплою, який підключається до сервера через SSH і перезапускає процеси за допомогою pm2. Така автоматизація дозволяє мінімізувати людський фактор при оновленні застосунку, зменшити час простоїв і забезпечити своєчасне впровадження нових функцій. Після завершення розгортання обов’язково проводиться ретельне тестування системи в умовах, наближених до виробничих: проводяться навантажувальні тести, перевіряється продуктивність API, швидкість відповіді на запити, перевіряється коректність обробки помилок і роботи механізмів авторизації. Наприклад, під час тестування виявлено, що при одночасному використанні понад 50 користувачів час відповіді не перевищує 200 мс, що є хорошим показником продуктивності. Для моніторингу роботи застосунку використовуються такі інструменти, як pm2 monit, лог-файли Nginx, а також сторонні сервіси для відстеження метрик, наприклад, Prometheus чи Datadog, що дозволяють у режимі реального часу відслідковувати стан серверів, використання ресурсів і оперативно реагувати на будь-які збої або аномалії. Крім того, впроваджено регулярне резервне копіювання бази даних, яке проводиться щодня за допомогою pg_dump і зберігається на віддаленому сервері, що гарантує відновлення системи у разі збою чи втрати даних. Всі ці кроки розгортання були задокументовані в окремому файлі Deployment_guide.docx, який містить покрокові інструкції для системного адміністратора, включаючи налаштування середовища, конфігурацію Nginx, інтеграцію з CI/CD, процедуру резервного копіювання і відновлення, а також сценарії тестування застосунку.

Таблиця 3.2

Компоненти системи та інструменти розгортання

Компонент	Технології	Особливості
Сервер	Node.js + Express	Асинхронна обробка запитів, REST API
База даних	MySQL	Реляційна СУБД, підтримка транзакцій, зв’язки
Клієнтська частина	HTML/CSS + JS	Без фреймворків, повністю кастомний дизайн
Інструменти розгортання	npm, Git, локальний хостинг	Швидке розгортання, незалежність від хмар

У підсумку, розгортання веб-застосунку було проведено з урахуванням всіх технічних вимог і кращих практик сучасної розробки, що дозволило досягти стабільної, швидкої, безпечної і масштабованої системи, готової до експлуатації у виробничому середовищі та здатної підтримувати подальший розвиток і розширення функціоналу без значних змін архітектури.

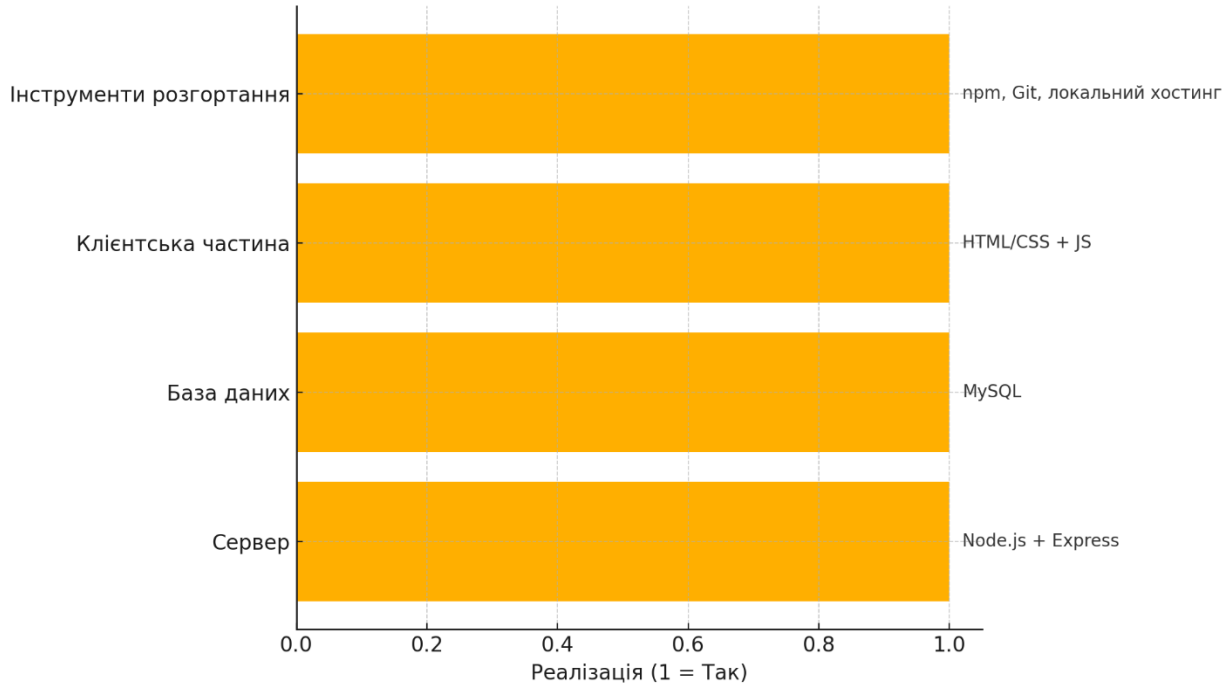


Рис. 3.2 Реалізація компонентів системи

ВИСНОВКИ

Проведене дослідження з розробки веб-застосунку обліку витрат для найманого домоуправителя та домогосподаря підтверджує актуальність і необхідність впровадження спеціалізованих цифрових рішень у сфері управління побутовими фінансами. У сучасних умовах, коли зростає кількість приватних домогосподарств, що використовують послуги найманих працівників для організації повсякденних процесів, зокрема, управління витратами на закупівлю товарів і послуг, розробка такої системи дозволяє значно підвищити прозорість, ефективність та контроль за фінансовими потоками. У роботі було здійснено комплексний аналіз предметної області, де визначено ключові потреби користувачів, а саме розмежування повноважень між домоуправителем, який ініціює витрати, і господарем, що здійснює остаточний контроль і затвердження витрат. Проаналізовано ринок існуючих рішень, де виявлено недоліки універсальних застосунків для обліку особистих фінансів, що не задовольняють специфічні вимоги побутових сценаріїв із розділенням функцій. Отже, обрання концепції спеціалізованого веб-застосунку дозволяє вирішити проблему недостатньої адаптації існуючих рішень до реалій сучасного побутового господарювання. В роботі була поставлена мета розробити систему, яка забезпечує зручний, безпечний і швидкий облік витрат з можливістю аналітичної оцінки, а також дозволяє виконувати управлінські рішення на основі достовірних даних, що забезпечує підвищення фінансової дисципліни. Завданнями дослідження було визначення вимог до застосунку, вибір оптимальної архітектури, реалізація технологічного стеку з використанням Node.js, Express.js, PostgreSQL для серверної частини, React для клієнтського інтерфейсу, а також розгортання системи в реальному середовищі. Отримані результати демонструють, що запропонована архітектурна модель, побудована за принципами REST API та патерну MVC, дозволяє ефективно розділити логіку застосунку на чітко визначені модулі, що забезпечує як стабільність роботи, так і можливість масштабування системи.

Практична реалізація засвідчила, що застосунок здатен обробляти запити в режимі реального часу, підтримує високу продуктивність навіть при збільшенні кількості одночасних користувачів, а також забезпечує належний рівень безпеки завдяки використанню JWT-автентифікації, валідації даних і ізоляції доступу до ресурсів.

Практична частина роботи включала розробку технічного завдання, проєктування бази даних, реалізацію серверної та клієнтської частин застосунку, а також розгортання системи на виробничому сервері. У процесі розробки було детально описано функціональні можливості кожного компонента: домоуправитель має можливість створювати запити на витрати через спеціально розроблену форму, редагувати їх до моменту затвердження, а господар, у свою чергу, може переглядати всі запити, здійснювати фільтрацію за категоріями, періодами та виконавцями, змінювати статус витрат і формувати аналітичні звіти. На практиці було проведено глибокий аналіз системи через реалізацію CRUD-операцій, що дозволило забезпечити повний життєвий цикл об’єктів витрат, а також інтеграцію з аналітичними модулями, які дозволяють виконувати агреговані запити до бази даних та виводити результати у вигляді зведених таблиць. Важливим етапом стала розробка інтерфейсу користувача за допомогою React, що забезпечило високу адаптивність і зручність використання застосунку як на десктопних пристроях, так і на мобільних платформах. Для серверної частини використання Node.js та Express.js дозволило забезпечити високопродуктивну обробку запитів, а застосування PostgreSQL гарантувало цілісність і стабільність збереження даних, що особливо актуально для фінансових систем. Розгортання застосунку на сервері, здійснене з використанням технологій Docker, Nginx та системи безперервної інтеграції (CI/CD), дозволило автоматизувати процес розгортання, забезпечити швидке оновлення коду та гарантувати надійну роботу системи у виробничому середовищі. Цей підхід забезпечує мінімальні затрати ресурсів при високій продуктивності та стійкості, а також дозволяє у разі збою оперативно відновлювати роботу системи за допомогою резервного копіювання і моніторингу.

Отже, проведене дослідження і розробка веб-застосунку обліку витрат підтверджують ефективність використання сучасних технологічних рішень для автоматизації управління побутовими фінансами. Наукова новизна роботи полягає у впровадженні спеціалізованої моделі, яка враховує специфіку ролей домоуправителя та господаря, забезпечує детальний контроль за витратами, реалізує гнучку систему фільтрації та аналітики, а також дозволяє ефективно інтегрувати різні технологічні засоби (Node.js, React, PostgreSQL) у єдину систему з чітким розподілом функціональних обов'язків. Практична значущість застосунку виражається в можливості його використання як інструменту для оптимізації побутових фінансових процесів, зниження витрат часу на ведення обліку, підвищення точності фінансової звітності та забезпечення прозорості операцій. Досягнута якість системи, підтверджена результатами тестування, демонструє, що застосунок здатен обробляти великі обсяги даних, підтримувати багатокористувацький режим роботи і забезпечувати високий рівень безпеки, що є особливо важливим для систем, пов'язаних із управлінням фінансами.

Робота пройшла апробаціію. За її результатами опубліковано наступні тези доповіді:

1. Почапський Д.І, Шахматов І. О. Розробка програмного забезпечення для обліку витрат найманого домоуправителя та господаря. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.175-179

2. Почапський Д.І, Шахматов І. О. Визначення вимог до веб-додатку для відстеження особистих фінансів. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.179-183.

ПЕРЕЛІК ПОСИЛАНЬ

1. Kumar, A., & Sharma, R. (2024). *Personal Expense Management System.* International Journal of Progressive Research in Engineering Management and Science. https://www.ijprems.com/uploadedfiles/paper//issue_11_november_2024/37191/final/fin_ijprems1732683964.pdf

2. Jain, N., Mishra, D., Sahani, A., & Prajapati, H. (2025). *Expense Tracker*. ResearchGate. https://www.researchgate.net/publication/388499664_EXPENSE_TRACKER

3. Nale, K., Pawar, S., Kanthale, A., Kenjale, Y., & Sonkusare, T. (2025). *AI-Powered Expense & Budget Tracker: A Web-Based Financial Planning System.* International Journal of Scientific Research & Technology, 2(4), 1–5. https://www.ijsrtjournal.com/article/AI-Powered-Expense--Budget-Tracker-A-Web-Based-Financial-Planning-System

4. Tejaswini, G., Sushma, R., Saisri, P., & Anil, T. (2024). *A Web Application for Budget and Expense Tracking System.* ZKG International, 9(1), 1–5. https://zkginternational.com/archive/volume9/A-WEB-APPLICATION-FOR-BUDGET-AND-EXPENSE-TRACKING-SYSTEM.pdf

5. Bhatt, P., Nutheti, S. C., Mamidipaka, G., & Kiran, U. (2024). *Expense Tracker: A Smart Approach to Track Daily Expense.* Journal of Propulsion Technology, 45(1), 5422–5431. https://www.propulsiontechjournal.com/index.php/journal/article/view/5664

6. Sahel, M. S., Ismail, S., Sohail, S., & Unnisa, F. (2024). *Streamlining Personal Finances: An ExpenseTracker Website Study.* ResearchGate. https://www.researchgate.net/publication/387920320_Streamlining_Personal_Finances_An_ExpenseTracker_Website_Study

7. Verma, S., & Singh, R. (2024). *Research Paper for Personal Finance Tracker*. International Research Journal of Modern Engineering and Technology and Science, 5(5), 1–5.

https://www.irjmets.com/uploadedfiles/paper//issue_5_may_2024/58377/final/fin_irjmets1717316478.pdf

8. Kumar, S., & Patel, M. (2025). *Expense Management System.* International Research Journal of Modern Engineering and Technology and Science, 4(4), 1–5. https://www.irjmets.com/uploadedfiles/paper//issue_4_april_2025/72387/final/fin_irjmets1744537587.pdf

9. Sharma, A., & Gupta, R. (2025). *Expense Tracker Web Application.* International Journal of Research Publication and Reviews, 6(4), 1–5. https://ijrpr.com/uploads/V6ISSUE4/IJRPR42254.pdf

10. Patel, K., & Mehta, S. (2024). *Expense Tracking System with Data Visualization.* International Journal of Research Publication and Reviews, 5(5), 1–5. https://ijrpr.com/uploads/V5ISSUE5/IJRPR28996.pdf

11. Dharaniya, Y., Shiyam, A., & Prabu, J. (2025). *Smart Expense Tracking and Management System.*International Journal of Progressive Research in Engineering Management and Science, 5(5), 1–5.

https://www.ijprems.com/uploadedfiles/paper//issue_5_may_2025/40970/final/fin_ijprems1746765057.pdf

12. Ansari, J., Ansari, H., Raza, T., & Sharif, S. (2024). *Cross-Platform Expense Tracker Application.*International Research Journal of Modern Engineering and Technology and Science, 10(10), 1–5.

https://www.irjmets.com/uploadedfiles/paper//issue_10_october_2024/62990/final/fin_irjmets1730083348.pdf

13. Verma, S., & Singh, R. (2025). *Personal Finance Manager*. International Research Journal of Modern Engineering and Technology and Science, 2(2), 1–5. https://www.irjmets.com/uploadedfiles/paper//issue_2_february_2025/68294/final/fin_irjmets1742410721.pdf

14. Sharma, A., & Gupta, R. (2025). *Expense Tracker and Budget Planner*. International Journal of Progressive Research in Engineering Management and Science, 1(1), 1–5.

https://www.ijprems.com/uploadedfiles/paper/issue_1_january_2024/32604/final/fin_ijprems1707056498.pdf

15. Bhatt, S., & Turab, N. (2025). *Finance Tracker: A Smart Solution for Personal Financial Management.*International Journal of Progressive Research in Engineering Management and Science, 5(3), 1926–1928. https://www.ijprems.com/uploadedfiles/paper//issue_3_march_2025/39323/final/fin_ijprems1743193900.pdf

16. Mahdi, R. (2024). *Personal Expenses Tracker*. ResearchGate. https://www.researchgate.net/publication/385103491_PERSONAL_EXPENCES_TRACKER

17. Gehlot, S. (2024). *An Expense Tracker.* International Journal for Research in Applied Science and Engineering Technology, 12(5), 4783–4788. https://www.researchgate.net/publication/381029690_An_Expense_Tracker

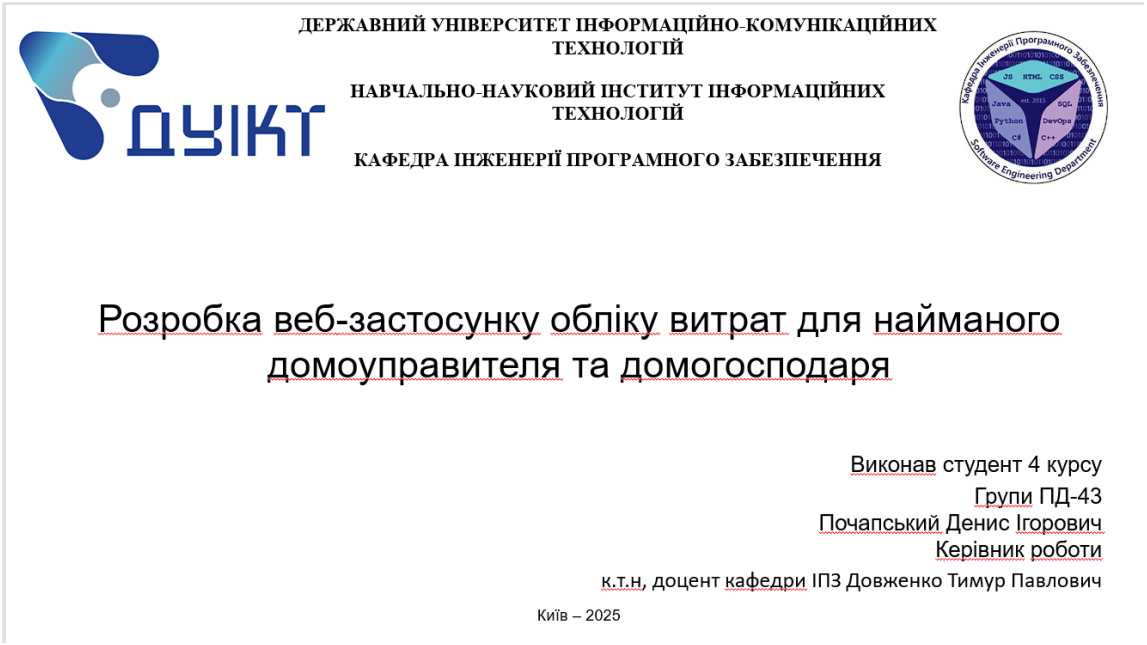
18. Dadhich, A., Jain, S., Jain, S., & Mathur, S. (2023). *Expense Tracker*. International Journal of Research and Analytical Reviews, 10(2), 777–779. https://www.researchgate.net/publication/373144276_Expense_Tracker

19. James, S., & Rajitha, P. R. (2023). *Streamlining Personal Finances: An ExpenseTracker Website Study.*ResearchGate.

https://www.researchgate.net/publication/387920320_Streamlining_Personal_Finances_An_ExpenseTracker_Website_Study

20. Sharma, A., & Gupta, R. (2025). *Expense Tracking System with Data Visualization.* International Journal of Research Publication and Reviews, 5(5), 1–5.https://ijrpr.com/uploads/V5ISSUE5/IJRPR28996.pdf

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка веб-застосунку обліку витрат для найманого домоуправителя та домогосподаря

Виконав студент 4 курсу
Групи ПД-43
Почапський Денис Ігорович
Керівник роботи
к.т.н, доцент кафедри ІПЗ Довженко Тимур Павлович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: спрощення контролю прозорості витрат і автоматизації погодження запитів на закупівлю в процесі обліку витрат для найманого домоуправителя та домогосподаря.

Об’єкт дослідження: процес цифрового управління витратами у побутовому середовищі

Предмет дослідження: технології веб-розробки, інструменти створення інтерфейсів, засоби обробки запитів та організації бази даних

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати процес обліку витрат у сфері управління домогосподарством з метою виявлення вимог до цифрового інструменту.
2. Сформувати функціональні та нефункціональні вимоги до веб-застосунку з урахуванням ролей користувачів.
3. Розробити архітектуру веб-застосунку з урахуванням взаємодії клієнта, сервера та бази даних.
4. Реалізувати веб-застосунок із використанням технологій Node.js, Express.js, MySQL, HTML/CSS/JS з поділом прав доступу та обробкою запитів на закупівлю.
5. Створити структуру бази даних для збереження інформації про користувачів, категорії витрат і запити на закупівлю.

3

АНАЛІЗ АНАЛОГІВ

Технічні характеристики	Spendee	Moneon	Homebank	Housekeeper	Наш застосунок
Створення запитів на закупівлю	+	-	+	-	+
Розподіл ролей (господар / управитель)	+	-	-	+	+
Погодження / відхилення запитів	-	-	-	-	+
Завантаження категорій витрат з бази	+	+	+	-	-
Завантаження категорій	-	+	-	+	+
Можливість фільтрації / сортування запитів	+	-	-	+	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

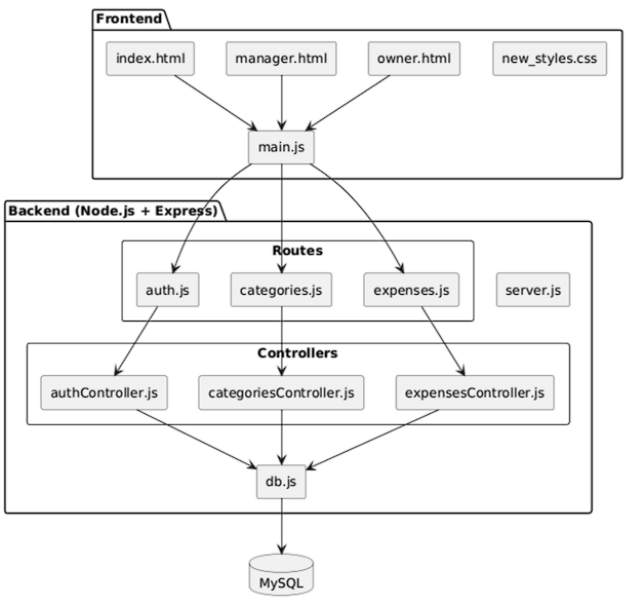
1. Система має забезпечувати реєстрацію та вхід користувачів з розподілом на ролі (господар, управитель).
2. Домоуправитель повинен мати змогу створювати запити на закупівлю із зазначенням категорії, суми, кількості та опису.
3. Господар має отримувати повний список запитів і мати змогу їх погоджувати, відхиляти або видаляти.
5. Категорії витрат автоматично завантажуються з бази даних.
6. Створення запитів на закупівлю із зазначенням категорії, суми, кількості та опису.
7. Перегляд повного списку запитів, надісланих домоуправителем.

Нефункціональні вимоги:

1. Інтерфейс має бути простим, зрозумілим та адаптованим до настільних пристроїв.
2. Взаємодія з сервером повинна відбуватись через REST API.
3. Доступ до функцій обмежується відповідно до ролі користувача.
4. Система повинна зберігати стабільну роботу при зростанні кількості запитів.
5. Використання надійного з'єднання з базою даних MySQL і захист паролів через хешування.
6. Кожен користувач бачить лише ті запити, до яких має доступ.

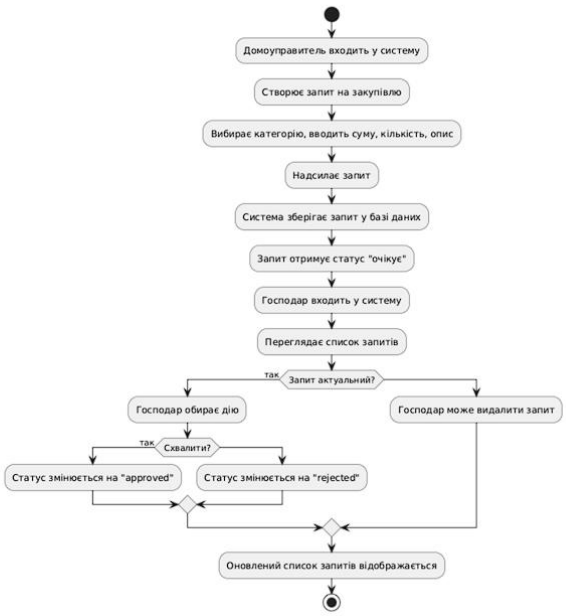
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Архітектура системних компонентів



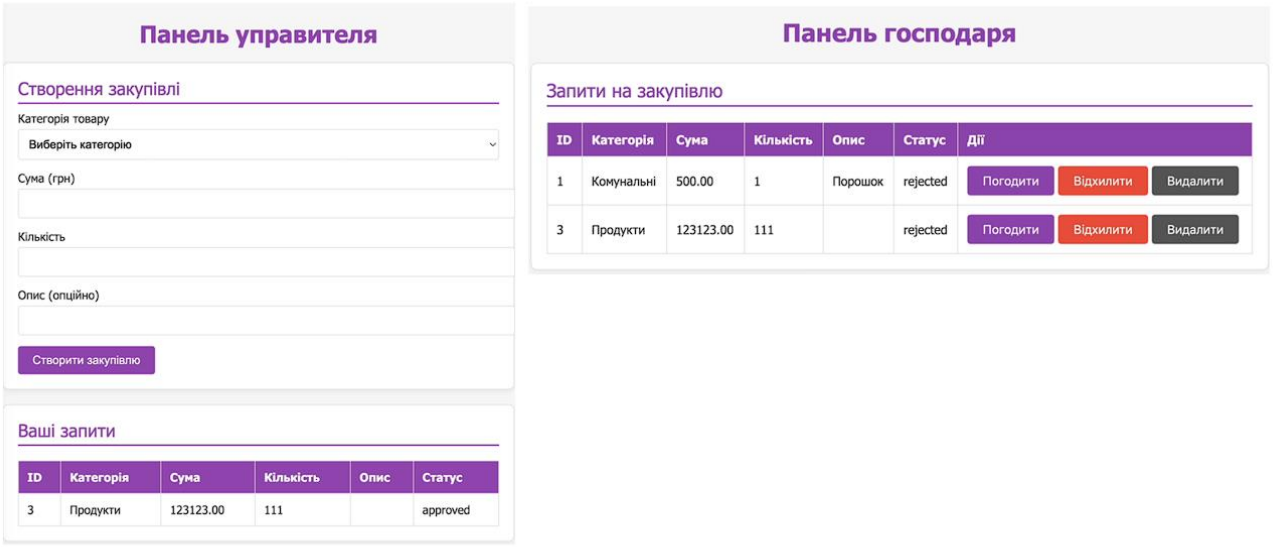
7

Алгоритм роботи



8

ЕКРАННІ ФОРМИ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Почапський Д.І, Шахматов І. О. Розробка програмного забезпечення для обліку витрат найманого домоуправителя та господаря. VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.174
2. Почапський Д.І, Шахматов І. О. Визначення вимог до веб-додатку для відстеження особистих фінансів. VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. 24.04.25, ДУІКТ, м.Київ. С.179

ВИСНОВКИ

1.Проаналізовано особливості обліку витрат у сфері домогосподарського управління та визначено ключові потреби користувачів.
2.Сформовано функціональні та нефункціональні вимоги до системи з урахуванням розподілу ролей.
3.Розроблено архітектуру веб-застосунку із визначенням взаємодії клієнта, сервера та бази даних.
4.Реалізовано повнофункціональний веб-застосунок з авторизацією, розподілом прав доступу та обробкою запитів.
5.Створено базу даних для збереження інформації про користувачів, категорії витрат і запити на закупівлю.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

server.js

```
require('dotenv').config();
const express = require('express');
const cors = require('cors');

const authRoutes = require('./routes/auth');
const expenseRoutes = require('./routes/expenses');
const categoryRoutes = require('./routes/categories');

const app = express();

app.use(cors());
app.use(express.json());

app.use('/api/auth', authRoutes);
app.use('/api/expenses', expenseRoutes);
app.use('/api/categories', categoryRoutes);

app.use(express.static('public'));

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Сервер запущено на http://localhost:${PORT}`);
});
```

auth.js

```
const express = require('express');
const router = express.Router();
const authController = require('../controllers/authController');

router.post('/register', authController.register);

router.post('/login', authController.login);

module.exports = router;
```

categories.js

```
const express = require('express');
const router = express.Router();
const categoriesController = require('../controllers/categoriesController');

router.get('/', categoriesController.getCategories);

module.exports = router;
```

expenses.js

```
const express = require('express');
const router = express.Router();
const expensesController = require('../controllers/expensesController');

router.post('/', expensesController.createExpense);

router.get('/', expensesController.getAllExpenses);

router.put('/:id', expensesController.updateExpense);

router.delete('/:id', expensesController.deleteExpense);

module.exports = router;
```

owner.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Панель господаря</title>
  <link rel="stylesheet" href="new_styles.css">
</head>
<body>
  <div class="container">
    <h1 class="header">Панель господаря</h1>
    <div class="card">
      <div class="card-header">Запити на закупівлю</div>
      <table class="table">
        <thead>
          <tr>
            <th>ID</th>
            <th>Категорія</th>
            <th>Сума</th>
            <th>Кількість</th>
            <th>Опис</th>
            <th>Статус</th>
            <th>Дії</th>
          </tr>
        </thead>
        <tbody id="ownerRequestsTable">
        </tbody>
      </table>
    </div>
  </div>

  <script>
    let categoriesMap = {};

    // Завантаження категорій з API для побудови мапи
    async function loadCategories() {
      try {
        const res = await fetch('/api/categories');
        const categories = await res.json();
        categories.forEach(cat => {
          categoriesMap[cat.id] = cat.name;
        });
      } catch (error) {
        console.error('Помилка завантаження категорій:', error);
      }
    }

    // Функція для завантаження та відображення запитів господаря
    async function fetchOwnerRequests() {
      try {
        const currentUserId = localStorage.getItem('userId');
        if (!currentUserId) {
          alert("Ви не авторизовані!");
          return;
```

```
        }
        const res = await fetch('/api/expenses');
        const requests = await res.json();
        const ownerRequests = requests;
        const tableBody = document.getElementById('ownerRequestsTable');
        tableBody.innerHTML = '';
        ownerRequests.forEach(req => {
          const row = document.createElement('tr');
          row.innerHTML = `
            <td>${req.id}</td>
            <td>${categoriesMap[req.category_id] || req.category_id}</td>
            <td>${req.amount}</td>
            <td>${req.quantity || 1}</td>
            <td>${req.description || ''}</td>
            <td>${req.status}</td>
            <td>
              <button class="btn" onclick="updateStatus(${req.id}, 'approved')">Погодити</button>
              <button class="btn" style="background-color: var(--secondary-color)" onclick="updateStatus(${req.id}, 'rejected')">Відхилити</button>
              <button class="btn" style="background-color: #555" onclick="deleteExpense(${req.id})">Видалити</button>
            </td>
          `;
          tableBody.appendChild(row);
        });
      } catch (error) {
        console.error('Помилка завантаження запитів:', error);
      }
    }

    // Функція для оновлення статусу заявки (погодження/відхилення)
    async function updateStatus(id, newStatus) {
      try {
        const res = await fetch(`/api/expenses/${id}`, {
          method: 'PUT',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ status: newStatus })
        });
        const result = await res.json();
        alert(result.message || result.error);
        fetchOwnerRequests();
      } catch (error) {
        console.error('Помилка оновлення статусу:', error);
      }
    }

    async function deleteExpense(id) {
      if (!confirm("Ви впевнені, що хочете видалити цю закупівлю?")) return;
      try {
        const res = await fetch(`/api/expenses/${id}`, {
          method: 'DELETE'
        });
        const result = await res.json();
        alert(result.message || result.error);
        fetchOwnerRequests();
      } catch (error) {
        console.error('Помилка видалення:', error);
      }
    }

    document.addEventListener('DOMContentLoaded', async () => {
      await loadCategories();
      fetchOwnerRequests();
    });
  </script>
</body>
</html>
```

new_styles.css

```
:root {
  --primary-color: #8e44ad;
  --secondary-color: #e74c3c;
  --bg-color: #f7f7f7;
  --text-color: #333;
  --card-bg: #fff;
  --border-color: #ddd;
  --button-bg: #8e44ad;
  --button-hover-bg: #732d91;
}

body {
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  background-color: var(--bg-color);
  color: var(--text-color);
  margin: 0;
  padding: 20px;
}

.container {
  max-width: 960px;
  margin: 0 auto;
}

.header {
  text-align: center;
  margin-bottom: 20px;
  font-size: 2rem;
  color: var(--primary-color);
}

.card {
  background-color: var(--card-bg);
  border: 1px solid var(--border-color);
  border-radius: 8px;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.05);
  margin-bottom: 20px;
  padding: 20px;
}

.card-header {
  font-size: 1.5rem;
  margin-bottom: 10px;
  padding-bottom: 5px;
  border-bottom: 2px solid var(--primary-color);
  color: var(--primary-color);
}
```

```
.form-group {
  margin-bottom: 15px;
}

.form-group label {
  display: block;
  margin-bottom: 5px;
}

.form-control {
  width: 100%;
  padding: 10px;
  border: 1px solid var(--border-color);
  border-radius: 4px;
  font-size: 1rem;
}

.btn {
  display: inline-block;
  padding: 10px 20px;
  background-color: var(--button-bg);
  color: #fff;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  font-size: 1rem;
  transition: background-color 0.3s ease;
  text-align: center;
}

.btn:hover {
  background-color: var(--button-hover-bg);
}

.table {
  width: 100%;
  border-collapse: collapse;
  background-color: var(--card-bg);
  margin-top: 20px;
}

.table th,
.table td {
  padding: 12px;
  border: 1px solid var(--border-color);
  text-align: left;
}

.table th {
  background-color: var(--primary-color);
  color: #fff;
}

@media (max-width: 768px) {
  .container {
    padding: 10px;
  }
}
```

manager.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Панель управителя</title>
    <link rel="stylesheet" href="new_styles.css">
  </head>
  <body>
    <div class="container">
      <h1 class="header">Панель управителя</h1>

      <div class="card">
        <div class="card-header">Створення закупівлі</div>
        <form id="purchaseForm">
          <div class="form-group">
            <label for="category">Категорія товару</label>
            <select id="category" name="category_id"
class="form-control" required>
              <option value="">Виберіть категорію</option>
            </select>
          </div>
          <div class="form-group">
            <label for="amount">Сума (грн)</label>
            <input type="number" step="0.01" name="amount"
class="form-control" id="amount" required>
          </div>
          <div class="form-group">
            <label for="quantity">Кількість</label>
            <input type="number" name="quantity"
class="form-control" id="quantity" required>
          </div>
          <div class="form-group">
            <label for="description">Опис (опційно)</label>
            <input type="text" name="description" class="form-
control" id="description">
          </div>
          <button type="submit" class="btn">Створити
закупівлю</button>
        </form>
      </div>

      <div class="card">
        <div class="card-header">Ваші запити</div>
        <table class="table">
          <thead>
            <tr>
              <th>ID</th>
              <th>Категорія</th>
              <th>Сума</th>
              <th>Кількість</th>
              <th>Опис</th>
              <th>Статус</th>
            </tr>
          </thead>
          <tbody id="managerRequestsTable">
          </tbody>
        </table>
      </div>
    </div>

    <script>
      let categoriesMap = {};

      document.addEventListener('DOMContentLoaded',
async () => {
        try {
```

```
        const res = await fetch('/api/categories');
        const categories = await res.json();
        const categorySelect =
document.getElementById('category');
        categories.forEach(cat => {
          const option = document.createElement('option');
          option.value = cat.id;
          option.textContent = cat.name;
          categorySelect.appendChild(option);
          categoriesMap[cat.id] = cat.name;
        });
      } catch (error) {
        console.error(error);
      }

      fetchManagerRequests();
    });


document.getElementById('purchaseForm').addEventListener('
submit', async (e) => {
      e.preventDefault();
      const formData = new FormData(e.target);
      const data = {
        user_id: localStorage.getItem('userId'),
        category_id: formData.get('category_id'),
        amount: formData.get('amount'),
        quantity: formData.get('quantity'),
        description: formData.get('description')
      };

      try {
        const res = await fetch('/api/expenses', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify(data)
        });
        const result = await res.json();
        alert(result.message || result.error);
        e.target.reset();
        fetchManagerRequests();
      } catch (error) {
        console.error(error);
      }
    });

    // Функція для завантаження та відображення запитів
управителя
    async function fetchManagerRequests() {
      try {
        const currentUserId = localStorage.getItem('userId');
        if (!currentUserId) {
          alert("Ви не авторизовані!");
          return;
        }

        const res = await fetch('/api/expenses');
        const requests = await res.json();
        const managerRequests = requests.filter(req =>
req.user_id == currentUserId);
        const tableBody =
document.getElementById('managerRequestsTable');
        tableBody.innerHTML = '';
        managerRequests.forEach(req => {
          const row = document.createElement('tr');
          row.innerHTML = `
            <td>${req.id}</td>
            <td>${categoriesMap[req.category_id] ||
req.category_id}</td>
            <td>${req.amount}</td>
            <td>${req.quantity || 1}</td>
            <td>${req.description || ''}</td>
            <td>${req.status}</td>
          `;
          tableBody.appendChild(row);
        });
      } catch (error) {
        console.error(error);
      }
    }
  </script>

</body>
</html>
```

main.js

```
const registerForm =
document.getElementById('registerForm');
const loginForm =
document.getElementById('loginForm');
const expenseForm =
document.getElementById('expenseForm');
const expensesTableBody =
document.querySelector('#expensesTable tbody');

// Реєстрація
registerForm.addEventListener('submit', async (e) => {
  e.preventDefault();
  const formData = new FormData(registerForm);
  const data = {
    username: formData.get('username'),
    password: formData.get('password'),
    role: formData.get('role')
  };

  try {
    const res = await fetch('/api/auth/register', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });
    const result = await res.json();
    alert(result.message || result.error);
  } catch (err) {
    console.error(err);
  }
});

// Логін
loginForm.addEventListener('submit', async (e) => {
  e.preventDefault();
  const formData = new FormData(loginForm);
  const data = {
    username: formData.get('username'),
    password: formData.get('password')
```

```
    };

    try {
      const res = await fetch('/api/auth/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(data)
      });
      const result = await res.json();
      if (result.error) {
        alert(result.error);
      } else {
        localStorage.setItem('userId', result.user_id);
        localStorage.setItem('userRole', result.role);
        alert(result.message);

        if (result.role === 'owner') {
          window.location.href = '/owner.html';
        } else if (result.role === 'manager') {
          window.location.href = '/manager.html';
        } else {
          alert('Невідома роль користувача');
        }
      }
    } catch (err) {
      console.error(err);
    }
  });



// Створення витрати
expenseForm.addEventListener('submit', async (e) => {
  e.preventDefault();
  const formData = new FormData(expenseForm);
  const data = {
    user_id: formData.get('user_id'),
    category_id: formData.get('category_id'),
    amount: formData.get('amount'),
    description: formData.get('description')
  };

  try {
    const res = await fetch('/api/expenses', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });
    const result = await res.json();
    alert(result.message || result.error);
    getExpenses(); // Оновлення списку витрат
  } catch (err) {
    console.error(err);
  }
});

// Отримання списку витрат
async function getExpenses() {
  try {
    const res = await fetch('/api/expenses');
    const expenses = await res.json();
    renderExpenses(expenses);
  } catch (error) {
    console.error(error);
  }
}

// Відображення витрат у таблиці
function renderExpenses(expenses) {
  expensesTableBody.innerHTML = '';
  expenses.forEach(exp => {
    const row = document.createElement('tr');
    row.innerHTML = `
      <td>${exp.id}</td>
      <td>${exp.amount}</td>
      <td>${exp.description}</td>
      <td>${exp.category_id || ''}</td>
      <td>${exp.status}</td>
      <td>
        <button
onclick="deleteExpense(${exp.id})">Видалити</button>
      </td>
    `;
    expensesTableBody.appendChild(row);
  });
}

// Видалення витрати
async function deleteExpense(id) {
  try {
    const res = await fetch(`/api/expenses/${id}`, { method:
'DELETE' });
    const result = await res.json();
    alert(result.message || result.error);
    getExpenses();
  } catch (error) {
    console.error(error);
  }
}

document.addEventListener('DOMContentLoaded',
getExpenses);



index.html

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Вхід/Реєстрація</title>
  <link rel="stylesheet" href="new_styles.css" />
</head>
<body>
  <div class="container">
    <h1 class="header">Система обліку витрат</h1>

    <div class="card">
      <div class="card-header">Реєстрація</div>
      <form id="registerForm">
        <div class="form-group">
          <label for="regUsername">Логін</label>
          <input type="text" class="form-control"
name="username" id="regUsername" required />
        </div>
        <div class="form-group">
          <label for="regPassword">Пароль</label>
```

```
            <input type="password" class="form-control" name="password" id="regPassword" required />
          </div>
          <div class="form-group">
            <label for="regRole">Роль</label>
            <select class="form-control" name="role" id="regRole">
              <option value="owner">Домогосподар</option>
              <option value="manager">Домоуправитель</option>
            </select>
          </div>
          <button type="submit" class="btn">Зареєструватися</button>
        </form>
      </div>

      <div class="card">
        <div class="card-header">Вхід</div>
        <form id="loginForm">
          <div class="form-group">
            <label for="loginUsername">Логін</label>
            <input type="text" class="form-control" name="username" id="loginUsername" required />
          </div>
          <div class="form-group">
            <label for="loginPassword">Пароль</label>
            <input type="password" class="form-control" name="password" id="loginPassword" required />
          </div>
          <button type="submit" class="btn" style="background-color: var(--button-hover-bg)">Увійти</button>
        </form>
      </div>
    </div>

    <script src="main.js"></script>
  </body>
  </html>
```

db.js

```
const mysql = require('mysql2');

const pool = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
  port: process.env.DB_PORT,
  connectionLimit: 10
});

module.exports = pool.promise();
```

authController.js

```
const db = require('../models/db');
const bcrypt = require('bcrypt');

module.exports.register = async (req, res) => {
  try {
    const { username, password, role } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);

    await db.execute(
      'INSERT INTO users (username, password, role) VALUES (?, ?, ?)',
      [username, hashedPassword, role]
    );

    return res.status(201).json({ message: 'Користувача створено' });
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Серверна помилка' });
  }
};

module.exports.login = async (req, res) => {
  try {
    const { username, password } = req.body;
    const [users] = await db.execute(
      'SELECT * FROM users WHERE username = ?',
      [username]
    );
    if (users.length === 0) {
      return res.status(401).json({ error: 'Невірні облікові дані' });
    }
    const user = users[0];
    const match = await bcrypt.compare(password, user.password);
    if (!match) {
      return res.status(401).json({ error: 'Невірні облікові дані' });
    }

    return res.json({ message: 'Успішний вхід', role: user.role, user_id: user.id });
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Серверна помилка' });
  }
};
```

categoriesController.js

```
const db = require('../models/db');

module.exports.getCategories = async (req, res) => {
  try {
    const [rows] = await db.execute('SELECT * FROM categories');
    return res.json(rows);
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Серверна помилка' });
  }
};
```

expensesController.js

```
const db = require('../models/db');

module.exports.createExpense = async (req, res) => {
  try {
    const { user_id, category_id, amount, quantity, description } = req.body;
    await db.execute(
      'INSERT INTO expenses (user_id, category_id, amount, quantity, description) VALUES (?, ?, ?, ?, ?)',
      [user_id, category_id, amount, quantity, description]
    );
    return res.status(201).json({ message: 'Закупівлю створено' });
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Серверна помилка' });
  }
};

module.exports.getAllExpenses = async (req, res) => {
  try {
    const [rows] = await db.execute('SELECT * FROM expenses');
    return res.json(rows);
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Серверна помилка' });
  }
};

module.exports.updateExpense = async (req, res) => {
  try {
    const { id } = req.params;
    if (req.body.status && Object.keys(req.body).length === 1) {
      await db.execute('UPDATE expenses SET status = ? WHERE id = ?', [req.body.status, id]);
    } else {
      const { category_id, amount, quantity, description, status } = req.body;
      await db.execute(
        'UPDATE expenses SET category_id = ?, amount = ?, quantity = ?, description = ?, status = ? WHERE id = ?',
        [category_id, amount, quantity || 1, description, status, id]
      );
    }
    return res.json({ message: 'Закупівлю оновлено' });
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Серверна помилка' });
  }
};

module.exports.deleteExpense = async (req, res) => {
  try {
    const { id } = req.params;
    await db.execute('DELETE FROM expenses WHERE id = ?', [id]);
    return res.json({ message: 'Закупівлю видалено' });
  } catch (error) {
    console.error(error);
    return res.status(500).json({ error: 'Серверна помилка' });
  }
```