m-math_2025_013

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Факультет математики та інформатики
кафедра математичного моделювання

«Розробка вебдодатка для управління та обміну цифровою інформацією»

Кваліфікаційна робота

Рівень вищої освіти – другий(магістерський)

Виконав:
студент 2 курсу, 601 групи
Рощенюк Владислав Олександрович
Керівник:
кандидат фізико-математичних наук,
доцент Івасюк Г. П.

До захисту допущено

на засіданні кафедри
протокол № 5 від 26 листопада 2024 р.
Зав. кафедрою _____ проф. Черевко І.М.

Чернівці - 2024

Анотація

Під час написання кваліфікаційної роботи було досліджено тематику архітектурних рішень для масштабованих додатків, з використанням різноманітних мов програмування та проаналізовано сферу обміну цифрових товарів в мережі.

Реалізований додаток, покликаний створити додаткову можливість для маловідомих, в мережі інтернет осіб чи пересічних користувачів мережі, або ж, невеликих компаній представляти собою “автора” цифрової інформації і створити додаткове джерело доходу, чи просто отримати подяку від небайдужих людей за свою творчість, напрацювання, тощо.

Ключові слова: крос-платформа, обмін інформації, пожертвування

Abstract

During the writting of qualification work, the subject of architectral solutions for scalable applications was investigated using various programming languages and of digital products exchange on the web was analysed.

The implemented application is designed to create an additional opportunity for little-known people on the Internet or ordinary network users, or small companies to be the sphere “author” of digital information and create an additional source of income, or simply receive graditude from caring people for their creativity, achievements, etc.

Keywords: exchange, currency, cross-platform, information exchange, donations, currency, charity.

Кваліфікаційна робота містить результати власних досліджень. Викори - стання ідей, результатів і текстів наукових досліджень інших авторів мають посилання на інше джерело.

_____ Рощенюк В. О.

(підпис)

Зміст

Вступ

Шукаючи інформацію в мережі інтернет ми завжди знаходимо щось нове. Витрачаючи свій час в інтернеті ми вчимося, шукаємо інформацію, розважаємось, працюємо, спілкуємось з іншими людьми. Щось в мережі є безкоштовним, щось платним, щось “умовно безкоштовним”. Це може бути будь-що, що представляє інтелектуальну цінність. Плата за документ, плата за використання додатка, плата за послуги сервісів, тощо.

Не дивно, що деякі речі в мережі поширюються на платній основі. Людська діяльність завжди ґрунтувалась на внутрішніх мотивах. У кожного вони різні, - бажання досягти визнання, бажання заробити, реалізувати власні ідеї чи забезпечити собі краще майбутнє. Заохочення у вигляді матеріального винагородження, чи соціального визнання в суспільстві є потужним рушієм для подальшого розвитку особи. Цей стимул надихає людей рухатись вперед, вдосконалювати свої ідеї, які будуть рухати прогрес, створювати щось кардинально нове, що буде корисне людству.

І так як, основними мотиваційними чинниками для багатьох людей до творчості є слава та визнання серед суспільства, популярність, то створити онлайн платформу, де люди, які не є достатньо відомими, щоб співпрацювати з агентствами на контрактній формі змогли отримати винагороду за свої старання, визнання серед невеликих груп користувачів - може відкрити нову нішу в сегменті створення інтелектуальної власності та обміну нею між користувачами. Адже багато людей, які розповсюджують серед інтернет спільноти свої знання, творіння задарма - можуть теж отримувати “мотиваційні” подяки, щоб творити ще більше і примножувати людські надбання, чи відкривати щось кардинально нове.

Актуальність такої теми полягає в нинішньому ставленні людей до власних фінансів і мережі інтернет. Люди все більше довіряють технологіям і готові інвестувати не тільки в реальні речі а й ті, що існують і в мережі. Все більше людей залюбки би підтримали інших пожертвуванням чи оформили би підписку на постійний контент від певних авторів.

Метою цієї роботи є дослідження тематики платних послуг в інтернеті та розробки системи, яка б надала змогу реалізовувати покупку чи обмін інформації між користувачами на одному ресурсі.

Об’єктом дослідження в цій роботі є процес покупки чи продажу інтелектуальної власності між такими групами: автор - користувач.

Предметом дослідження є планування і розробка додатку, аналіз його продуктивності та доцільності використання обраної архітектури, взаємодії окремих елементів між собою.

Кваліфікаційна робота складається зі вступу, трьох розділів, списку використаних джерел та додатків. В першому розділі описані аналоги і тематика проєкту, в другому розділі вказані інструменти які використовувались під час розробки, в третьому розділі описано процес розробки та архітектурні рішення.

Загальний обсяг роботи складає 46 сторінок тексту та 10 рисунків.

Розділ 1. Аналіз предметної області і постановка завдання

1. Аналіз предметної області

Так як сервіси-майданчики для розміщення власного авторського контенту стають все більш популярними, то закономірно з’являється і ніша на ринку електронних ресурсів, де користувачі зможуть здійснювати віртуальні угоди на основах власних інтересів.

Як було визначено раніше, основна мета таких сервісів полягає в створенні простого і зручного інструменту взаємодії між двома ключовими ролями в обмінах грошей на будь - що. На майданчиках з продажу ці ролі являють собою покупця і продавця, а в контексті цього сервісу - споживача контенту (інформації) і її творця - автора, який водночас виступає і продавцем, тому що має змогу сам вирішувати за яким принципом і ціною буде розповсюджуватись його здобутки. Цей принцип може стати доволі популярний, і набирати популярність щороку.

Стрімкий розвиток інтернету і залучення багатьох людей до проводження свого часу займаючись пошуком інформації в мережі створив попит на електронні ресурси, які стали широко відомі по всьому світу і набрали стрімко велику кількість постійних користувачів. Це і пошукові браузери, і відеохостинги з безліччю цікавих відео від абсолютно різних людей зі всього світу. Це і соціальні мережі, де люди більш тісно могли обмінюватись повідомленнями і ділитись інформацією, і файлообмінники, які давали змогу користувачам завантажувати файли один одного, знаходячись на великих відстанях, так само і форуми, чи спільноти людей, різних професій, думок.

Ріст популярності цих ресурсів призвів до великих вкладень інвесторів і розвитку сфери комерції. Сотні тисяч маркетологів, продавців, менеджерів почали переносити свої бізнеси в мережу, створюючи нові тенденції і нові правила всередині. Розвиток цих правил перетворив мережу інтернет в великий ринок, де сучасні користувачі платять за “безкоштовну” інформацію або своїм часом, переглядаючи рекламу, або грошима, купуючи у сервісів доступ до зручного перегляду без реклами за місячну плату.

Цей принцип змусив багатьох користувачів шукати альтернативи в такій великій мережі інтернет, а кмітливих розробників - створювати нову концепцію додатків зііі зміненою системою, де б не було реклами, але за контент довелось би платити прозоро і без зобов'язань. Це і створило нову нішу серед електронних ресурсів, якааа залучалалала нових людей, які в свою чергу були тими ж самими користувачами, які вже були готові платити за розважальний чи повчальний контент, але прозоро і без реклами, а інші були авторами, які зневірились в великих платформах, які почали ставити все вищі й вищі вимоги, забираючи свободу творчості та велику комісію за посередництво.

Тепер, коли зрозумілі мотиви і причини, можна вияснити, що ж, в свою чергу, будуть отримувати користувачі? Автори, при переході на такі сервіси, отримують широкий функціонал для реалізації своєї творчості, максимально зручно і прозоро, прогнозовано, оскільки можуть приблизно розраховувати суми, які отримуватимуть, і знатимуть, що ніхто не привласнить собі право на їхні старання під час "випадкового" оновлення політики компанії, як це часто буває в великих корпораціях, які зародились на початку інтернету. Користувачі, ж, теж отримують прозору систему, в якій мають змогу споживати і купляти файли, фото, відео, текст, код в такій кількості, якій йому потрібно, а також вплинути на майбутні публікації авторів, проявивши активність в повідомленнях чи коментарях. Користувачі також отримують переваги у вигляді відсутності реклами, та чіткого контролю витрат. Користувач точно знає, що його кошти прямують точнісінько до автора, оминаючи інші витрати.

Як передбачається, ресурс матиме вигоду у вигляді невеликого збору під час виведення коштів на реальні фізичні гаманці користувачів. Тобто, якщо кошти будуть всередині ресурсу - ця частка не повинна стягуватись. Це має забезпечити заробіток сервісу.

1.2 Огляд аналогів та їх порівняння

Концепція подібних додатків не є новою. Вона зародилась ще на початку 2000-них років. Але набула своєї популярності і доцільності лише ближче до кінця 2010-тих років. Це зумовлено деякими чинниками. По перше - швидкий розвиток довіри людей до інтернет ресурсів та соціальних мереж, і в подальшому їх інтеграцію в буденне життя. Людство, в переважній більшості отримало доступ до інтернету настільки, що тепер це складає велику частину їхнього життя. Друга причина - це розвиток онлайн-банкінгу. Ця ніша активно розвивається і на теперішній час ми можемо оплачувати будь-які покупки натисканням однієї кнопки. По третє - люди почали витрачати свої гроші на послуги, яких раніше не існувало до появи інтернету в їх житті. Тепер це покупки саме через інтернет чи інтернет-підписки на сервіси, на блокувальники реклами, на проксі-сервіси тощо. Сукупність цих факторів дозволяє розвиватись подібним сервісам.

Для цього дослідження сформовано список з декількома сервісами для підтримки людей, авторів, які є найбільш відомими та популярними в країнах заходу. Незважаючи на все ще доволі низькі доходи населення і консерватизм українських користувачів щодо подібних ініціатив, ці сервіси популярні і в нашій країні.

Ось детальний список сервісів, для порівняння:

- **Kofi** (Хороший амбіційний проєкт),
- **BuyMeACoffee** (Довголітній лідер, спрощений функціонал),
- **Patreon** (Найбільш прогресивний сервіс, широкий функціонал),
- **Donatello** (український сервіс, націлений на пряму підтримку),
- **Kickstarter** (Сервіс, націлений лише на збори).

Для того, щоб зрозуміти переваги кожного з сервісів та попит на їх функціонал, їх потрібно розглянути детальніше і виділити їхні сильні і слабкі сторони, які роблять їх особливими. Тепер детальніше про самі сервіси.

Сервіс Ko-fi є відносно новим в цій сфері, але він водночас приносить

доволі багато цікавих функцій, які в майбутньому можуть змінити і інші сервіси. Основна філософія проєкту базується на добровільній підтримці і простоті користування. Кожен бажаючий може зареєструвати акаунт та вказати, на чому саме він хоче заробити. Це можуть бути прямі пожертвування в довільній сумі грошей, чи місячні підписки, щоб підтримати автора. Далі просто поділитись своїм посиланням на профіль у інших соціальних мережах.

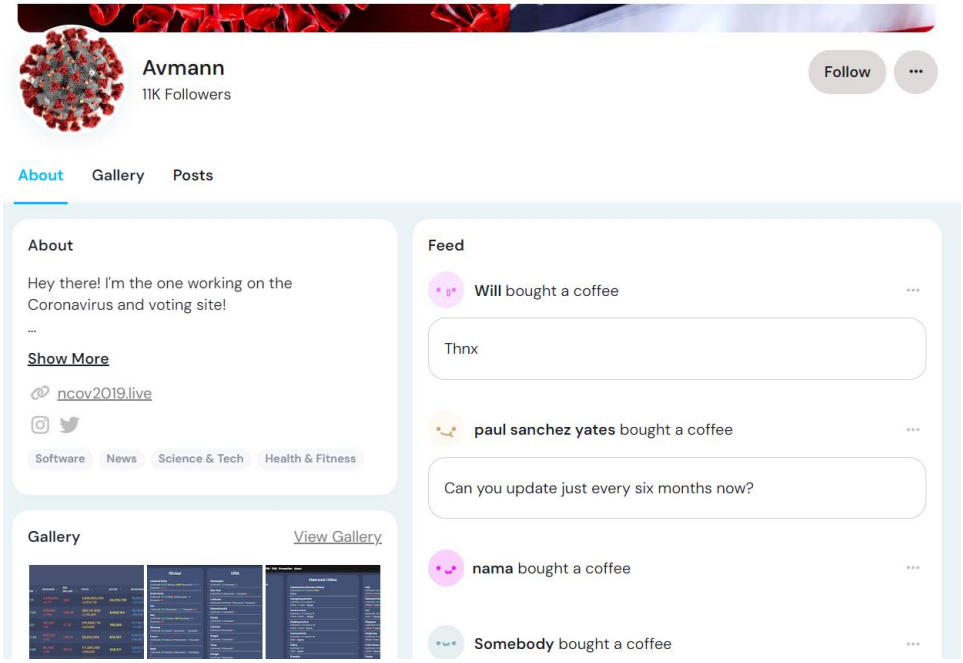

Рисунок 1.1: Вигляд сторінки користувача сервісу ko-fi

У користувачів є функціонал для того, щоб створювати колекції контенту, зображень, постів, є можливість створювати пости, де можна вести власний блог (рисунок 1.1). Проте жодних обмежень щодо доступу немає і це все автори повинні робити на добровільній основі.

Виходячи з цієї інформації, як для пересічного користувача можна виділити такі його переваги: простота використання, можливість вести блог, немає труднощів з виводом коштів чи поширенням профілю.

До недоліків платформи можна присвоїти такі особливості як: відсутність будь - яких налаштувань приватності свого блогу, невелике поширення і малу спільноту, обмежені можливості для росту.

Сервіс BuyMeACoffee (дослівно - ”купіть мені каву”) уособлює собою західну традицію купити людині каву в знак подяки за щось. Цей проєкт існує доволі давно на просторах мережі і відомий багатьом авторам контенту як зручний інструмент. Його основна ідея проста і полягає в прямій підтримці автора. Будь-хто може зробити пожертвування на довільну суму для автора вказавши в повідомленні свої побажання.

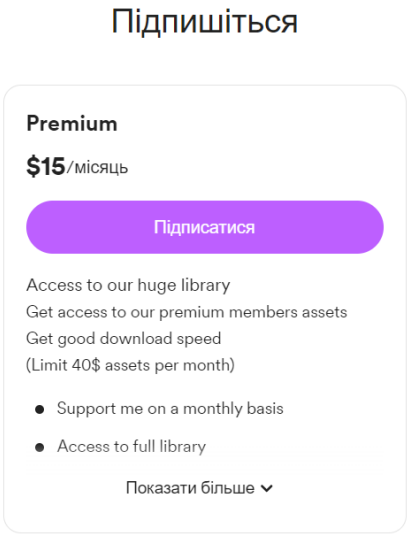

Рисунок 1.2: приклад підписки на сервісі BuyMeaCoffee

Наразі сервіс є платформою з великою спільнотою користувачів, де є можливість як створювати підписки на місяць (рисунок 1.2). Підписки дають перевагу розділяти свій контент на постійній безкоштовний чи платний основі. Сервіс має кращий функціонал для ведення блогу, ніж Ko-fi, оскільки тут вже присутні налаштування підписок за рівнем доступу.

Отже, переваги сервісу: велика спільнота, впізнаване ім’я бренду, налаштування підписок та постів для блогу, більший функціонал для автора.
З недоліків можна виокремити відсутність великого функціоналу для користувача та висока комісія зі сторони сервісу.

Сервіс Patreon наразі є вершиною прогресу серед західної та української спільноти. Він є одним з найпопулярніших і точно має найбільше можливостей для фільтрування розповсюдження власного контенту серед аудиторії. Також є додаткові інструменти для продажу окремих публікацій та гнучкого

налаштування видимості контенту. Такі можливості надають автору простір для створення нового контенту. А для користувачів, окрім доволі зручного інтерфейсу, є ще рекомендації авторів за інтересами чи спілкування з ними в публічному чаті.

Переваги платформи для поширення контенту Patreon: велика кількість інструментів для налаштування підписок та публікацій, велика спільнота користувачів, інноваційні функції, зручний інтерфейс, просування контенту всередині платформи.

До недоліків можна віднести високу комісію платформи за посередництво та обмеження щодо розміщення великих файлів.

Сервіс Donatello є українською адаптацією популярних західних аналогів. Він має функції прямої підтримки, але дуже обмежений в своєму функціоналі. Найближчі аналоги - це kofi та buymeacoffe. Користувачі сервісу можуть робити благодійні внески для авторів та використовувати їхній контент за підпискою чи без у вигляді публікацій. Але, це як правило, необов'язково, тому що основний фокус сервісу - пряма підтримка контенту без заохочення користувачів проводити свій час всередині платформи. Тобто, будь який автор може просто поділитись посиланням в інших своїх блогах, і автору не обов'язково вести свій блог ще й на сервісі donatello. Якщо описати діяльність вітчизняного сервісу коротко, то за своєю філософією він більше подібний до kofi, а функціонал аналогічний до buymeacoffee.

Для цього сервісу можна виокремити такі переваги як: зручність і простота використання, як автору, так і пересічному користувачу, інтеграція з місцевими банковими сервісами, широкий функціонал, який не поступається західним аналогам.

Недоліків суттєвих немає, окрім як використання партнерських платіжних систем, які мають часті перебої у роботі, і в загальному, стабільність сервісу є непостійною і користувачі часто стикаються з проблемами.

Сервіс Kickstarter відрізняється своєю концепцією підтримки. Основний акцент зосереджено на тому, щоб допомагати не тільки людям, а й групам,

компаніям збирати грошові засоби на певну ціль. Тобто набуття якогось початкового бюджету на створення освітнього сайту, чи відкриття бізнесу. Це і робить цей сайт особливим для різних користувачів які мають, мету створити свій стартап (англ. Startup - тобто компанія чи бізнес, яка знаходиться лише на своєму початку). Особливо поширеним цей продукт є у західних країнах та у Сполучених Штатах Америки.

Перевагами очевидно є унікальна система зборів. Вони мають опис, мету і тривалість. Автори зазначають свій план, для чого їм кошти, і як вони їх використовуватимуть, для того, щоб привернути увагу інвесторів в свій проєкт. Недоліки цього сервісу не можна оцінити об'єктивно зважаючи на його особливості. Він є кардинально відмінним від інших аналогів, але його функціонал є у своєму роді інноваційним і корисним для проектування додатка, який розроблений під час цього дослідження.

1.3 Постановка завдання

Порівнявши всі популярні сучасні сервіси для розповсюдження інформації чи послуг на платній основі, найбільш прогресивними і продуманими можна вважати Patreon та BuyMeACoffee за постійні оновлення та стабільність роботи, а також KickStarter за свою унікальну систему зборів. Тому докладний аналіз та розбір функціоналу цих сайтів дозволить спроектувати систему, яка буде стабільною і врахує всі можливості та функціонал, які є вдалими рішеннями на тому, чи іншому сервісі. Під час проектування варто враховувати основні важливі функції, такі як: функціонал підписок, покупок, зборів, функціонал для автора, і структуру сайту в загальному, яка спроектована для цієї тематики.

Завданням цього дослідження є детальний розбір функціоналу сайтів аналогічних сервісів, вивчення тематики побудови масштабованих додатків, аналіз та вибір архітектури додатка, яка найкраще впорається з поставленими для системи завданнями. Також, важливими темами цієї роботи є розробка програмного забезпечення та вирішення проблем та задач, які з'являються в процесі розробки.

Основне завдання додатка для обміну та управління цифровою інформацією є посередництво між авторами інформації чи послуг різних категорій творчості та спеціальностей і споживачами контенту, які є достатньо спроможними фінансово та є абсолютно довірливі до сучасних технологій і грошових операцій через мережу Інтернет.

Посередництво платформи (сервісу) реалізовано шляхом забезпечення грошових відносим між сторонами у різних формах власності. Сервіс не повинен претендувати на інтелектуальне право власності на будь-який вміст, ідеї, файли, чи думки авторів, але і не зобов'язується їх захищати від розповсюдження. Захист зі сторони сервісу полягатиме в персональному розповсюдженні контенту тільки після повної оплати за інформацію зі сторони користувача.

Системи, які забезпечують це посередництво, повинні мати декілька видів взаємодії для варіативності реалізації інформації:

- система підписок,
- система покупки окремого поста,
- система цілей,
- система прямої підтримки.

Система підписок повинна передбачати декілька варіантів підписок з гнучкими налаштуваннями, в яких автор може зацікавити користувача придбати її. Система покупки окремого поста надає доступ до інформації без обмежень після покупки. Пост - це одиниця інформації, яку автор додатково матиме змогу розповсюджувати за ціною разової покупки. Система цілей - підтримка автора задля виконання його цілей збору. Збори матимуть детальний опис, та бажану суму для отримання. Система прямої підтримки забезпечуватиме переказ прямих благодійних платежів на баланс користувача з повідомленнями, або без них.

Короткий опис цих систем уособлює загальну картину додатка, який є основним завданням цього дослідження. Користувач матиме широкий вибір систем задля благодійних внесків компаніям чи окремим авторам.

Програмне забезпечення повинно складатись з двох структур. Перша структура - це клієнтський додаток, з яким взаємодіятиме користувач. Він

здійснює відображення інтерактивного інтерфейсу в браузері та відправку даних на віддалений серверний додаток. Друга структура - це серверний додаток, який виконуватиме всі операції і повертатиме результати роботи назад в клієнтський додаток користувача.

Розділ 2. Опис використаних технологій

2.1 Середовище розробки WEB Storm та Rider

Під час розробки програмного забезпечення важливо використовувати інструменти, які забезпечують швидкість, якість, та інформативність розробки. Для цього в сучасному світі розробки існують два типи програм: текстові редактори та інтегровані середовища розробки (IDE).

Текстові редактори забезпечують інформативність під час написання коду та зручність розробки за допомогою вбудованої підсвітки синтаксису та терміналів, інколи зустрічається інтеграція інших інструментів розробника.

Інтегровані середовища розробки являють собою більш професійне ПЗ, яке має велику кількість функціоналу, яке суттєво пришвидшує розробку на всіх етапах. Переважна більшість з них підтримує функціонал для запуску додатків, підсвітку синтаксису для кращого розуміння коду, автоматичне заповнення коду для економії часу, інструменти версіонування коду, аналіз продуктивності, відстеження помилок в реальному часі тощо.

Для виконання завдання з розробки додатку мною було обрано два середовища розробки від компанії “JET Brains”. Ці середовища мають назву “JetBrains WebStorm” та “JetBrains Rider”. Вони є гнучко налаштовуваними та значно пришвидшують всі рівні розробки, надають високий рівень інформативності під час розробки та забезпечують всі потреби розробника.

Їхня гнучкість дозволяє використовувати лише одну версію IDE для розробки, але рішення використати різні середовища для різних задач, для яких вони першочергово були спроектовані, було правильним з точки зору економії часу.

2.2 Мова програмування C#

Мова програмування C# (укр. Сі-шарп) - є високорівневою мовою програмування, яка дозволяє створювати додатки будь-якого розміру та орієнтована для розробки програм для широко списку платформ та призначення. Мова C# є об’єктно-орієнтованою. Її можливості дозволяють створювати

додатки для мобільних пристроїв, настільні програми для комп'ютерів та обчислювальні додатки для вебсторінок. Мову C# також використовують для розробки складного програмного забезпечення на графічних та ігрових рушіях [3].

Підтримкою мови C# на момент створення додатка займається компанія Microsoft, і продовжує вносити покращення до синтаксису та розповсюджувати її разом з платформою для розробки .NET (детальніше див. пп. 2.4).

2.3 Мова програмування JavaScript

Мова програмування JavaScript (укр. джаваскрипт) - це мультипарадигмова мова програмування, яка існує вже доволі давно, але завжди залишається одною з найпопулярніших інструментів для розробки. В питаннях розробки вебсторінок є незамінною, оскільки тісно працює з браузерами і на її основі побудовано багато популярних інструментів, таких як: ReactJs, AngularJs, NextJs, Nuxt, VueJs, Jquerry та інші.

Можливості мови JavaScript дозволяють створювати повноцінні вебдодатки без використання інших мов завдяки стандарту Ecma Script 6 та рушію V8 від компанії "Google", який дозволяє компіляцію файлів формату JavaScript всесердині платформи NodeJs (детальніше див. розділ 2.5). Наразі розвиток мови відбувається за рахунок спільноти розробників і офіційно ніхто не розробляє нові версії і стандарти. Останній стандарт Ecma Script 6 вийшов ще в 2015 році. Наразі фокус на розвиток змістився в бік логічного продовження мови - TypeScript.

2.4 Платформа .NET

Платформа .NET - відкрита платформа для розробки програмного забезпечення з відкритим програмним кодом та підтримкою з боку компанії Microsoft. Підтримує розробку різними мовами програмування, але зосереджена саме на C# та F#. Платформа дозволяє створення вебдодатків різних розмірів, мобільних та крос-платформних додатків загального та специфікованого

направлення. Платформа має велике поширення і спільноту, яка адаптує різні інструменти для розробки, а також має величезну бібліотеку власних розширень для розробників.

2.5 Платформа NodeJS

Платформа NodeJs - це платформа, яка базується на рушії V8, який використовується для компіляції файлів JavaScript та TypeScript, що перетворює ці вузькоспеціалізовані мови, в мови загального призначення і не обмежує їх використання тільки в веббраузерах. Платформа підтримує різні типи та версії ОС, такі як: Windows, Linux, BSD, MacOS.

Платформа набуває великої популярності та має велику спільноту, яка створює безліч рішень і розширень для різноманітних задач (ресурс NPM). Додатки, які написані і підтримуються цією платформою, працюють у браузерах, мобільних пристроях, використовуються як настільні програми для ПК чи працюють на віддалених серверах.

2.6 Фреймворк React

Фреймворк React створений для швидкої розробки вебсторінок з високим функціоналом і акцентом на швидкодію. Він має унікальну технологію відображення графічних елементів та даних на вебсторінці [1].

Раніше, до появи цього фреймворку всі сторінки в браузері при взаємодії з ними або перезавантажувались або посилались на віддалений сервер, щоб відобразити інший HTML-документ. Ця технологія працювала повільно та постійно створювала деякі проблеми для користувачів. Постійне перезавантаження часто не зберігало попередньо введені користувачем дані чи змушувало більше очікувати завантаження, оскільки при натисканні якогось функціонального елементу на сторінці відбувався запит на сервер, і звідти відсилалась абсолютно така ж сторінка, тільки з невеликими змінами. Постійні запити навантажували серверні додатки, тому в 2013 році компанія Facebook представила розробникам нове рішення, а саме фреймворк React. Він був

покликаний вирішити цю проблему і додати сторінкам інтерактивності та швидкодії.

Фреймворк вирішує цю проблему постійних перезавантажень дуже розумним рішенням. Замість постійних змін в DOM-дереві браузера, він створює абсолютну копію наявного, і всі зміни на сторінці відбуваються саме на віртуальній копії елементів (рисунок 2.1) Таким чином, вдалось зменшити кількість запитів на серверні додатки і зменшити кількість будь-яких перезавантажень сторінки, оскільки React вносить зміни лише в тому елементі, де відбулась взаємодія.

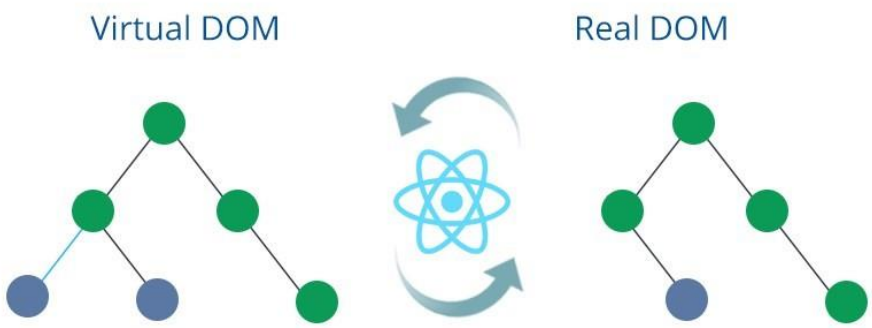

Рисунок 2.1: Принципи роботи з DOM в фреймворку React

Додаток задає кожному елементу стан (state) і при його зміні чітко розуміє, де саме були зміни стану компонента, перш, ніж відіслати данні. Цей же принцип відбувається при посиланні на інше посилання всередині сайту. React просто змінює стан елемента і відображає його як окрему сторінку, а не намагається отримати інший HTML документ.

Це робить фреймворк React одним з кращих інструментів на теперішній час, через його широкий функціонал, велику кількість сумісних розширень, безкоштовне поширення та відкритий вихідний код - будь хто може налаштувати його для себе.

2.7 Засіб автоматизації Docker

Коли перед розробниками та системними архітекторами постає питання розгортання додатка, то завжди з'являються проблеми з розгортанням, з поширенням на різних операційних системах, з автоматизацією додатку, з інструментами керування ним. Docker вирішує ці питання за допомогою унікальної технології.

Насамперед, Docker - це технологія автоматизації розгортання та обслуговування додатків. Її принцип полягає в запуску додатків в окремих ізольованих віртуальних середовищах. Ці середовища глибоко налаштовуються і легко керуються за допомогою спеціальних файлів - .Dockerfile. Докер-файли описують всі характеристики додатку, команди для запуску, команди для встановлення додаткових пакетів, внутрішні порти для комунікації і різні додаткові опції. Технологія докер створює віртуальний контейнер для додатку і виконує інструкції, описані в докер-файлі. Ці віртуальні контейнери - абсолютно ізольовані (рисунок 2.2) і не мають ніяких зв'язків, окрім як протоколи обміну повідомлень через мережу та AMQP-протокол (детальніше в розділі 2.8).

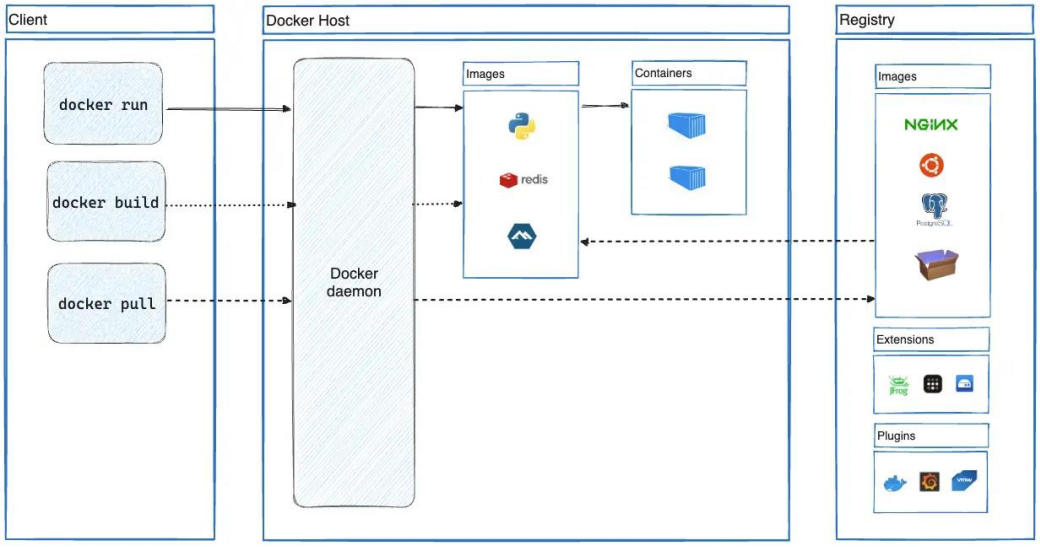

Рисунок 2.2: Архітектура системи з використанням Docker

Таке рішення є особливо корисним для створення системи, яка підтримуватиме принцип CI – CD (Continuous Integration – Continuous Delivery),

тобто, постійна (безперебійна) інтеграція - постійна доставка. Цей термін пояснює процес автоматичного тестування та автоматичного розгортання. А саме технологія докер є суміжною технологією з Kubernetes - рішення для управління контейнерами, через скрипти на різні ситуації.

Контейнерами легко керувати і оновлювати. Використання цієї технології дозволяє створювати безперебійні системи, у випадку складних мікросервісних додатків, та легко оновлювати код та функціонал. Це мінімізує постійні проблеми з розгортанням та доставкою додатка на різні операційні системи та пристрої, налаштування мережі та портів.

2.8 Протокол обміну внутрішніх повідомлень

Для реалізації великих додатків, які складаються з множити невеликих компонентів, які тісно пов’язані між собою не обійтись без використання додаткових протоколів обміну інформації та Шин повідомлень (Message Bus).

Великі масштабовані додатки - це ціла окрема система, яка складається з додатків невеликого розміру з своєю базою даних у кожного. Для постійної роботи вони потребують швидкого асинхронного і синхронного обміну даними між собою у лічені мілісекунди.

І у випадку спілкування клієнтських додатків з серверними використовується стандарт HTTP, то серверні додатки спілкуються між собою у форматі AMPQ при правильно спроектованій системі. Це рішення є найбільш оптимальним на сьогоднішній день.

2.9 Інструменти проектування

Для ефективного проектування додатка потрібно використовувати інструменти, які збільшують інформативність проєкту та зменшують час, витрачений на розробку діаграм, макетів, прототипів.

Щоб спроектувати найбільш важливі для розуміння діаграми (діаграми класів (рисунок 2.3), Use-Case діаграми) варто використати інструменти, які широко розповсюджені та перевірені часом. Оскільки, діаграми створюються на

початкових етапах проєкту, зміна стилізації чи технології може зменшити ефективність архітекторів та розробників, які щоразу будуть вивчати додатково нові стандарти розмітки, щоб прочитати діаграми. Саме тому варто обирати UML-стандарти, які широко використовуються по всьому світу і зрозумілі кожному менеджеру чи розробнику.

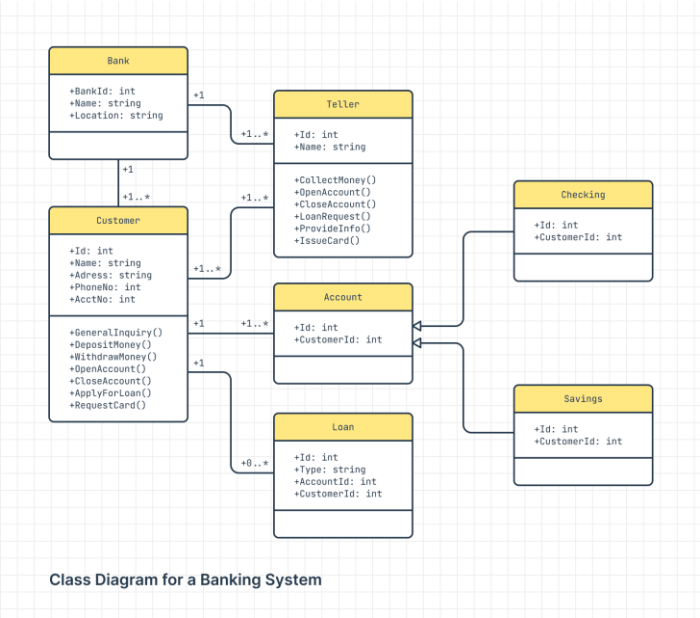

Рисунок 2.3: Приклад діаграми класів уніфікованою мовою UML

Для задач з проєктування UML-діаграм та простих схем використовується ПЗ “Star UML” та ресурс “draw.io”. Це полегшує процеси проєктування та дозволяє легко поширювати діаграми між розробниками у разі потреби.

Прототипування інтерфейсу вимагає інших інструментів. Популярними рішеннями в цій ніші є Adobe Photoshop та Figma. Вони вирішують задачі побудови інтерфейсу покроково з елементів, прописуючи їм точні розміри та властивості, такі як анімації чи зміна поведінки.

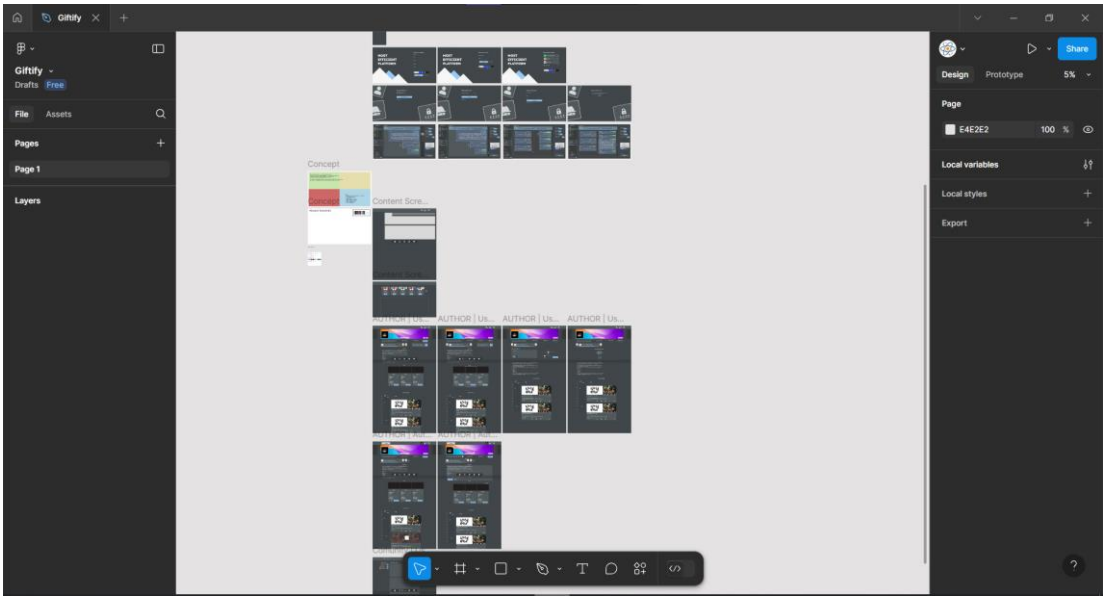

Рисунок 2.4: Інтерфейс додатка Figma під час проєктування

Для прототипування обрано саме інструмент Figma (рис. 2.4) через його простоту використання, функціонал та високу оптимізацію при великій кількості елементів. Ця програма дозволяє налаштовувати поведінку елементів і відображати ефекти, а також є режим розробника, який конвертує всі елементи з векторної графіки одразу в код. Photoshop не є чудовим вибором на сьогоднішній день з ряду причин. Найперша - це просто редактор фото, а не ПЗ, націлене на розробку інтерфейсів, друга - погана оптимізація і значна трата часу для побудови простих елементів.

Розділ 3. Планування та розробка додатка

3.1 Планування розробки додатка за етапами SLDC

Для ефективної розробки додатків в сучасному світі великі компанії та невеликі групи розробників притримуються циклу Software Development Life Cycle (SLDC). Дослівно перекладається як “життєвий цикл розробки програмного забезпечення.

Цей процес розбиває розробку на логічні етапи з чіткими вимогами до кожного етапу. Цей цикл дозволяє розробникам чітко визначити свою роль та задачі на кожному окремому етапі та ефективно спланувати наступні рішення.

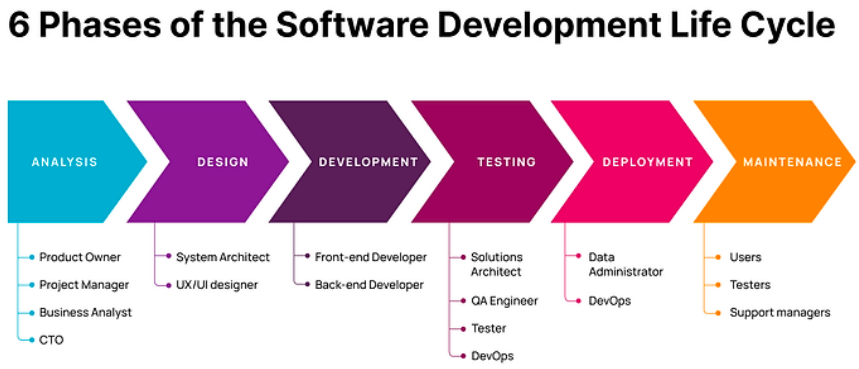

Рисунок 3.1: Основні етапи SDLC.

Цикл розробки ПЗ SLDC передбачає такі етапи створення додатків:

- аналіз,
- проєктування,
- розробка,
- тестування,
- розгортання,
- підтримка.

На етапі аналізу (Analisys) складається ряд вимог до майбутнього проєкту та описується ряд завдань для реалізації. На цьому етапі визначаються основна

ідея додатка, функції які повинні бути реалізовані в ході розробки, тип пристроїв на який він буде орієнтований, визначають мету проєкту та характеристику кінцевих користувачів. Результатом цього етапу є чіткий план дій та чітко описані вимоги до проєкту. Цей етап передбачає повний аналіз і прогнозування додатка, на основі предметної області та аналогів майбутнього проєкту.

Етап проєктування (Planning) являє собою формування вимог з попереднього етапу в чітко описані функції та вибір архітектури з детальним описом всіх елементів. Цей етап важливий для уникнення подальших проблем в ході розробки і є чітким планом, якого треба притримуватись для успішної реалізації. У контексті цієї роботи, вимоги, які були зібрані на етапі аналізу будуть обґрунтовано спроектовані у вигляді чіткої архітектури, відносно предметної області.

Етап розробки (Development) є найдовшим етапом, який передбачає розробку усіх необхідних елементів системи, оптимізацію коду. На етапі розробки проект набуває свого мінімально-працюючого вигляду, який згодом переходить на наступний етап для перевірки. Етап розробки є водночас найскладнішим етапом, оскільки на розробку деяких частин додатка можуть витрачатись великі часові проміжки та постійна зміна підходів і рішень. На етапі проєктування зарання вирішуються питання, пов’язані з архітектурою, технологіями, але не настільки досконало, щоб точно знати, який підхід чи метод буде краще використати. Постійний пошук рішень займає багато часу, але дозволяє на виході отримати код, який буде найбільш ефективним та займатиме невелику кількість ресурсів системи, а отже збереже кошти на обчислювальних операціях. У цьому проєкті етап розробки являтиме собою вибір оптимальних практик, створення основи додатка та поступове нарощування функціональних можливостей, поступове з’єднання окремих частин між собою і налаштування спілкування між ними.

На етапі тестування (Testing) вихідний код, ретельно перевіряється на дефекти та наявність критичних помилок. Це робиться для того, щоб кінцевий користувач не стикався з проблемами додатка, тому варто виявити їх завчасно і

усунути, перш ніж код буде зкомпільовано і доставлено до користувачів. Цей етап передбачає різні типи тестування, такі як: ручне, автоматизоване, тестування навантаження, та багато інших технік, які випробовують додаток в усіх його частинах, починаючи від елементів інтерфейсу користувача та закінчуючи найменшими елементами на стороні серверу. Цей етап при розробці допоможе виявити наявні недоліки поточного проєкту та швидко усунути їх, перш ніж код буде розгорнуто на сервері, де помилка буде вартувати великих затрат часу та ресурсів.

Етап розгортання коду (Deployment) являє собою процес компіляції вихідного коду та налаштуванні його на віддаленому сервері. Розгорнутий додаток стає доступний для користувачів і працює у безперебійному режимі, за умови, що на етапі тестування було усунуто всі помилки. Задля успішного розгортання потрібно правильно налаштувати середовище сервера, внутрішні шляхи до файлів, адреси, порти для запитів та постійно відслідковувати як працює додаток, а також обов'язково робити часті збереження даних, які називаються "бек-апи" (backup - англ. відновлення). Такі точки збереження даних бажано робити щодня на випадок непередбачуваних ситуацій. Для полегшення цього процесу і автоматизації цих задач використовуються практики CI - CD з використання сучасних рішень, таких як Jenkins та його аналоги в поєднанні з засобами автоматизації та контейнеризації Docker, що при вдалому налаштовуванні створюють автономну систему, яка у разі помилок може зберігати данні і запускатись без втручання спеціалістів.

Розгортання буде розглядатись в подальшому, як етап контейнеризації частин додатка та налаштування внутрішньої структури, шляхом послідовного запуску усіх частин проєкту та правильних налаштувань портів та адрес. Ризиками на цьому етапі є велика витрата ресурсів через неправильні властивості компонентів, яка може призвести до великих навантажень віддалених серверів, і в свою чергу - високі матеріальні витрати у вигляді плати за послуги зі сторони хостинг-сервісу (сервіси, які надають обчислювальні можливості)

Етап обслуговування і підтримки (Maintance) додатка являє собою процес,

який може тривати роками, і передбачає постійне виявлення проблем, активну підтримку користувачів та впровадження нового функціоналу. Тобто, коли додаток вже працює на віддаленому сервері, це не означає, що його життєвий цикл закінчено, а означає, що він все ще може отримувати зміни в роботі чи структурі, зміну інтерфейсу чи технологій, все ще, йде пошук помилок та їх усунення. Просто на цьому етапі, ці дії відбуваються уже віддалено і поступово. Додатково отримується зворотній відгук від кінцевих користувачів, які допомагають знайти недоліки чи запропонувати функціонал, який міг би вирішити їхню потребу.

Цей етап у контексті розробки додатка розглядатиметься лише частково, тому що, потребує певну кількість часу для збору додаткових вимог та списку помилок для їх вирішення.

3.2 Планування архітектури додатка

Для успішного виконання поставленого завдання роботи є важливим притримуватись принципів життєвого циклу розробки програмного забезпечення (розділ 3.1), оскільки визначено їх вплив на кінцеву стабільність додатка та ефективність під час розробки.

Перший етап процесу є збір вимог. Збір вимог до програмного забезпечення відбувається на основі вже поставленої задачі, або під час її визначення в ході перемовин. В реальних проєктах, де є дві сторони зацікавлених осіб, а саме замовника і компанії, яка надає послуги з розробки ПЗ, збір вимог відбувається шляхом прямої комунікації між замовником і керівництвом компанії та її менеджерами. В інших випадках - вимоги можуть бути сформовані іншим чином.

У контексті цього проєкту автор праці виступає в ролі замовника і в ролі сторони, яка реалізує це завдання Тому задача і інформація про додаток вже сформовані в аналізі предметної області (розділ 1.1) та поставленому завданні (розділ 1.3). На основі завдань до проєкту, порівняння аналогів (розділ 1.2) формується цілий ряд вимог, функціональних і нефункціональних для того, щоб

мати змогу правильно обрати архітектуру та технології.

Отже, під час аналізу предметної області та тематики додатка було сформовано наступний ряд вимог, які поділяються на функціональні та нефункціональні.

До функціональних вимог можна віднести такі критерії, які можна розділити на функціональні категорії. Найзручнішою і передовою практикою у плануванні та зборі функціональних вимог є використання User Stories сценаріїв користувача.

Формат User Stories передбачає опис вимог від користувача у вигляді: “користувач повинен чи передбачає” і далі опис конкретної вимоги. Цей підхід дозволяє краще зрозуміти, що саме потрібно кінцевому користувачу і спланувати додаток максимально ефективно.

В форматі сценаріїв опис вимоги відбувається від певної ролі. Наразі, передбачається дві ролі для системи:

- Користувач,
- Автор.

Користувач (User) - це зареєстрований, в системі користувач, який має змогу переглядати публікації, авторів, редагування власних даних, взаємодія з іншими користувачами шляхом листування та коментарів.

Автор (Author) - це зареєстрований користувач, який пройшов додатковий етап перевірки та реєстрації, і має змогу переглядати публікації, взаємодіяти з користувачами шляхом коментарів і листування, а також має змогу створювати публікації з інформацією та отримувати кошти.

До Опис функціональних вимог у вигляді User Stories відносяться наступні вимоги.

1. Реєстрація та авторизація. Вимоги до автентифікації користувача:
   - користувач повинен мати змогу авторизації і реєстрації акаунту за допомогою пошти та паролю;
   - користувач повинен передбачати можливості акаунту відносно ролі. Автор може створювати контент, користувач - споживати інформацію.

Тип акаунту повинен автоматично визначатись;

- користувач повинен мати змогу відновити доступ до його акаунту зручним способом, наприклад через електронну пошту;
- користувач повинен мати змогу змінити тип акаунту в будь-який момент;

2. Управління профілем користувача. Вимоги до профілю:
   - користувач повинен мати змогу редагувати свої облікові і публічні дані профілю в будь-який час;
   - користувач повинен мати змогу відслідкувати витрати;
   - користувач повинен мати змогу керувати активними підписками;
   - користувач повинен мати змогу бути проінформованим про взаємодії з кого акаунтом і отримувати сповіщення;
   - користувач повинен мати змогу залишати посилання на свої соціальні мережі.
3. Підписки та платний контент. Опис властивостей платних послуг:
   - користувач, у ролі автора повинен мати змогу редагувати опис підписок та їх ціну;
   - користувач, у ролі автора повинен мати змогу обирати, який тип користувачів бачитиме публікації;
   - користувач, у ролі автора повинен мати змогу редагувати данні профілю, відносно власних вимог;
   - користувач, у ролі автора повинен мати змогу продавати публікації;
   - користувач, у ролі автора повинен мати змогу отримувати статистику надходження коштів.
4. Соціальні вимоги. Взаємодія між користувачами:
   - користувач повинен мати змогу шукати контент за ім’ям автора чи ключовими словами;
   - користувач повинен мати змогу оцінювати публікації та писати коментарі;

- користувач, у ролі автора повинен мати змогу побачити, хто саме надсилає кошти, в якому розмірі та з яким повідомленням;
- користувач повинен мати змогу комунікації з іншими користувачами.

До нефункціональних вимог можна віднести наступні критерії.

1. Продуктивність. Технічні вимоги до системи та її ефективності:

- Припускається, що максимальна затримка між дією користувача та результатом обчислень не повинна перевищувати 200-250мс;

- Система повинна підтримувати стабільну роботу для одночасних 1000 користувачів на випуску і мати запас потужності до 5000 одночасних користувачів на майбутнє.

2. Надійність та безперебійність. Те, як система витримує великі навантаження.

- Система потребуватиме резервного копіювання даних на випадок збоїв в системі чи втрати даних;

- Використання асинхронної обробки повідомлень в системі повинна забезпечити швидкодію при навантажені;

- Автоматизація перезавантаження сервісів і логування рекомендована для системи.

3. Безпека. Вимоги, які описують стандарти безпеки сервісу.

- Використання JWT токенів, які матимуть обмежений час валідації для збільшення захисту кожного окремого акаунта, матиме ключову роль в безпеці сервісу;

- Шифрування особистих даних користувача та фінансових операцій повинне захищати важливі дані від перехоплення чи викриття їх, що забезпечуватиме ще одну ступінь захисту, що є важливим при використанні грошових операцій.

4. Масштабованість. Здатність системи до розширення і покращення.

- Система має побудована таким чином, щоб забезпечити впровадження нових елементів в систему без значних змін в структурі, для того, щоб мати змогу легко обслуговувати додаток і обновлювати її функціонал.

5. Сумісність. Критерій, який описує вимоги до коректної роботи на різних платформах і пристроях.

- Додаток повинен підтримувати найбільш популярні браузери і їх актуальні версії (Google Chrome, Firefox, Edge, Safari, OperaGX, Arc);
- Додаток повинен бути адаптованим під різні розміри екрану (16:9, 9:16) і платформи (Windows, Mac, Linux, Android, IOS).

Отримавши чіткий список вимог, спроектувати систему стає наступним важливим завданням, яке відноситься до процесу SLDC і передбачає вибір архітектури та інструментів.

Наступний етап процесу життєвого циклу - це обґрунтування і вибір архітектури додатку. Цей етап важливий для майбутньої стабільності додатку, його масштабованості та створення можливості для його майбутньої підтримки, що теж є етапом процесу життєвого циклу ПЗ.

Неправильний вибір архітектури в майбутньому принесе власнику великі витрати, складну підтримку і впровадження нового функціоналу, чи великі витрати на необов’язкові елементи.

При виборі архітектури варто керуватись такими критеріями:

- кінцеві користувачі,
- задачі проєкту,
- час,
- бюджет.

Важливо враховувати різні періоди активності і кількість користувачів, щоб знати, чи варто планувати велику систему, яка буде здатна витримувати великі навантаження.

Задачі проєкту і його мета визначають, які обчислювальні потужності і швидкодія припустимі для кінцевих користувачів, які основні функції важливо реалізувати і які ресурси знадобляться.

Які технології потрібно обрати і якого масштабу система потрібна, виходячи з наявного часу на розробку .

Вибір технологій та матеріального забезпечення також враховується

відносно бюджету на розробку, розгортання та підтримку кінцевого продукту.

Додаток з управління та обміну інформацією, повинен бути доволі великою одиницею, виходячи з вимог, які були описані. Це повинен бути додаток з значною кількістю логічних сторінок, кожна з яких передбачає немалий обсяг функціоналу, розділеного за його призначенням. За своєю тематикою, проєкт являє собою великий майданчик з обміну, продажу, створення оголошень, який має більш специфічне призначення, що, враховуючи досвід аналогічних сервісів потребує особливого рішення, та великих обчислювальних можливостей.

Великі додатки, як правило, використовують багато ресурсів для обчислення операцій та зберігання даних. І будувати майбутню систему потрібно на основі цих тверджень. Розраховуючи на успіх системи і популярність ресурсу, врахований досвід аналогів, вимоги та правила проєктування, слідують наступні архітектурні вимоги до системи:

1) система повинна бути безвідмовна і стабільна;
2) система повинна бути швидкою і витримувати великі навантаження;
3) система повинна бути придатною для покращення і обслуговування.

Під ці вимоги суворо не рекомендується використовувати будь - який тип монолітної архітектури. Зумовлено це високим навантаженням на потужності сервера, постійні відмови через перевантаження операціями, і відсутністю можливості до розширення функціоналу.

Найкращим рішенням в цій ситуації буде повне проєктування і використання мікросервісної архітектури. Це рішення передбачає використання невеликих проєктів, кожен з який має власну архітектуру, власне сховище даних, і невелику кількість функціоналу, що не перенавантажує систему.

Кожна одиниця (сервіс) формують собою цілу мережу, в якій кожна з них робить якусь одну дію, але в комплексі формують цілісний проєкт. Як правило, ці додатки мають один основний сервіс, і допоміжні, які менші за розміром і простіші у функціоналі. Основний сервіс керує логікою, спілкуючись з іншими сервісами, формуючи запити і відповіді [4].

Переваг у реалізації мікросервісної архітектури доволі багато. Оскільки окремі сервіси мають механізми розгортання, тестування, керування прямо всередині кожного з них, легко контейнеризуються та оркеструються. Вони розділяють всю бізнес-логіку на частини, щоб притримуватись принципу Single Responcibility (“одинична відповідальність”).

Для кожного сервісу окремо сплановано свою унікальну архітектуру, яка покриває його задачі та має простір для розширення функціоналу в разі потреби. Сервіси мають своє окреме сховище, яке децентралізоване, але забезпечує швидкодію у обчислювальних задачах окремого сервісу. Переважно, сервіси використовують “чисту архітектуру” (Clean Architecture) (рисунок 3.2) додатка [11].

Clean Architecture - це структура, яка створює окремі етапи керування інформацією, яке нагадує поступові шари, через що і отримала додаткову назву “Onion Architecture”.

Кожен шар відповідальний за окремий етап взаємодії з даними. Ці шари це “сутності” таблиць в базі даних, і обчислювальні сервіси і інтерфейси для зовнішньої взаємодії і контроллери для відловлювання запитів.

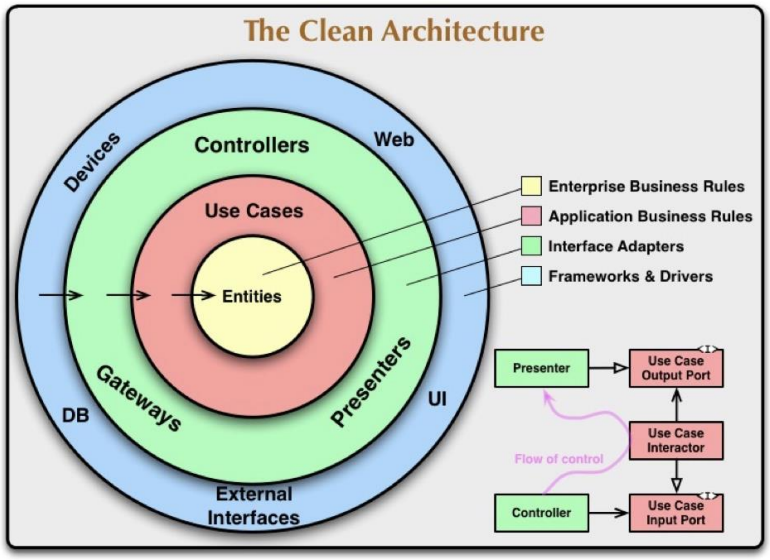

Рисунок 3.2: Принципи чистої архітектури додатка

Використання цієї архітектури в сервісах забезпечує високу надійність,

чітке розподілення коду на бізнес-логіку та інфраструктурні елементи, спрощене керування, знижує залежності між елементами, що створює більшу безвідмовність. Така архітектура дозволяє легше керувати додатком, вносити зміни та масштабувати проєкт, розширюючи функціонал.

До недоліків такого підходу можна віднести такі проблеми, як: складна реалізація на початкових етапах, де потрібно точно знати весь функціонал, велика кількість файлів і класів, яка потребує компетенції розробника.

Використання “чистої архітектури”, з використанням принципів Domain-Driven-Design і Dependency Injection, є найбільш правильним рішенням, яке було прийняте в ході порівняння з іншими популярними типами архітектур, які програють в багатьох аспектах, і мають менше переваг над чистою архітектурою.

Використання “чистої архрітектури” передчає створення таких структурних елементів і класів:

- Entity,
- DTO,
- Controller,
- Service,
- Repository (Interface).

Entity (сутність) - це клас, який уособлю собою бізнес одиницю в системі, навколо якої будується вся логіка і функціонал. Ними можуть виступати такі сутності, як “користувач”, “файл”, “транзакція”. Тобто якась одиниця, яка створює логіку.

DTO, або ж Data Transfer Object, - це файл - шаблон даних, для обміну даними між сервером і клієнтом, або між сервісами. Вони не містять в собі ніякої логіки, а містять лише данні і характеризуються специфічною структурою, яка залежить від вимог задачі, поставленої перед ними. Наприклад, якщо існує потреба в передачі якоїсь конкретної одиниці інформації, то DTO формуватиме з моделі вибірку лише за цим конкретним полем.

Controller (Контроллер) є точкою входу запиту в серверний додаток. Зустрічається часто і в інших типах архітектур і являє собою елемент, який

постійно прослуховує визначені раніше адреси і викликає потрібні методи, коли знаходить збіг між ними. Характеризується тим, що не має бізнес логіки в собі, а лише передає інформацію далі для обробки, викликаючи сторонні методи з сервісів.

Service (Сервіс) - це файли і класи для обробки бізнес логіки. Це шар, де реалізуються всі процеси і правила, які причетні до бізнес задач, але не належать до сутностей. Вони виконують операції, використовуючи репозиторії чи інші сервіси.

Repository (репозиторії чи інтерфейси) визначають правила, як повинні взаємодіяти інші класи та елементи системи, і є проміжками між компонентами, розділяючи зв’язність, що в породжує незалежність компонентів між собою і перевикористання коду.

3.3 Розробка клієнтської частини.

Розробка клієнтської частини проєкту починається з ініціалізації проєкту фреймворку React, шляхом виконання команди “npx create react-app <назва проєкту>” в терміналі. Далі, після ініціалізації і першого запуску йде повна структуризація папок, налаштування скриптів та бібліотек.

Архітектура проєкта для клієнтської частини теж має значення навіть в контексті одного фреймворку. Для цього проєкту дотримується архітектура логічної вкладеності, де всі логічні елементи, такі, як функції запитів до серверу, функції керування стану компонентів, та безпосередньо самі компоненти мають вкладену структуру.

Вкладена структура означає використання такого послідовного розміщення файлів, яке передбачатиме таке ж розміщення елементів і всередині DOM-дерева в браузері. Це не тільки поліпшує організованість проєкту, а, й пришвидшує час на розробку, тому що розробник завжди знає, де знаходиться елемент чи файл. Такий код водночас і легше підтримувати, на випадок, якщо над проектом працюватимуть інші розробники, які раніше не розробляли його.

Важливо переконатись, що на цьому етапі вже є готові макети UI та UX

дизайну, які задовольняють потреби майбутнього проєкту і є можливими для реалізації на обраних технологіях. Цей етап можна вважати виконаним, оскільки розробку макету було завершено ще на етапі планування. Макети покривали основні бізнес - завдання і вимоги, і відповідали всім основним принципам дизайну і були готові до наступних етапів.

Необхідність всіх макетів потрібна для повної розробки одразу всіх компонентів, розмітки функціональних елементів на їх місцях, і дотримання правил ідеальної верстки. Верстка за макетом передбачає розмітку, навіть без підключення запитів на сервер. Запити на сервер не є необхідними на ранніх етапах розробки, тому шо сервер не надає відповіді, оскільки сам ще не має логіки.

Для початку, створювалась основа для функціоналу, використовуючи синтаксис JSX, що являє собою поєднання мови HTML та JavaScript і дозволяє робити вебсторінки набагато функціональнішими. Всі данні для етапу розробки були попередньо заготовленні таким чином, щоб у випадку підключення до сервера ці данні відображались так, як це було задумано, без значного втручання в код, а під час розробки - просто допомагали бачити, як виглядатиме сайт в майбутньому.

Після виконання певного етапу розробки, сервер почав відповідати на запити, що вже дозволяло рухатись в бік розширення функціональності клієнтського додатка. Тепер фокус розробки змістився в бік функціональності і передбачав створення функціональних файлів, які б містили логіку запитів, з використанням бібліотеки Axios. Ця бібліотека, призначена для розширених налаштувань http-запитів мовою JavaScript і містить багато налаштувань. Використання цієї бібліотеки значно збільшило ефективність коду, в порівнянні зі стандартним методом fetch, який має свої недоліки.

Розширення логіки призвело процес розробки до помилок, які руйнували логічну роботу додатка, і унеможливлювали користування ним. Ці проблеми пов’язані зі нечерговістю запитів до сервера, та втратою даних, при оновленні сторінки. Вирішення цієї проблеми передбачає два шляхи, один з яких, це

використання вбудованого методу React Store або ж використання інструментів керування станом додатка. Одним з найпопулярніших і сучасних рішень є Redux - менеджер стану додатка і компонентів. Саме цей інструмент було обрано, на перевагу стандартним методам, які обмежені ресурсами і функціоналом. Використання такого менеджера стану дозволило вирішити проблеми черговості завантаження і створити збереження даних в пам’ять системи, під час використання [2].

Ще одним нововведенням до набору інструментів стало використання препроцесора стилів Less. Під час написання стилів для елементів звичайною мовою CSS виявилось, що коду стало багато, вага файлів збільшилась, і час їх завантаження знизив роботу додатка. Використання такого рішення, як Less зменшило об'єм коду, пришвидшило етап розробки і зменшило вагу файлу завдяки його особливостям. Цей препроцесор дозволяє перевикористовувати код, створювати змінні, які можна використовувати в будь-якому місці, і стандартизувати стилі.

Закінчивши розробку основних елементів було створено вебдодаток з широким функціоналом, який був готовий для подальших етапів впровадження та виправлення помилок. Технології, які були використані під час його написання дозволять продовжувати його підтримку і розробку нового функціоналу, використовуючи менше часу і ресурсів, що є великим плюсом і вдалою інвестицією в успішність кінцевого продукту.

3.4 Розробка серверної частини

Розробка серверного додатка являє собою найдовший етап, який починається з ініціації структури і підключення проєкту до системи контролю версій Git. Це забезпечить чітке версіонування і можливість змінювати код, з можливість повернення змін. Це особливо важливо, коли є потреба в невеликій зміні структури і при експериментах з окремими частинами коду.

Після цього, розробка почалась з створення і налаштування першого

сервісу, який відповідав би за всі данні користувача і операції над ним. Притримуючись плану проєкту, вся серверна логіка побудована на черговості запитів між сервісами, але є одна особливість, яка зумовлює створення сервісу користувача в першу чергу. Оскільки, весь додаток, побудований навколо взаємодії між користувачами, то жоден запит не обходиться без їх ідентифікації в системі. Тобто, кожен запит, різного ступеню важливості містить в собі логіку, яка передбачає що користувачі існують в системі і мають всі можливості для взаємодії, кожна їх дія відслідковується і записується в бази даних сервісів. Через це сервіс identity повинен будуватись першим.

Побудова сервісу відбувається шляхом ініціалізації проєкту і налаштування структури. Сервіс структуровано як “чисту” архітектруру, що означає створення таких елементів як: сервіси, контроллери, моделі, об’єкти обміну даних. Ці елементи забезпечують його логіку і обробку даних.

Функціональність цього сервісу передбачає обробку даних, пов’язаних з користувачами, а саме:

- створення нового акаунту користувача в системі,
- авторизація користувача за його даними,
- оновлення даних, зміна облікових даних,
- відновлення доступу за поштою при втраті облікових даних.

Реалізація цих функцій покриває основні вимоги до сервісу. Для побудови сервісу використано інструменти платформи .NET, які дозволяють створювати додатки, притримуючись архітектури та пришвидшити етап розробки, додатково зменшити кількість коду і збільшити його підтримуваність і читабельність. До цих інструментів відносяться Entity Framework Core, який включає в себе низку інструкцій і інструментів для роботи з базою даних. Сховище даних для цього сервісу - це реляційна БД під назвою SQL Server, яка поширюється компанією Microsoft [6].

Для реалізації системи авторизації, з активними сесіями користувачів, було використано технологію JWT токенів. Ця технологія дозволяє залишатись користувачам у статусі авторизованого акаунту впродовж певної кількості часу.

Особливість цих токенів полягає в постійному порівнянні ідентифікатору токену на сервері і клієнтському додатку, що виключає підміну даних і збільшує захист. При певній неактивності з боку користувача час токену спливає, і потребує повторного входу в систему.

За задачі пов'язані з обробкою публікацій відповідає сервіс контенту (content service). Цей елемент обробляє вхідні запити пов'язані з усіма типами публікацій, логікою взаємодії користувачів і постійно обмінюється інформацією з іншими частинами системи серверу.

Для розробки цього сервісу було створено проєкт на платформі .NET і спроектовано повну структуру з дотриманням принципів чистої архітектури, оскільки цей елемент є найскладнішим в усій структурі і є найбільш навантаженим. Він містить багато класів, але велика кількість файлів дозволяє розмежувати зони відповідальності коду і створити можливості для легкої підтримки коду і розширення його можливостей. Для чіткої маніпуляції даними використовується реляційна система "SQL Server", яка має синтаксис SQL і вже використовується іншими сервісами і показує високу стабільність у поєднанні з платформою .NET і Entity Framework.

Сервіс соціальної взаємодії (social service) забезпечує обробку і зберігання даних про соціальну активність, коментарі і сповіщення. Він поєднує в собі одразу дві важливі функціональності: розсилка сповіщень, їх обробка, зберігання та зберігання інформації про активність користувачів (коментарі та колекції публікацій). Сервіс так само притримується єдиної визначеної архітектури і зберігає данні в реляційній системі SQL Server.

Логіку обробки транзацкій забезпечує сервіс payment. Цей сервіс базується все ще на технологіях платформи .NET з використанням "чистої архітектури" і реляційної бази даних SQL Server. Зумовлено це важливістю для безпеки транзакцій і точних розрахунках. Мова C# обробляє такі операції бездоганно, а її об'єктно орієнтований підхід дозволить налаштувати код до найменших дрібниць.

Розроблений сервіс відповідає поставленим вимогам і дозволяє розробку

наступних елементів системи, а також притримується всіх сучасних стандартів безпеки, використовуючи додаткові методи захисту і відновлення доступу.

Наступний елемент в системі це поштовий сервіс. Він відповідає за розсилання повідомлень на електронні пошти користувачів. Сервіс виконує багато задач, пов'язаних як з інформуванням користувачів про акції, так і бере участь при відновленні доступу чи оповіщенні користувачів про його дії. Цей сервіс тісно взаємодіє з іншими, і не має прямої взаємодії з користувачем, тому що є внутрішнім. Це означає, що всі його адреси використовуються лише іншими сервісами.

Сервіс виконує задачі з розсилки повідомлень, які потребують сервіси ідентифікації для відновлення паролів і даних, сервіс оплати для надсилання звітностей про підписки і платежі, сервіс контенту для оповіщення про нові публікації і взаємодії користувачів.

Сервіс базується на платформі .NET з використанням сторонніх технологій. Архітектура сервісу встановлена при плануванні як "чиста", і розроблена згідно вимог сервісу. Сторонніми технологіями є SMTP сервер, для розсилки повідомлень через пошту.

Цей інструмент дозволяє керувати повідомленнями прямо з коду і спрощує процес надсилання повідомлення, надаючи для цього необхідні інструменти. Все, що потребує сервер SMTP для належної роботи - це його мінімальне налаштування і акаунт поштового сервісу. Акаунт теж потребує певних налаштувань. В випадку використання акаунта Google, потрібно налаштувати поштовий сервер як сторонній додаток, який має доступ і права, для розсилки, і захистити акаунт двох факторною автентифікацією.

Використання цих інструментів дозволило створити сервіс, який здатний до розсилки повідомлень, на різні випадки. І хоч користувач не має прямого доступу до адрес сервісу, він взаємодіє з ним постійно через інші. Розробка цього сервісу забезпечила додаткові шляхи взаємодії з аудиторією сайту.

Останнім в пріоритеті, але не менш важливим елементом є сервіс для обробки персональних повідомлень і групових чатів. Він має таку ж назву - chat

service. Цей сервіс відповідає за обмін повідомлень між користувачами в реальному часі.

Так як, у питанні обміну повідомлень важливим аспектом є швидкодія обробки повідомлень, звичний набір технологій не підходить. При плануванні було визначено, що сервіс повинен бути швидким, й водночас не витрачати багато системних ресурсів. Саме тому він базується на технологіях платформи NodeJs. Серверну логіку обробляє розширення Express, а сховище даних - це нереляційна система MongoDB, яке підходить в контексті цього сервісу і чудово працює з в інструментами платформи NodeJs. Сервіс має іншу платформу, але все ще базується на чистій архітектурі, хоч із деякими відмінностями від її загальних принципів. Звичний протокол запитів http конкретно в цьому випадку не є доречним, тому використовується технологія веб-сокетів [5].

Основна відмінність і перевага використання веб-сокетів над http-запитами полягає в самому принципі спілкування клієнта та сервера (рисунок 3.3). За замовчуванням, вебсторінки працюють у режимі очікування.

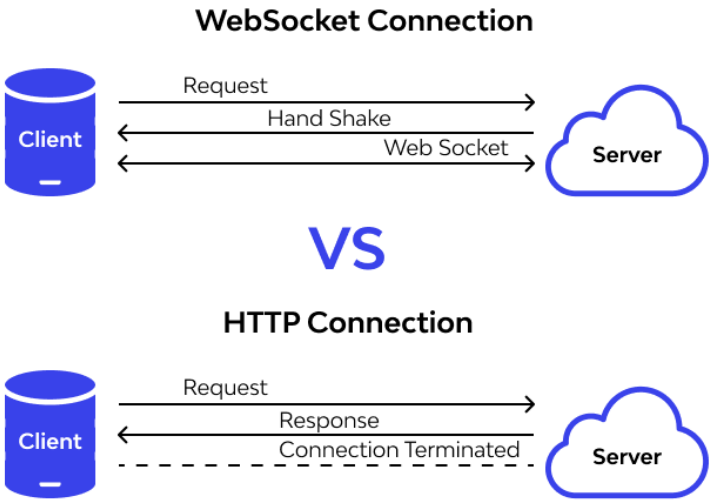

Рисунок 3.3: Переваги веб-сокетів над http-запитами.

Тобто, коли користувач натискає кнопку чи викликає функцію запиту, то на сервер, за посиланням надходить повідомлення, яке містить в собі тип запиту

(GET, POST, PUT, PATCH, DELETE), далі адресу, за якою знаходиться сервер і тіло запиту, яке містить данні користувача. Потім клієнтський додаток очікує відповіді від сервера, створюючи чергу, де наступний етап неможливий без іншого. Цей формат спілкування є одноразовим для системи, оскільки взаємодія з сервером припиняється одразу після отримання відповіді.

Веб-сокет, в свою чергу, це кардинально інший підхід обміну повідомлень. З'єднання з сервером створюється і не припиняється, поки цього не потребує ситуація. В час відкритого з'єднання відбувається обмін повідомлень в реальному часі, з найбільшою ефективністю і швидкодією.

Використання технології веб-сокетів вирішує поставленні перед сервісом вимоги, шляхом обміну повідомлень в режимі реального часу. На прикладі спілкування користувачів, це працюватиме таким чином:

- користувачі під'єднуються до сервера коли переходять на сторінку чатів, створюється з'єднання;
- користувачі обмінюються повідомленнями в чатах, іде обмін повідомлень і їх обробка сервером в режимі реального часу;
- користувачі переходять на іншу сторінку чи покидають чат. З'єднання з сервером закривається.

Цей принцип і підхід, дозволив побудувати максимально швидкий сервіс для обміну повідомлень, а ресурси і можливості платформи NodeJs, дозволяють витримувати одночасні великі навантаження, хоч і передбачається, що кількість одночасних користувачів, які будуть використовувати час, не є значною.

Вхідною точкою до кожного з сервісів є структурний елемент Api-gateway. Він приймає всі запити з клієнтського додатка і маршрутизує їх вже до конкретних сервісів. В випадку великої кількості сервісів використання різних портів і адрес може легко пошкодити роботу додатку і призвести до великої кількості помилок.

Структура API Gateway вирішує ці проблеми і зводить всі маршрути до одного порту та стандартизує API всередині додатка, так як виступає посередником між всіма сервісами і чітко визначає порти сервісів, їх шляхи і

типи запитів, які надходитимуть на конкретний сервіс.

Рисунок 3.4: Налаштування сервісу структурою OcelotAPI

Для реалізації цього сервісу використовується проєкт на платформі .NET з використанням інструменту OcelotAPI, який має багато налаштувань і не вимагає складної структури. Для використання в проєкті OcelotAPI достатньо лише установити в проєкт це розширення і створити json файл, в якому прописати налаштування в визначеній структурі (рисунок 3.4), В налаштуваннях можна контролювати адреси, порти, методи запитів, а ще обмежити кількість запитів та встановлювати перерви на використання, щоб не перевантажувати сервіси (див. додаток 1.1). Ocelot не вимагає складної структури проекти чи чіткої архітектури, достатньо лише правильно налаштувати файл Program.cs (точка входу в програму) і прописати налаштування в json файлі ocelot [8].

Таким чином було розроблено функціонал серверного додатку, який забезпечує обробку даних користувача шляхом злагодженої роботи сервісів.

3.5 Тестування додатка

Тестування таких великих додатків потребує кілька типів і рівнів тестувань. Але результати дозволили виявити недоліки в якості додатка і виправити помилки до того, як код буде доставлено на сервери, і всі проблеми виявились вже в процесі користування.

Для тестування клієнтської частини

Для тестування серверної частини було використано багато типів тестувань з різними значеннями, які дали позитивний результат і інформацію щодо загальної якості додатка, та виправлень, які необхідно виправити негайному порядку, щоб забезпечити стабільність. Зокрема, під час тестування серверних додатків за їхніми шляхами, надсилаючи запити в інтерфейсі програмного забезпечення Postman та Swagger, яке призначено для цих задач і дозволяє імітувати запити, було виявлено декілька прогалин в безпеці даних, в сервісах ідентифікації та транзакцій. Тестування було припинене до моменту усунення цих помилок.

Також під час тестування швидкодії сервісу при навантаженні, було виявлено великий вплив і недоліки http-запитів між сервісами в REST архітектурі мережі. Тому було прийнято рішення про прискорення роботи додатка шляхом зміни звичних запитів на постійний обмін повідомлень по шині інструмента RabbitMQ. Тепер сервіси спілкувались між собою у швидкому форматі AMPQ, який забезпечує інструментарій RabitMQ. Також, при поступовому збільшенні навантаження можливе використання елементу “load balancer” який би розподіляв запити і їх кількість на шлюз API Gateway [7].

Після повного завершення тестування було внесено зміни до архітектури та виправлені основні помилки, які б не дозволяли користуватись сервісом в повній мірі.

3.6 Розгортання додатка

Розгортання додатку це процес, який включає декілька етапів, і є різним для клієнтської частини і для серверної. Розгортання додатків тягне за собою витрати, тому важливо контролювати кількість ресурсів. Наприклад, для економії ресурсів одного додатка, було поєднано функціонал двох сервісів в один, який тепер виконує одразу декілька функцій. Це допоможе зекономити ресурси віддаленого сервера і зменшити кількість витрат, зберігаючи високу продуктивність.

В тестовому режимі клієнтський додаток було розгорнуто на платформі GitHub за технологією GitHub Pages, яку підтримує цей сервіс. Для етапу розробки це припустимо, але для повноцінної роботи додатка на етапі його випуску на загальний огляд доведеться змінити потужності і інший сервіс.

Незважаючи на обмежену потужність, додаток працює на сторонньому сервісі. Під час розробки це є найкращим варіантом, оскільки оновлення та розгортання не займають багато часу і ресурсів чи специфічних знань в цій галузі. Робиться це в автоматичному режимі на серверах GitHub.

Для розгортання серверного додатку використати ресурс GitHub Pages не буде вдалим рішенням, через обмежені обчислювальні можливості і концепцію додатка. Сам додаток передбачає окреме розгортання всіх контейнерів на ресурсі, який розрахований на такі дії і підтримує docker контейнери, які були використані як для виконання вимог так і для зручності.

Використання Docker контейнерів необхідне рішення у додатках такого типажу. Для кожного сервісу було прописано список команд і налаштувань для створення контейнера. У цих контейнерах, додатки мають власне ім’я, власну адресу і власний порт доступу.

Важливо розуміти, що розгортання та запуск кожного контейнеру має відбуватись у певній послідовності, яку має щось контролювати. На рішення цього питання відповідає файл Docker-Compose, в якому можна чітко визначити послідовність запуску контейнерів і їх додаткові налаштування (див. дод. 1.3).

Після розгортання всіх контейнерів з додатками і їх базами даних, для безперебійної роботи вартує змінити налаштування клієнта і перевірити справність. Надалі сервіс готовий для користування.

Вихідний код додатка є повністю відкритим для користувачів мережі, і доступний кожному, тому що завантажений на віддалений репозиторій [9-10].

Висновок

В ході дослідження тематики цієї роботи, яка орієнтована на розробку і аналіз сервісів, які надають послуги для користувачів з продажу і покупкою їхніх власних напрацювань, було розроблено додаток, який має обширний функціонал і працює безвідмовно завдяки його структурі.

Під час написання роботи було отримано ключові знання, які можуть бути застосовані в різних типах задач, які пов’язані з програмуванням чи управлінням проєктами. Оскільки, під час розробки було використано особливості, які належать до таких ролей, як розробник, системний архітектор, менеджер проєктів, тестувальник тощо. Здобуті навички допоможуть у покращенні проєкту, створенні шляху його подальшого розвитку чи при іншій важливій ситуації.

Список використаних джерел

1. “Документація по фреймворку React”. Електронний ресурс
   URL: Quick Start – React
2. “Навчальний посібник по інструменту Redux”. Електронний ресурс.
   URL: Getting Started with Redux | Redux
3. “Офіційна документація по мові C#”. Електронний ресурс.
   URL: C# docs - get started, tutorials, reference. | Microsoft Learn
4. “Архітектура контейнеризованих додатків на платформі .NET”. Офіційний електронний ресурс від Microsoft
   URL: .NET Microservices. Architecture for Containerized .NET Applications
5. “Базове налаштування шляхів в Express” Офіційний електронний ресурс.
   URL: Express basic routing
6. “Центр документації по Entity Framework”. Електронний ресурс Microsoft
   URL: Entity Framework documentation hub | Microsoft Learn
7. “Налаштування Message broker RabitMQ”. Електронний ресурс.
   URL: RabbitMQ tutorial - Work Queues | RabbitMQ
8. “Налаштування шлюзів Ocelot API Gateway”. Електронний ресурс документації від Microsoft.
   URL: Implementing API Gateways with Ocelot - .NET | Microsoft Learn
9. “Репозиторій з відкритим вихідним кодом клієнтського додатка”. Електронний ресурс.
   URL: GitHub | Gift.fy - Gift For You. Project client app repository
10. “Репозиторій з відкритим вихідним кодом серверного додатка”. Електронний ресурс.
    URL: GitHub | Gift.fy - Gift For You. Project main repository
11. “Типи архітектур серверних додатків”. Електронний ресурс.
    URL: Common web application architectures - .NET | Microsoft Learn

Додатки

1. Додатки серверної частини проєкту.

Додаток 1.1 API GATEWAY - OcelotAPI

Ocelot.json

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/user/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 6050 }
      ],
      "UpstreamPathTemplate": "/identity/user/{everything}",
      "UpstreamHttpMethod": [ "POST", "GET", "PUT", "DELETE" ]
    },
    {
      "DownstreamPathTemplate": "/content/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 6060 }
      ],
      "UpstreamPathTemplate": "/content/{everything}",
      "UpstreamHttpMethod": [ "POST", "GET", "PUT", "DELETE" ]
    },
    {
      "DownstreamPathTemplate": "/payment/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        { "Host": "localhost", "Port": 6070 }
      ],
      "UpstreamPathTemplate": "/payment/{everything}",
```

```
    "UpstreamHttpMethod": [ "POST", "GET", "PUT", "DELETE" ]
  },
  {
    "DownstreamPathTemplate": "/social/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
      { "Host": "localhost", "Port": 6080 }
    ],
    "UpstreamPathTemplate": "/social/{everything}",
    "UpstreamHttpMethod": [ "POST", "GET", "PUT", "DELETE" ]
  },
  {
    "DownstreamPathTemplate": "/mailing/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
      { "Host": "localhost", "Port": 6100 }
    ],
    "UpstreamPathTemplate": "/mailing/{everything}",
    "UpstreamHttpMethod": [ "POST", "GET", "PUT", "DELETE" ]
  },
  {
    "DownstreamPathTemplate": "/chat/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
      { "Host": "localhost", "Port": 7070 }
    ],
    "UpstreamPathTemplate": "/chat/{everything}",
    "UpstreamHttpMethod": [ "POST", "GET", "PUT", "DELETE" ]
  }
],
"GlobalConfiguration": {
  "BaseUrl": "http://localhost:5000"
}
```

```
}
```

Program.cs

```
using Ocelot.DependencyInjection;

using Ocelot.Middleware;


var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddJsonFile("ocelot.json", optional: false,
reloadOnChange: true);

builder.Services.AddOcelot();


var app = builder.Build();

app.UseOcelot().Wait();

app.Run();
```

Додаток 1.2 Приклад сервісу на платформі .NET (content-servie)

PostController.cs

```
namespace content_service.Controllers;

[ApiController]
[Route("api/posts")]
public class PostController : ControllerBase
{
    private readonly PostService _postService;
    private readonly HttpClient _httpClient;

    public PostController(PostService postService, IHttpClientFactory
httpClientFactory)
    {
        _postService = postService;
        _httpClient = httpClientFactory.CreateClient();
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetPostById(int id, [FromQuery] Guid
userId)
    {
        var post = await _postService.GetPostByIdAsync(id, userId);
        if (post == null)
            return NotFound("Post not found.");
        return Ok(post);
    }
```

```
    private async Task<Guid?> GetUserIdByEmailAsync(string email)
    {
        var response = await
_httpClient.GetAsync($"/identity/user/GetUserProfile/profile?email={email}
");
        if (!response.IsSuccessStatusCode) return null;

        var userProfile = await
response.Content.ReadFromJsonAsync<UserProfileDto>();
        return userProfile?.Id;
    }

    [HttpGet]
    public async Task<IActionResult> GetAllPosts([FromQuery] string email)
    {
        var userId = await GetUserIdByEmailAsync(email);
        if (userId == null) return Unauthorized("User not found.");

        var posts = await _postService.GetAllPostsAsync(userId.Value);
        return Ok(posts);
    }

    [HttpPost]
    public async Task<IActionResult> CreatePost([FromBody] CreatePostDto
dto, [FromQuery] string email)
    {
        var userId = await GetUserIdByEmailAsync(email);
        if (userId == null) return Unauthorized("User not found.");

        var post = await _postService.CreatePostAsync(dto, userId.Value);
        return CreatedAtAction(nameof(GetPostById), new { id =
post.Id.ToString() }, post);
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdatePost(int id, [FromBody]
UpdatePostDto dto)
    {
        var updatedPost = await _postService.UpdatePostAsync(id, dto);
        if (updatedPost == null)
            return NotFound("Post not found.");
        return Ok(updatedPost);
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeletePost(int id)
    {
        var result = await _postService.DeletePostAsync(id);
        if (!result)
            return NotFound("Post not found.");
        return NoContent();
    }
```

```
    [HttpPost("{id}/like")]
    public async Task<IActionResult> LikePost(int id)
    {
        var result = await _postService.LikePostAsync(id);
        return result ? Ok("Post liked.") : NotFound("Post not found.");
    }
}
```

AppDbContext.cs

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) { }

    public DbSet<Post> Posts { get; set; }
    public DbSet<Goal> Goals { get; set; }
    public DbSet<FileAttachment> FileAttachments { get; set; }
    public DbSet<UserSubscription> UserSubscriptions { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>().HasMany(p =>
p.FileAttachments).WithOne(f => f.Post).HasForeignKey(f => f.PostId);
        base.OnModelCreating(modelBuilder);
    }
}
```

Models – Post.cs

```
public class Post
{
    public int Id { get; set; }
    [Required]
    [MaxLength(100)]
    public string Title { get; set; }
    public string Content { get; set; }
    public Guid AuthorId { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public bool IsPublic { get; set; }
    public int Likes { get; set; }
    public ICollection<FileAttachment> FileAttachments { get; set; } = new
List<FileAttachment>();
}
```

Dto – PostDto.cs

```
public class PostDto
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime CreatedAt { get; set; }
```

```
    public bool IsPublic { get; set; }
    public int Likes { get; set; }
    public List<int>? FileIds { get; set; }
}
```

PostService.cs

```
public class PostService
{
    private readonly PostRepository _postRepository;
    private readonly UserSubscriptionRepository _subscriptionRepository;
    private readonly IMapper _mapper;

    public PostService(PostRepository postRepository,
UserSubscriptionRepository subscriptionRepository, IMapper mapper)
    {
        _postRepository = postRepository;
        _subscriptionRepository = subscriptionRepository;
        _mapper = mapper;
    }

    public async Task<PostDto?> GetPostByIdAsync(int id, Guid userId)
    {
        var post = await _postRepository.GetPostByIdAsync(id);
        if (post == null) return null;

        bool hasAccess = await
_subscriptionRepository.IsSubscribedAsync(userId, post.AuthorId);
        if (!post.IsPublic && !hasAccess) throw new
UnauthorizedAccessException("Access denied.");

        return _mapper.Map<PostDto>(post);
    }

    public async Task<List<PostDto>> GetAllPostsAsync(Guid userId)
    {
        var posts = await _postRepository.GetAllPostsAsync(userId);
        return _mapper.Map<List<PostDto>>(posts);
    }

    public async Task<PostDto> CreatePostAsync(CreatePostDto
createPostDto, Guid userId)
    {
        var post = _mapper.Map<Post>(createPostDto);
        post.AuthorId = userId;
        post.CreatedAt = DateTime.UtcNow;
        await _postRepository.AddPostAsync(post);
        return _mapper.Map<PostDto>(post);
    }

    public async Task<PostDto?> UpdatePostAsync(int id, UpdatePostDto
updatePostDto)
    {
```

```
        var post = await _postRepository.GetPostByIdAsync(id);
        if (post == null) return null;

        _mapper.Map(updatePostDto, post);
        await _postRepository.UpdatePostAsync(post);
        return _mapper.Map<PostDto>(post);
    }

    public async Task<bool> DeletePostAsync(int id)
    {
        var post = await _postRepository.GetPostByIdAsync(id);
        if (post == null) return false;

        await _postRepository.DeletePostAsync(post);
        return true;
    }

    public async Task<bool> LikePostAsync(int id)
    {
        var post = await _postRepository.GetPostByIdAsync(id);
        if (post == null) return false;

        post.Likes += 1;
        await _postRepository.UpdatePostAsync(post);
        return true;
    }
}
```

PostRepository.cs

```
public class PostRepository
{
    private readonly AppDbContext _context;

    public PostRepository(AppDbContext context)
    {
        _context = context;
    }

    public async Task<List<Post>> GetAllPostsAsync(Guid userId)
    {
        return await _context.Posts
            .Where(p => p.AuthorId == userId)
            .OrderByDescending(p => p.CreatedAt)
            .ToListAsync();
    }

    public async Task<Post?> GetPostByIdAsync(int postId)
    {
        return await _context.Posts.FirstOrDefaultAsync(p => p.Id ==
postId);
    }
```

```
    public async Task AddPostAsync(Post post)
    {
        await _context.Posts.AddAsync(post);
        await _context.SaveChangesAsync();
    }

    public async Task UpdatePostAsync(Post post)
    {
        _context.Posts.Update(post);
        await _context.SaveChangesAsync();
    }

    public async Task DeletePostAsync(Post post)
    {
        _context.Posts.Remove(post);
        await _context.SaveChangesAsync();
    }
}
```

Program.cs

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<AppDbContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultCon
nection")));
builder.Services.AddScoped<PostRepository>();
builder.Services.AddScoped<GoalRepository>();
builder.Services.AddScoped<UserSubscriptionRepository>();
builder.Services.AddScoped<FileRepository>();



builder.Services.AddScoped<PostService>();
builder.Services.AddScoped<GoalService>();
builder.Services.AddScoped<RecommendationService>();
builder.Services.AddScoped<StorageService>();

builder.Services.AddAutoMapper(typeof(MappingProfile));
builder.Services.AddControllers();
builder.Services.AddHttpClient();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "ContentService API",
Version = "v1" });
});

var app = builder.Build();
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
"ContentService API v1"));
```

```
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Додаток 1.3 Файли контейнеризації сервісів на прикладі content-service.

.Dockerfile (content-service)

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER $APP_UID
WORKDIR /app
EXPOSE 8080
EXPOSE 8081

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["src/services/content-service/content-service.csproj",
"src/services/content-service/"]
RUN dotnet restore "src/services/content-service/content-service.csproj"
COPY . .
WORKDIR "/src/src/services/content-service"
RUN dotnet build "content-service.csproj" -c $BUILD_CONFIGURATION -o
/app/build

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "content-service.csproj" -c $BUILD_CONFIGURATION -o
/app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "content-service.dll"]
```

Docker-Compose (Загальна конфігурація)

```
services:
  api-gateway:
    image: api-gateway
    build:
      context: .
      dockerfile: src/api-gateway/Dockerfile
  identity-service:
    image: identity-service
    build:
```

```
      context: .
      dockerfile: src/services/identity-service/Dockerfile
  content-service:
    image: content-service
    build:
      context: .
      dockerfile: src/services/content-service/Dockerfile
  payment-service:
    image: payment-service
    build:
      context: .
      dockerfile: src/services/payment-service/Dockerfile
  mail-service:
    image: mail-service
    build:
      context: .
      dockerfile: src/services/mail-service/Dockerfile
  social-service:
    image: social-service
    build:
      context: .
      dockerfile: src/services/social-service/Dockerfile
  chat-service:
    image: chat-service
    build:
      context: .
      dockerfile: src/services/chat-service/Dockerfile
```

2. Додатки клієнтської частини проєкту.

Додаток 2.1 Файли запитів (на прикладі запитів до сервісу content)

Api/content.js

```
import axios from 'axios';
const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:5000';

export const getGoals = async () => {
    const response = await axios.get(`${API_URL}/api/goals`);
    return response.data;
};
export const createGoal = async (goalData) => {
    const response = await axios.post(`${API_URL}/api/goals`, goalData);
    return response.data;
};
export const updateGoal = async (id, goalData) => {
    const response = await axios.put(`${API_URL}/api/goals/${id}`,
goalData);
    return response.data;
};
export const deleteGoal = async (id) => {
    const response = await axios.delete(`${API_URL}/api/goals/${id}`);
    return response.data;
};
```

```
export const getPostById = async (id, userId) => {
    const response = await axios.get(`${API_URL}/api/posts/${id}`, {
        params: { userId },
    });
    return response.data;
};
export const updatePost = async (id, postData) => {
    const response = await axios.put(`${API_URL}/api/posts/${id}`,
postData);
    return response.data;
};
export const deletePost = async (id) => {
    const response = await axios.delete(`${API_URL}/api/posts/${id}`);
    return response.data;
};
export const getPostsByEmail = async (email) => {
    const response = await axios.get(`${API_URL}/api/posts`, {
        params: { email },
    });
    return response.data;
};
export const createPost = async (email, postData) => {
    const response = await axios.post(`${API_URL}/api/posts`, postData, {
        params: { email },
    });
    return response.data;
};
export const likePost = async (id) => {
    const response = await axios.put(`${API_URL}/api/posts/${id}/like`);
    return response.data;
};
```

Додаток 2.2 Компоненти

Компонент для головної сторінки користувача User.jsx

```
const UserPage = ({user, role}) => {
    return (
        <div className="user-page">
            {role === 'user' ?
                <div className="author-page-content author">
                    fff
                </div>
                :
                <div className="author-page-content user">
                    <Identic user={user}/>
                    <Describer userdata={userdata} mydata={mydata}/>
                    <Actions user={user}/>
                    <UserAbout about={userdata.long_about}/>
                    <SubsPlans/>
                    <Activity/>
```

```
                </div>
            }
        </div>
    );
};
export default UserPage;
```

Вкладений компонент DonationGoal.jsx - приклад компоненту з логікою

```
const GoalPopup = ({toggle, goal = {}}) => {
    const progressPercentage = Math.min((goal.currentAmount /
goal.goalAmount) * 100, 100);
    const [value, setValue] = useState(0);
    const [prediction, setPrediction] = useState(goal.goalBalance);
    const [error, setError] = useState('');
    const [agreed, setAgreed] = useState(false);

    const handleInput = (e) => {
        const newValue = parseFloat(e.target.value) || 0;
        if (newValue < 0) {
            setError("Donation amount cannot be negative.");
            setValue(0);
            setPrediction(goal.goalBalance);
            return;
        }
        setValue(newValue);
        setPrediction(goal.goalBalance - newValue);
        if (goal.goalBalance - newValue < 0) {
            setError("Insufficient balance for this donation.");
        } else {
            setError("");
        }
    };

    const handleDonate = () => {
        if (!agreed) {
            setError("You must agree to the terms to donate.");
            return;
        }
        if (value <= 0 || goal.goalBalance < value) {
            setError("Invalid donation amount.");
            return;
        }
        setError("");


    };

    return (
```

```
        <div className="goal-popup" onClick={toggle}>
            <div className="g-popup-body" onClick={(event) =>
event.stopPropagation()}>
            <div className="g-popup-header">
             <div className="popup-title">{goal.goalTitle}</div>
             <div className="close-button" onClick={toggle}>×</div>
             </div>
                <div className="goal-content">
                <div className="goal-credential">
          <Link to={`/user/${goal.user}`} className="author-credential">
                <img src={goal.userImage} alt="user" className="user-img"/>
                <div className="user-name">{goal.user}</div>
                 </Link>
                 <div className="goal-date">{goal.date}</div>
                 </div>
                    <div className="goal-description-block">
                        {goal.goalImage ? (
                            <div className="description-with-image">
                                <a href={goal.goalImage} target="_blank"
                                    rel="noopener noreferrer">
                                    <img src={goal.goalImage} alt="Goal"
                                      className="goal-image" />
                                </a>
<div className="goal-description-text"{goal.goalDescription}</div>
                            </div>
                        ) : (
<div className="goal-description-text-noimg">{goal.goalDescription}</div>
                        )}
                    </div>
                </div>
 <div className="goal-controls">
    <div className="progress-block">
         <div className="current-value">
           {formatNumber(goal.currentAmount)}
         </div>
         <div className="progressbar-block">
            <div className="percentage" style={{marginLeft:
`${progressPercentage-5}%`}}>{progressPercentage} %</div>
            <div className="progressbar">
              <div className="progress" style={{width:
`${progressPercentage}%`}}></div>
              </div>
                        </div>
      <div className="target-value">{formatNumber(goal.goalAmount)}</div>
                    </div>
                    <div className="g-donation-block">
                        <div className="user-balance">
                            <div className="actual">
            Balance: <span className="balance">{goal.goalBalance}</span>
                            </div>
                            <div className="predict">
Remains: <span className="balance">{prediction >= 0 ? prediction :
0}</span>
```

```
                            </div>
                        </div>
                        <div className="value-handler">
                <div className="error-message">{error ? error : ''}</div>
                            <input
                                type="number"
                                className="donate-amount"
                                placeholder="Enter amount"
                                value={value}
                                onChange={handleInput}
                            />
                        </div>
                        <div className="submittion">
                            <div className="submitted">
                                <label htmlFor="agree" className="label">
                                    I agree to the terms
                                </label>
                                <input
                                    type="checkbox"
                                    id="agree"
                                    className="submit"
                                    checked={agreed}
                                    onChange={() => setAgreed(!agreed)}
                                />
                            </div>
             <div className="donate-button-body" onClick={handleDonate}>
Donate <img className="btn-currency" src={currencyIcon} alt="currency
icon" />
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </div>
    );
};

export default GoalPopup;
```

Компонент з використанням навігації App.jsx

```
function App() {
  return (
    <div className="App">
        {isLoggedIn && <Header user = {user} userRole = {userRole}/>}
        {isLoggedIn && <Navigator/>}
        { isLoggedIn &&
            <div className="app-content">
                <Routes>
                <Route path="/" element={<Homepage />} />
                <Route path="/search" element={<Search/>} />
         <Route path="/create" element={<CreatePost role={userRole}/>}/>
               <Route path="/notifications" element={<Notification/>} />
```

```
                <Route path="/chats" element={<Chats/>} />
     <Route path={`/user/${user.username}`} element={<UserPage user={user}
role = {userRole}/>} />
                <Route path="*" element={<Navigate to="/" />} />
                </Routes>
            </div>
        }
        {!isLoggedIn &&
            <Routes>
                <Route path="/login" element={<Login/>} />
                <Route path="/register" element={<Registration/>} />
                <Route path="/recover/*" element={<Recover />} />
                <Route path="*" element={<Navigate to="/login" />} />
            </Routes>
        }
    </div>
  );
}
export default App;
```

Додаток 2.3 REDUX - сховища даних

Файл сховища та реєстрації Редюсера - store.js

```
const store = configureStore({
    reducer: {
        user: userReducer,
  content: contentReducer,
  payment: paymentReducer,
  social: socialReducer,
    },
});
export default store;
```

Файл обробки даних - UserReducer.js

```
export const authenticateUser = createAsyncThunk(
    'user/authenticateUser',
    async (credentials, thunkAPI) => {
        try {
            const response = await loginUser(credentials);
            if (response.token) {
                localStorage.setItem('token', response.token);
                return response.user;
            }
        } catch (error) {
            return thunkAPI.rejectWithValue(error.response.data);
        }
    }
);
```

```
export const fetchUserProfile = createAsyncThunk(
    'user/fetchUserProfile',
    async (email, thunkAPI) => {
        try {
            const response = await getUserProfile(email);
            return response;
        } catch (error) {
            return thunkAPI.rejectWithValue(error.response.data);
        } } );

const userSlice = createSlice({
    name: 'user',
    initialState: {
        userData: null,
        isLoggedIn: false,
        loading: false,
        error: null,
    },
    reducers: {
        logout: (state) => {
            state.isLoggedIn = false;
            state.userData = null;
            localStorage.removeItem('token');
        }, },
    extraReducers: (builder) => {
        builder
            .addCase(authenticateUser.pending, (state) => {
                state.loading = true;
                state.error = null;
            })
            .addCase(authenticateUser.fulfilled, (state, action) => {
                state.loading = false;
                state.isLoggedIn = true;
                state.userData = action.payload;
            })
            .addCase(authenticateUser.rejected, (state, action) => {
                state.loading = false;
                state.error = action.payload;
            })
            .addCase(fetchUserProfile.pending, (state) => {
                state.loading = true;
            })
            .addCase(fetchUserProfile.fulfilled, (state, action) => {
                state.loading = false;
                state.userData = action.payload;
            })
            .addCase(fetchUserProfile.rejected, (state, action) => {
                state.loading = false;
                state.error = action.payload;
            }); }, });
export const { logout } = userSlice.actions;
export default userSlice.reducer;
```