МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ ОСОБИСТИМИ ФІНАНСАМИ ЗА ДОПОМОГОЮ QUARKUS ТА REACT»**

Виконав: студент 4 курсу, групи 6.1210-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Н.В. Волков
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Лимаренко Ю.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент професор кафедри комп'ютерних наук, професор, д.т.н. Шило Г.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Волкову Нікіті Володимировичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи управління особистими фінансами за допомогою Quarkus та React

керівник роботи Лимаренко Юлія Олексіївна, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 21 » грудня 2023 року № 2180-с

2. Строк подання студентом роботи 03.06.2024 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка системи управління фінансами.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою докладу

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 25.12.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	09.01.2024	
2.	Збір вихідних даних.	12.01.2024	
3.	Обробка методичних та теоретичних	25.01.2024	
	джерел.		
4.	Розробка першого та другого розділу.	12.04.2024	
5.	Розробка третього розділу.	20.05.2024	
6.	Оформлення та нормоконтроль	27.05.2024	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	16.06.2024	

Студент _____ Н.В. Волков
(підпис) (ініціали та прізвище)

Керівник роботи _____ Ю.О. Лимаренко
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка системи управління особистими фінансами за допомогою Quarkus та React»: 53 с., 23 рис., 10 джерел.

МІКРОСЕРВІСИ, ФІНАНСИ, CLOUD NATIVE, JAVA, MARIADB, REACT.

Об’єкт дослідження – розробка додатка для контроля особистих фінансів використовуючи мікросервісну архітектуру.

Мета роботи: розробити функціональне та практичне програмне забезпечення для контролю фінансів з використанням мікросервісної архітектури.

Метод дослідження – методи програмної інженерії.

Результати та їх новизна: був розроблений додаток який дозволяє користувачу контролювати свої транзакції розділені на категорії, також можливо керувати своїми боргами.

Взаємозв’язок з іншими роботами: дана робота базується на дослідженні Quarkus та мікросервісної архітектури на Java, способів взаємодії між сервісами та між клієнтом та сервером.

Рекомендації щодо використання результатів роботи: розроблений додаток для контроля особистих фінансів може бути використаний як для задоволення одного користувача так і багатьох завдякі масштабованості архітектури.

Значимість роботи та висновки: розробки додатка для контроля особистих фінансів є дуже актуальною оскільки дуже важливо розуміти принципи планування власних коштовних ресурсів та доцільно їх використовувати. В разі вкладення достатніх ресурсів, додаток може замінити особистого фінансового консультанта, при цьому коштувати набагато дешевше.

SUMMARY

Bachelor’s qualifying paper “Development of a Personal Finance Management System Using Quarkus and React”: 53 pages, 23 figures, 10 references.

MICROSERVICES, FINANCE, CLOUD NATIVE, JAVA, MARIADB, REACT.

The object of the study is the development of an application for personal finance control using microservice architecture.

The aim of the study: to develop functional and practical software for financial control using microservice architecture.

The method of research is software engineering methods.

Results and novelty: an application has been developed that allows the user to control their transactions divided into categories, it is also possible to manage their debts.

Relationship to other works: this work is based on the study of Quarkus and microservice architecture in Java, the ways of interaction between services and between client and server.

Recommendations for using the results of the work: the developed application for personal finance control can be used to satisfy both one user and many due to the scalability of the architecture.

Significance of the work and conclusions: the development of an application for personal finance control is very relevant because it is very important to understand the principles of planning your own valuable resources and use them appropriately. If sufficient resources are invested, the application can replace a personal financial advisor, while costing much less.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Аналіз предметної області..... 9
1.1 Характеристика предметної області 9
1.2 Огляд існуючих систем для контроля особистих фінансів 10
1.2.1 Додаток YNAB (You Need A Budget) 11
1.2.2 Додаток Personal Capital..... 12
1.3 Стек технологій..... 12
1.3.1 Бізнес логіка 13
1.3.2 База даних..... 15
1.3.3 Користувацький інтерфейс 18
1.4 Огляд концепцій розгортання мікросервісних додатків..... 20
1.4.1 Огляд cloud native 20
1.4.2 Огляд cloud enabled..... 21
1.4.3 Огляд cloud optimized..... 22
1.5 Використання Quarkus в контексті cloud native..... 23
1.6 Висновки до розділу 1 24
2 Проєктування вебдодатку 25
2.1 Формування технічного завдання 25
2.1.1 Технічні вимоги 25
2.1.2 Етап концептуального проєктування..... 26
2.2 Етап фізичного проєктування..... 30
2.3 Схема бази даних 32
2.4 Висновки до розділу 2 34
3 Особливості реалізації та результати тестування..... 35

3.1 Розробка серверної частини додатка 35
3.1.1 Створення та ініціалізація проєкту 35
3.1.2 Схема бази даних 36
3.1.3 Особливості реалізації логіки вебзастосунка 38
3.2 Особливості реалізації інтерфейсу вебзастосунку 42
3.3 Перевірка кожного сервіса та способів їхньої інтеграції..... 46
3.4 Висновки до розділу 3 50
Висновки 51
Перелік посилань..... 53

ВСТУП

У сучасному світі, де фінансова грамотність стає все більш важливою складовою успіху та стабільності, існує потреба у зручних та ефективних інструментах для управління особистими фінансами. Одним із найбільш потужних та перспективних інструментів в розробці програмного забезпечення на сьогоднішній день є Quarkus та React.Js.

Мета даної роботи полягає у розробці системи управління особистими фінансами з використанням цих технологій. Користувачам потрібні рішення які можуть автоматизувати звичайні рутинні справи та використовуватись без встановлення на особистий комп’ютер. Рішення на основі Quarkus та React.Js, може надати цю можливість.

У даній роботі буде детально розглянуто процес проєктування та розробки системи, від архітектурних рішень до реалізації функціоналу та тестування. Також ми проаналізуємо можливості та переваги використання Quarkus та React.Js у контексті проєкту, а також буде проведено порівняльний аналіз з іншими технологіями.

Основні принципи, на яких ґрунтується така система, – це доступність, зручність та безпека. Шляхом поєднання Java з її надійністю та швидкістю і React.Js з його інтуїтивним синтаксисом та можливостями реалізації сучасних вебдодатків, ми прагнемо створити інструмент, який буде ефективним для широкого кола користувачів.

Очікуваним результатом цієї роботи є створення комплексної та інтуїтивно зрозумілої системи, яка допоможе користувачам з легкістю відслідковувати, аналізувати та планувати свої фінанси. Ця робота спрямована на створення інструменту, що підвищить фінансову свідомість та забезпечить кожного користувача необхідними знаннями та інструментами для досягнення фінансового благополуччя.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Характеристика предметної області

Система управління особистими фінансами (Personal Finance Management System, PFMS) – це комплекс інструментів та методів, які допомагають індивідуумам ефективно керувати своїми фінансами. Вона включає в себе планування бюджету, відстеження витрат, управління боргами, інвестування та довгострокове фінансове планування. У сучасному цифровому світі ці системи часто реалізуються у вигляді програмного забезпечення або мобільних додатків.

Однією з ключових функцій PFMS є планування бюджету. Це включає встановлення фінансових цілей, таких як накопичення коштів на купівлю житла, автомобіля або фінансування освіти. Користувачі можуть створювати персональні бюджети, які відображають їхні доходи та витрати, що дозволяє їм планувати, як найкраще використовувати свої фінансові ресурси.

Відстеження витрат є ще однією важливою функцією. Системи управління особистими фінансами дозволяють автоматично або вручну класифікувати витрати за різними категоріями (транспорт, розваги тощо). Це допомагає користувачам отримати чітке уявлення про те, куди саме витрачаються їхні гроші, виявити потенційні області для економії.

Управління доходами також є важливою складовою PFMS. Системи дозволяють користувачам вносити дані про всі джерела доходів, включаючи зарплати, доходи від інвестицій та інші джерела. Це дає можливість аналізувати загальний рівень доходів і їх стабільність, що є важливим для довгострокового фінансового планування.

Сучасні системи управління особистими фінансами часто включають додаткові можливості, такі як інтеграція з банківськими рахунками та кредитними картками, що дозволяє автоматично імпортувати транзакції і зменшує потребу в ручному введенні даних. Це підвищує точність та зручність

використання системи.

Також важливими є інструменти для інвестування та управління активами. Користувачі можуть відстежувати свої інвестиційні портфелі, аналізувати їх продуктивність та отримувати рекомендації щодо оптимізації інвестиційних стратегій.

Для багатьох користувачів системи управління особистими фінансами стають незамінними інструментами для досягнення фінансової стабільності та впевненості в майбутньому. Вони допомагають краще розуміти свої фінансові звички, уникати зайвих витрат і ефективніше накопичувати кошти. Крім того, вони сприяють розвитку фінансової грамотності, що є ключовим фактором для досягнення довгострокових фінансових цілей.

В системі управління особистими фінансами користувачі мають можливість вводити дані про свої доходи з різних джерел, таких як заробітна плата, дивіденди, рента чи інші форми доходу. Ця інформація дозволяє аналізувати фінансовий стан, робити прогнози та приймати обґрунтовані рішення щодо витрат та інвестицій. Система може пропонувати інтерактивні графіки та звіти, які допомагають користувачам бачити, як змінюються їхні доходи з часом, і виявляти можливі тенденції або проблеми.

Системи управління особистими фінансами є незамінними інструментами для сучасних споживачів, які прагнуть ефективно керувати своїми грошима. Вони забезпечують комплексний підхід до фінансового планування, допомагають уникнути непередбачених витрат та забезпечують досягнення фінансових цілей. Завдяки постійному розвитку технологій ці системи стають все більш функціональними та доступними, що дозволяє користувачам отримувати ще більше користі від їх використання.

1.2 Огляд існуючих систем для контроля особистих фінансів

В даному розділі ми розглянемо існуючі системи для контроля особистих фінансів. Їх особливості, вартість та методологію яку вони

пропонують впровадити користувачу. Також буде розглянуто наявність обов'язкових атрибутів, таких як наявність контроля заборгованостей, автоматичне або ручне категоризування витрат.

1.2.1 Додаток YNAB (You Need A Budget)

YNAB (You Need A Budget) – це платний додаток для управління особистими фінансами, який акцентує увагу на методології активного бюджетування. Основою цього додатка є чотири правила бюджетування: кожному долару потрібно призначити завдання, пристосовувати бюджет до реальних змін, бути готовим до несподіванок і припинити життя від зарплати до зарплати. Ці правила допомагають користувачам більш ефективно керувати своїми фінансами та досягати фінансових цілей.

Додаток дозволяє автоматично імпортувати транзакції з банківських рахунків та кредитних карток, забезпечуючи актуальність даних. Для зручності користувачів є мобільний додаток для iOS та Android, що дозволяє керувати фінансами з будь-якого місця.

Окрім того, YNAB надає доступ до освітніх ресурсів, таких як вебінари та підручники, які допомагають покращити фінансову грамотність. Візуалізація доходів і витрат, а також аналітичні звіти допомагають користувачам краще розуміти свої фінансові звички та ефективність бюджетування.

YNAB пропонує безкоштовну 34-денну пробну версію, після чого користувачі можуть обрати щорічну підписку на рік або на місяць. Основні переваги YNAB включають ефективну методологію бюджетування, інтеграцію з банківськими рахунками та освітні матеріали. Однак, додаток може здатися дорогим для деяких користувачів, а його методологія потребує часу для освоєння.

1.2.2 Додаток Personal Capital

Personal Capital – це платний додаток для управління інвестиціями та особистими фінансами, який об’єднує функції бюджетування з інструментами для аналізу та управління інвестиційним портфелем. Додаток дозволяє підключати банківські рахунки, кредитні картки та інвестиційні рахунки для автоматичного збору даних, що забезпечує повну картину фінансового стану користувача.

Один із ключових аспектів Personal Capital – управління інвестиціями. Додаток надає можливість аналізувати інвестиційний портфель, оцінювати ризики та продуктивність активів, пропонує плани з диверсифікації та рекомендації з інвестування. Personal Capital також має інструменти для пенсійного планування, які дозволяють прогнозувати пенсійні заощадження та оцінювати фінансове здоров’я користувача.

Автоматичне категоризування витрат, створення бюджету та відстеження виконання є невід’ємними частинами додатка. Personal Capital забезпечує високий рівень безпеки даних за допомогою багаторівневого шифрування. Для користувачів преміум-рівня доступні консультації з фінансовими радниками, що дозволяє отримувати персоналізовані фінансові поради.

Основні переваги Personal Capital включають потужні інструменти для управління інвестиціями та фінансового планування, інтеграцію з різними фінансовими рахунками та високий рівень безпеки. Однак, додаток може бути складним для новачків, а деякі функції доступні тільки за додаткову плату.

1.3 Стек технологій

В цьому розділі буде розглянуто стек технологій які будуть використані. Для розробки клієнтської частини додатку буде використано фреймворк React.

Для валідації форм та обробки даних користувачів буде застосовано бібліотеку React Hook Form. На серверній частині буде використано Quarkus – фреймворк для Java, який оптимізований для роботи в хмарних середовищах та контейнерах.

1.3.1 Бізнес логіка

Розглядаючи Java у контексті вибору технологій для розробки бізнес логіки додатка для контроля особистих фінансів можна відзначити кілька важливих аспектів.

Java має широкий вибір бібліотек і фреймворків для Java, які допомагають розробникам швидше та ефективніше створювати програми. Більшість з цих бібліотек є відкритими, що сприяє зручності їх використання та налагодженню.

Також важливо відзначити безпеку Java, яка вбудована у мову завдяки різним механізмам, таким як віртуальна машина Java (JVM). Це допомагає запобігати багатьом типам атак та забезпечує відокремлення програм від операційної системи.

Quarkus – це інноваційний фреймворк для розробки Java-додатків, який спеціально створений для хмарних і мікросервісних архітектур. Одна з основних переваг Quarkus полягає в його високій продуктивності та ефективності використання ресурсів. Він оптимізований для швидкості розгортання, споживає мінімум пам’яті та працює ефективно в умовах облачних середовищ.

Поєднання Quarkus і Kubernetes забезпечує ідеальне середовище для створення масштабованих, швидких і легких додатків. Quarkus значно підвищує продуктивність розробників завдяки інструментам, вбудованим інтеграціям, прикладним сервісам тощо.

Quarkus має вбудовані можливості для використання RBAC (Role based

access control). Зважаючи на те що у додатку будуть зберігатись персональні данні, та данні про фінансовий стан користувача, виток такої інформації може привести до жахливих наслідків. Враховуючи чутливість інформації, ретельна настройка RBAC є критично важливою.

У процесі розробки мікросервісного додатку на Quarkus для управління фінансами, важливо створити зв’язок між сервісами, наприклад, за допомогою REST, gRPC або брокерів повідомлень. Фреймворк має підтримку багатьох брокерів повідомлень таких як RabbitMQ, Apache Kafka, Hazelcast. Для між сервісної комунікації був обраний брокер повідомлень NATS та gRPC для комунікації між шлюзом та сервісами. Саме такий підхід зможе задовільнити такі архітектурні вимоги як масштабованість, низька зв’язність сервісів та незалежне масштабування окремих компонентів [8].

Однією з ключових складових успішного мікросервісного додатку є його моніторинг та логування. Для цього Quarkus надає інструменти, такі як Micrometer та інтеграції з такими платформами як: ELK Stack та OTEL для централізованого збору та аналізу логів. Це дозволяє забезпечити високу доступність та відслідковуваність додатку, що є критичними аспектами в сучасних розподілених системах.

Використання Micrometer дозволяє отримати детальну інформацію про продуктивність додатку, включаючи метрики, що відображають стан системи, таких як затримки відповіді, кількість оброблених запитів та використання ресурсів. Інтеграція з ELK Stack (Elasticsearch, Logstash, Kibana) забезпечує потужні можливості для збору, зберігання та візуалізації логів, що полегшує виявлення та усунення проблем у додатку. Застосування OTEL (OpenTelemetry) надає єдиний стандарт для збору телеметрії, що спрощує інтеграцію з іншими інструментами та платформами моніторингу.

Завдяки цим інструментам, команди розробників можуть оперативно реагувати на інциденти, аналізувати причини проблем та проводити оптимізацію додатку. Це також сприяє покращенню якості обслуговування користувачів, оскільки дозволяє мінімізувати час простою та забезпечити стабільну роботу системи.

1.3.2 База даних

MariaDB – це розроблена спільнотою, комерційно підтримуваний форк реляційної системи управління базами даних MySQL, яка має намір залишатися вільним програмним забезпеченням з відкритим вихідним кодом під ліцензією GNU General Public License.

MariaDB має кілька переваг, що роблять її популярним вибором для багатьох розробників та підприємств. По-перше, це проєкт з відкритим кодом, що означає безкоштовне використання та активну спільноту розробників, які постійно вдосконалюють її. По-друге, MariaDB є форком MySQL і підтримує високу сумісність з нею, що дозволяє легко мігрувати з MySQL на MariaDB без значних змін у коді або конфігураціях.

MariaDB також забезпечує високу продуктивність та надійність завдяки покращенням у внутрішній архітектурі, таким як оптимізовані індекси та розширені можливості реплікації. Крім того, вона пропонує додаткові функції, яких немає в MySQL, наприклад, підтримку різних механізмів зберігання даних, включаючи Aria, ColumnStore та MyRocks, що дозволяє налаштовувати базу даних під конкретні потреби проєкту.

Для роботи з базою даних була обрана така ORM як Hibernate через велику кількість переваг по-перше, Hibernate забезпечує прозору обробку збереження та завантаження об’єктів, що значно спрощує роботу з базою даних. Завдяки автоматичному керуванню транзакціями та зв’язками між об’єктами, розробники можуть зосередитися на бізнес-логіці, не турбуючись про рутинні операції з базою даних [3].

По-друге, Hibernate підтримує широкий спектр баз даних і легко інтегрується з іншими технологіями Java, що забезпечує гнучкість і масштабованість застосунків. Важливою перевагою є також використання HQL (Hibernate Query Language), який дозволяє писати запити до бази даних на об’єктному рівні, що робить код більш читабельним і підтримуваним.

Крім того, Hibernate пропонує розширені можливості управління

транзакціями, що дозволяє легко реалізовувати складні сценарії обробки даних. Завдяки підтримці різних рівнів ізоляції транзакцій і оптимістичного блокування, Hibernate забезпечує високу цілісність даних та мінімізує конфлікти під час одночасного доступу до бази даних.

Інструменти автоматичного створення і оновлення схеми бази даних, такі як Hibernate Schema Export, спрощують управління базою даних, дозволяючи розробникам легко синхронізувати об’єктну модель з фізичною структурою бази даних.

Ще однією важливою перевагою Hibernate є його розширюваність та налаштовуваність. Розробники можуть використовувати різноманітні анотації та XML-конфігурації для точного налаштування поведінки ORM. Це дозволяє адаптувати Hibernate до конкретних потреб проєкту та оптимізувати продуктивність застосунків.

Завдяки активній спільноті та широкій базі знань, Hibernate має відмінну документацію і безліч ресурсів для навчання та підтримки. Це допомагає розробникам швидко вирішувати проблеми та ефективно використовувати цей інструмент у своїх проєктах. Інтеграція з популярними фреймворками, такими як Spring та Quarkus, робить Hibernate ще більш привабливим для розробників, оскільки спрощує створення масштабованих та ефективних застосунків.

Узагальнюючи, Hibernate ORM пропонує потужний набір інструментів та функцій для роботи з базами даних, забезпечуючи високу продуктивність, гнучкість та легкість у використанні, що робить його одним з найпопулярніших виборів серед розробників Java-застосунків.

Для міграцій був обраний такий інструмент як Liquibase. По-перше, він дозволяє керувати змінами в базі даних через файли, що легко версіонуються. Це допомагає підтримувати актуальність бази даних серед різних середовищ розробки та виробництва [5].

По-друге, Liquibase підтримує різноманітні бази даних, що робить його універсальним інструментом для різних проєктів. По-третє, зручність автоматизації змін в базі даних дозволяє інтегрувати його в CI/CD процеси,

забезпечуючи безперервну інтеграцію та розгортання.

Крім того, Liquibase надає можливості для зворотного відкату змін, що забезпечує безпеку та гнучкість у випадку помилок. Його XML, YAML або JSON файли конфігурації легко читаються та зрозумілі, що спрощує процес написання та підтримки скриптів змін. У результаті, використання Liquibase допомагає забезпечити контрольоване та структуроване управління змінами в базі даних, що покращує якість та стабільність розробки програмного забезпечення.

Liquibase також надає можливість відстежувати історію змін бази даних, що є важливим для розуміння того, які саме зміни були внесені і коли. Це може бути корисним під час налагодження проблем або аудиту. Інструмент забезпечує підтримку різних стратегій міграції бази даних, таких як інкрементальні зміни або повні перезаписи, що дозволяє вибирати найефективніший підхід для конкретного проєкту.

Однією з ключових переваг Liquibase є його здатність інтегруватися з іншими інструментами розробки та оркестрації, такими як Maven, Gradle, Jenkins, та інші. Це робить його частиною єдиної екосистеми DevOps, забезпечуючи безшовний робочий процес для розробників і адміністраторів баз даних.

Liquibase також має активну спільноту та регулярні оновлення, що забезпечує доступ до нових функцій і виправлень помилок. Це гарантує, що інструмент залишається актуальним і корисним для сучасних вимог до розробки програмного забезпечення. Крім того, існує детальна документація та численні приклади використання, що спрощує початок роботи з інструментом та його інтеграцію в проєкт.

Використання Liquibase сприяє стандартизації процесів управління змінами бази даних, що допомагає уникнути людських помилок та покращує загальну ефективність команди розробників. Це особливо важливо для великих команд, де необхідна чітка координація та контроль за всіма змінами, що вносяться в систему.

Загалом, Liquibase є потужним і гнучким інструментом, який значно полегшує управління змінами в базі даних, забезпечуючи при цьому високу якість, стабільність і безпеку розробки програмного забезпечення.

1.3.3 Користувацький інтерфейс

React – це javascript бібліотека для розробки користувацьких інтерфейсів. Інтерфейс користувача будується з невеликих блоків, таких як кнопки, текст та зображення. React дозволяє об’єднувати їх у багаторазові вкладені компоненти. Від вебсайтів до додатків для телефонів – все, що ви бачите на екрані, можна розбити на компоненти [1].

React відрізняється великою кількістю переваг, серед яких значимість інтерфейсу компонентами, здатність до простоти використання без великої кількості коду.

Крім того, React забезпечує високу продуктивність завдяки своїй архітектурі та підходу до оновлення інтерфейсу користувача. Використовуючи Virtual DOM, React мінімізує кількість операцій з реальним DOM, що значно прискорює рендеринг і оновлення інтерфейсу. Це особливо важливо для великих додатків, де кількість змін і оновлень може бути значною.

Іншим важливим аспектом є можливість використання JSX – розширення синтаксису JavaScript, яке дозволяє писати HTML-подібний код безпосередньо в JavaScript. Це спрощує розробку інтерфейсу і робить код більш читабельним та підтримуваним. JSX також дозволяє легко впроваджувати логіку всередині компонентів, що робить їх більш гнучкими та потужними.

Зважаючи на вищесказане компоненти у React також можуть бути використані повторно. Це означає, що ми можемо створювати універсальні компоненти, які можна використовувати на різних сторінках нашого додатка

для контролю особистих фінансів. Наприклад, компонент для відображення таблиці може бути використаний як на сторінці транзакцій, так і на сторінці заборгованостей.

React має широку екосистему бібліотек і інструментів, таких як Redux для управління станом додатка, React Router для управління навігацією, та багато інших. Це дозволяє розробникам легко розширювати функціональність своїх додатків і інтегрувати React з іншими технологіями.

На основі Redux є така бібліотека як RTK Query (Redux Toolkit Query) є потужним інструментом і забезпечує ефективне управління серверними запитами та кешуванням даних. RTK Query значно спрощує роботу з асинхронними запитами, такими як отримання, додавання, оновлення і видалення даних з сервера, завдяки своїм вбудованим функціям.

Однією з головних переваг RTK Query є автоматичне кешування результатів запитів. Це дозволяє уникнути зайвих запитів до сервера, покращуючи продуктивність додатка та зменшуючи навантаження на сервер. RTK Query автоматично оновлює кеш при зміні даних, що забезпечує консистентність і актуальність інформації в додатку.

Простота використання RTK Query робить його привабливим для розробників. Використовуючи його, можна легко створювати та керувати асинхронними запитами без необхідності написання великої кількості коду для обробки стану, помилок та інших аспектів. Вбудована обробка помилок та автоматичне повторення запитів у разі помилок мережі забезпечують стабільну роботу додатка навіть за умов нестабільного інтернет-з’єднання.

RTK Query також забезпечує відмінну інтеграцію з існуючою інфраструктурою Redux. Це дозволяє розробникам легко інтегрувати RTK Query у свої додатки, які вже використовують Redux для управління станом. Використання RTK Query разом з Redux Toolkit спрощує налаштування та зменшує кількість коду, необхідного для управління станом і запитами.

Ще однією важливою особливістю RTK Query є підтримка серверного рендерингу (SSR). Це дозволяє використовувати його у додатках, які

потребують попереднього рендерингу на сервері для покращення SEO та швидкості завантаження сторінок. RTK Query надає інструменти для простого налаштування SSR, що робить його відмінним вибором для сучасних додатків, що потребують високої продуктивності та оптимізації.

RTK Query також підтримує інвалідизацію кешу, що дозволяє точно контролювати, коли дані в кеші повинні бути оновлені. Це особливо корисно для додатків, які часто працюють з динамічними даними, що змінюються в реальному часі.

Таким чином, React є потужним інструментом для розробки сучасних вебдодатків, що забезпечує високу продуктивність, гнучкість і масштабованість. Його підхід до компонентної архітектури дозволяє створювати модульні і повторно використовувані елементи інтерфейсу, що значно спрощує процес розробки та підтримки додатків.

1.4 Огляд концепцій розгортання мікросервісних додатків

1.4.1 Огляд cloud native

Cloud native розглядається через призму контейнеризації, яка дозволяє упаковувати додатки та їх залежності для забезпечення портативності та швидкого розгортання. Мікросервісна архітектура дозволяє створювати додатки як набір невеликих, незалежних компонентів, що полегшує розробку та підтримку. Автоматизація інфраструктури та використання контейнерних оркестраторів спрощують керування інфраструктурою та забезпечують масштабованість та надійність додатків.

Одним із ключових аспектів cloud native є автоматизація інфраструктури, яка допомагає автоматизувати процеси розгортання, масштабування та керування інфраструктурою. Це робить управління додатками більш ефективним та забезпечує стабільність та безпеку системи.

Автоматизації розгортання можливо досягнути завдяки Kubernetes це інструмент для масштабування та керування контейнерами у cloud native середовищах. Контейнерні оркестратори спрощують управління додатками, роблять їх більш масштабованими та надійними, а також дозволяють розробникам концентруватися на функціональності додатків, а не на інфраструктурних деталях.

Забезпечення безпеки та захисту даних є однією з основних вимог для cloud native додатків. Використання механізмів шифрування, ідентифікації та автентифікації, а також регулярне оновлення та моніторинг інфраструктури є необхідними кроками для забезпечення безпеки у cloud native середовищах.

Хоча cloud native підхід дозволяє розробникам швидше впроваджувати зміни та реагувати на ринкові вимоги, його впровадження також вимагає від організацій зміни культури, процесів розробки та управління, що може бути складним завданням, але при цьому забезпечує більшу гнучкість, швидкість реакції та зниження часу до випуску продукту на ринок.

1.4.2 Огляд cloud enabled

Cloud enabled це cloud native системи, модифіковані під конкретні потреби компанії, але не обов’язково під структуру її бізнесу. У хмарному рішенні додаток організації розгортається в публічній хмарі, але для його роботи все одно потрібен власний фізичний сервер.

Різниця між cloud-native та cloud-enabled системою полягає в тому, що cloud-native система не потребує жодної обчислювальної інфраструктури на місці. Усіма стратегіями доступності займається зовнішній постачальник послуг публічної хмари. Підприємствам, які прагнуть заощадити кошти та розширити функціональність, але мають численні застарілі системи, доведеться обирати cloud-native.

Якщо узагальнити cloud-native працюють без власної обчислювальної

інфраструктури і мають можливості горизонтального масштабування на декількох серверах. А cloud-base рішення, що покладаються на внутрішнє апаратне забезпечення для певних операцій.

1.4.3 Огляд cloud optimized

Cloud optimized це рішення, модифіковане спеціально для хмарних середовищ. Це означає, що хмарно-оптимізовані системи працюють в рамках власних технологій та API постачальників хмарних послуг, що також забезпечує сумісність з іншими платформами хмарного зберігання даних.

Cloud-optimized рішення можуть одночасно бути cloud-diagnostic, cloud-native, cloud-enabled або cloud-first продуктами. Як правило, постачальники хмарних послуг пропонують хмарну оптимізацію як безкоштовну послугу для всіх своїх клієнтів. Cloud-optimized пропонує багато з тих самих переваг хмарних обчислень без внесення змін до існуючої інфраструктури або обладнання.

Такі рішення мають декілька ключових переваг, серед яких підвищена гнучкість, масштабованість та доступність. Вони дозволяють підприємствам швидко реагувати на зміни в робочих навантаженнях і забезпечують високу доступність даних завдяки використанню хмарних технологій. Це також сприяє зниженню витрат на обслуговування та оновлення апаратного забезпечення, оскільки більшість операцій здійснюється на стороні постачальника хмарних послуг.

Крім того, cloud-optimized рішення забезпечують підвищену безпеку даних через вбудовані механізми захисту та відповідність міжнародним стандартам. Вони також можуть легко інтегруватися з іншими хмарними сервісами та додатками, що дозволяє створювати комплексні та ефективні системи управління даними. Завдяки цьому підприємства можуть зосередитись на своїх основних бізнес-процесах, не турбуючись про технічні аспекти управління інфраструктурою.

1.5 Використання Quarkus в контексті cloud native

Зважаючи на вищесказане було вирішено обрати концепцію cloud-native. Дана концепція відмінно впроваджується в фреймворк Quarkus.

Quarkus – це фреймворк для розробки Java-додатків, який активно використовується в контексті cloud native додатків. Одна з головних переваг Quarkus у цьому контексті полягає у високій продуктивності та ефективності роботи в хмарних середовищах.

Ось кілька ключових переваг Quarkus у контексті cloud native.

Швидкість завантаження: Quarkus оптимізований для швидкого старту, що особливо важливо у хмарних середовищах, де потрібно швидко реагувати на зміни обсягу ресурсів.

Мале споживання пам’яті: Quarkus використовує менше пам’яті, завдяки мінімізації використання рефлексії що дозволяє економно використовувати ресурси в хмарних середовищах і оптимізувати витрати.

Завдяки низькому споживанню ресурсів, Quarkus ідеально підходить для створення масштабованих хмарних додатків, які забезпечують високу швидкість реакції на запити користувачів. Він дозволяє розробникам зосередитися на функціональності додатків, не переймаючись про витрати на ресурси або час, потрібний для розгортання.

Quarkus був розроблений з урахуванням підтримки різних парадигм. Це означає, що модель розробки Quarkus видозмінюється, щоб адаптуватися до типу програми, яку ви розробляєте: моноліт, мікросервіс, реактивний, event driven, функції [10].

Окрім того, Quarkus має дуже активну спільноту та широкий спектр розширень, які допомагають прискорити розробку та забезпечити високу якість коду. Це робить його ідеальним інструментом для командної розробки, де кожен учасник може внести свій внесок та підтримувати стабільну роботу додатків у хмарному середовищі.

Підтримка різних джерел даних: Quarkus дозволяє легко підключати

різні джерела даних, включаючи бази даних, сервіси кешування та інші зовнішні сервіси, що є необхідним для хмарних додатків.

Розширені можливості тестування: Quarkus надає зручні засоби для тестування додатків, включаючи інтеграційні та модульні тести, що сприяє покращенню якості коду в cloud native середовищах.

В цілому, Quarkus є потужним інструментом для розробки cloud native додатків, який дозволяє забезпечити високу продуктивність, ефективне використання ресурсів та простоту управління додатками в хмарних середовищах.

1.6 Висновки до розділу 1

У даному розділі ми детально розглянули концепції cloud native та те як вони реалізовані в quarkus. Ці концепції буде використано як основні компоненти для розробки додатку для контролю фінансів. Використання quarkus, завдяки першокласній підтримці різних парадигм, дозволить створювати рішення будь-якого масштабу за дуже короткий час.

2 ПРОЄКТУВАННЯ ВЕБДОДАТКУ

2.1 Формування технічного завдання

Перед початком створення додатку для контроля особистих фінансів потрібно сформувати технічне завдання, діаграми та схеми за якими цей додаток буде реалізований. Після цього необхідно визначити архітектуру системи, включаючи вибір технологій та фреймворків для клієнтської та серверної частин.

2.1.1 Технічні вимоги

На початку роботи були сформовані такі технічні вимоги: Серверна частина ПЗ повинна працювати на системах сімейства Linux, використовувати базу даних: MariaDB або MySQL. Версія Java повинна бути не старша ніж 21. Серверна частина повинна використовувати Quarkus 3. Авторизація за допомогою JWT токенів. Аутентифікація за допомогою логіна та пароля. Клієнтська частина повинна бути SPA типу. UI частина має бути розроблена на Material UI. Має бути валідація на формах інтерфейса з використанням React Hook Form. Взаємодія з бекендом має бути реалізована асинхронно з використанням RTK Query.

Крім того, особлива увага приділяється оптимізації завантаження сторінок для забезпечення швидкого відгуку користувача інтерфейсу. Це досягається за допомогою розумного поділу коду та динамічного імпортування модулів, що дозволяє зменшити час першого завантаження.

Під час розробки мають бути використані сучасні практики CI/CD для автоматизації процесів збірки та розгортання застосунку. Розгортання має бути в контейнерному середовищі такому як Kubernetes.

2.1.2 Етап концептуального проєктування

В цьому розділі наведемо загальну архітектуру вебзастосунка за допомогою діаграм прецедентів, діаграм послідовності, та BPMN нотацій.

На рисунку 2.1 зображена нотація яка демонструє перевірку під час будь яких дій у системі.

Рисунок 2.1 – BPMN нотація перевірки входу користувача

Нотація починається з того, що система перевіряє чи ввійшов користувач у систему, якщо так то дія яку виконує користувач буде дозволена для виконання, але якщо користувач не ввійшов у систему то система буде перенаправляти на сторінку входу.

На рисунку 2.2 зображена нотація створення транзакції. Нотація починається з перевірки входу користувача, і якщо користувач увійшов у систему то йому буде дозволено створити транзакцію, після відправлення форми з даними транзакції, буде виконана перевірка вхідних даних, якщо вхідні дані вірні, то вони будуть збережені.

В разі якщо користувач не ввійшов у систему або дані які він ввів не пройшли валідацію він отримає повідомлення про помилку.

На рисунку 2.3 зображена нотація створення боргу. Нотація починається з перевірки входу користувача, і якщо користувач увійшов у систему то йому

буде дозволено створити борг, після відправлення форми з даними про борги користувача, буде виконана перевірка вхідних даних, якщо вхідні дані вірні, то вони будуть збережені.

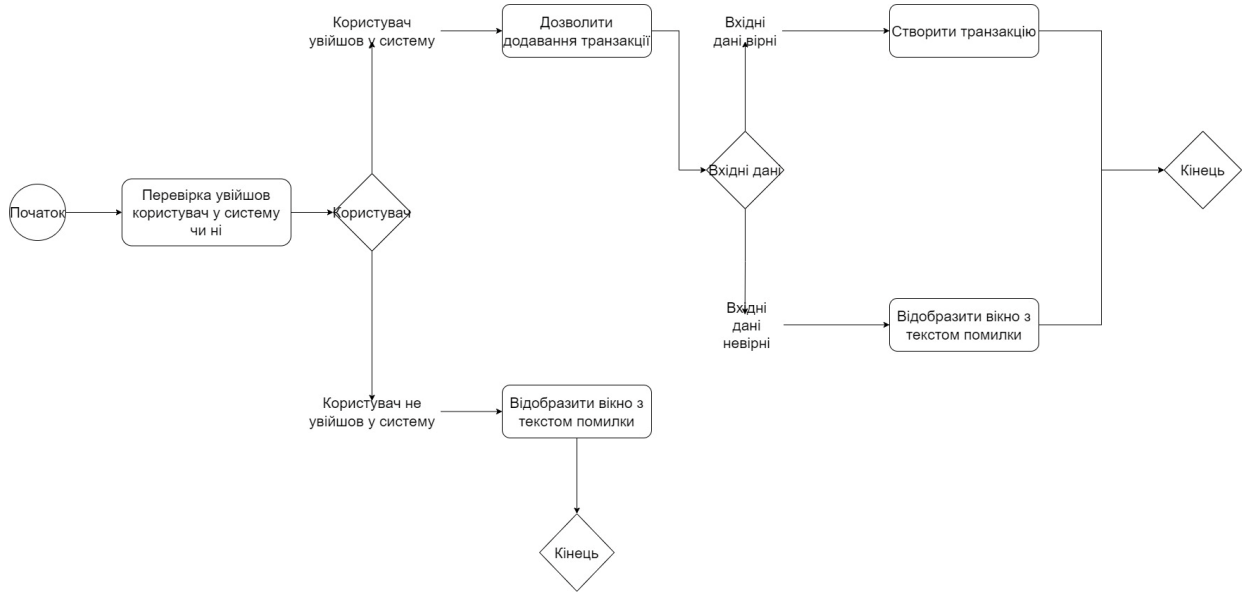

Рисунок 2.2 – BPMN нотація створення транзакції

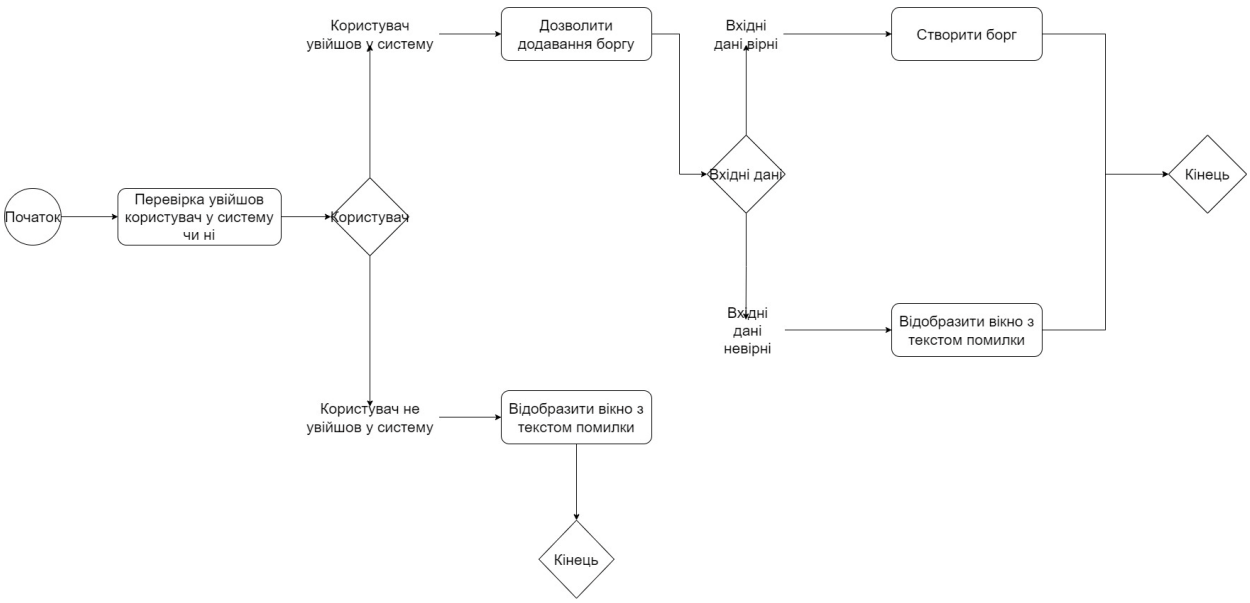

Рисунок 2.3 – BPMN нотація створення боргу

На рисунку 2.4 зображена діаграма прецедентів, її діячом є користувач. Він може виконувати такі дії як: створення транзакцій, створення боргів, редагування та видалення.

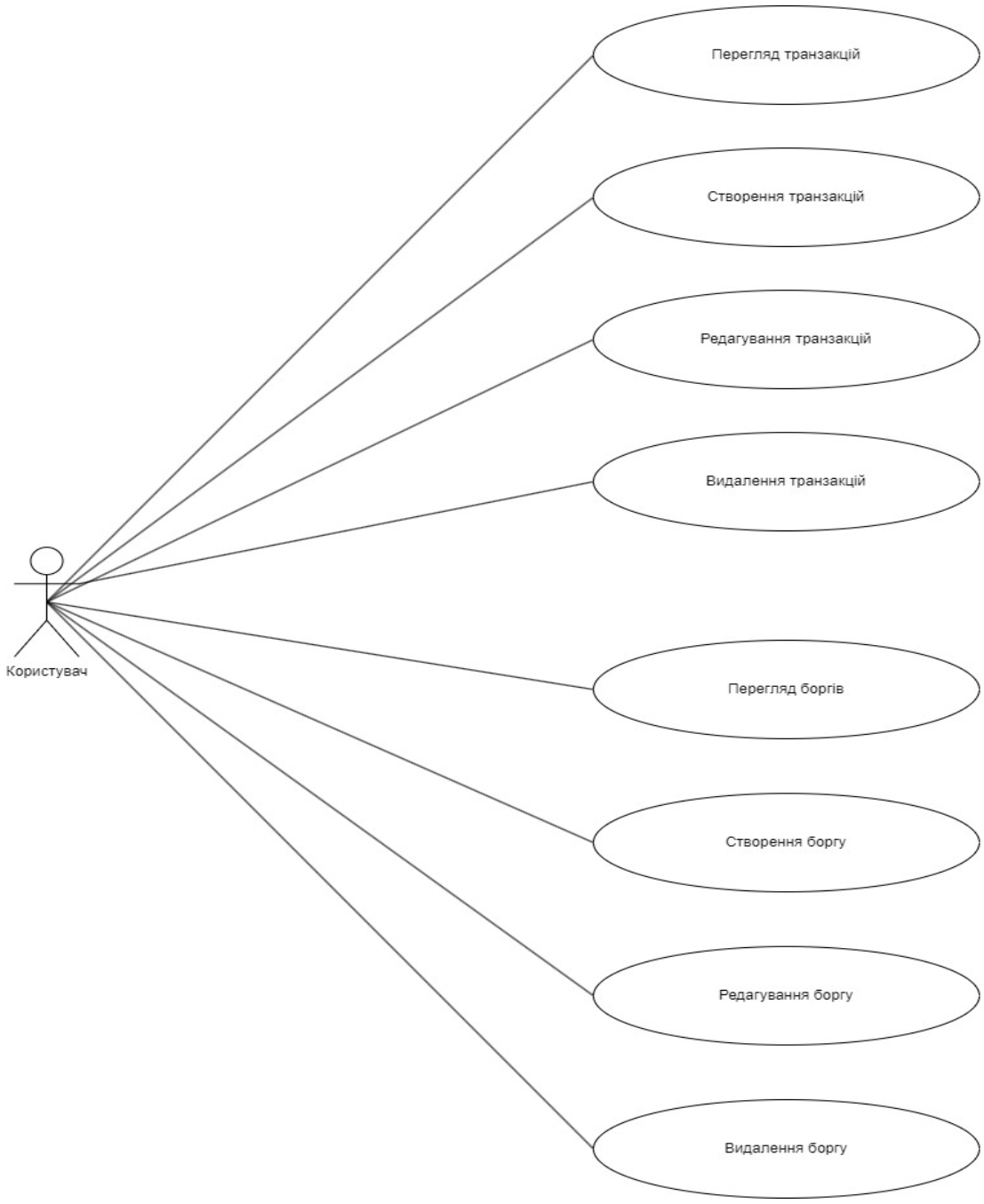

Рисунок 2.4 – Діаграма прецедентів

На рисунках 2.5 та 2.6 зображена діаграма послідовності для роботи з транзакціями. Коли користувач відкриває сторінку перегляду таблиці то frontend відправляє GET запит до API Gateway який в свою чергу відсилає запит до сервісу транзакцій по gRPC, сервіс бере дані із БД, та передає їх у шлюз він в сою чергу повертає їх у frontend в форматі json. Наприкінці frontend відображає дані. Коли користувач створює транзакцію то frontend відправляє POST запит а всі інші кроки залишаються такими ж як при отриманні.

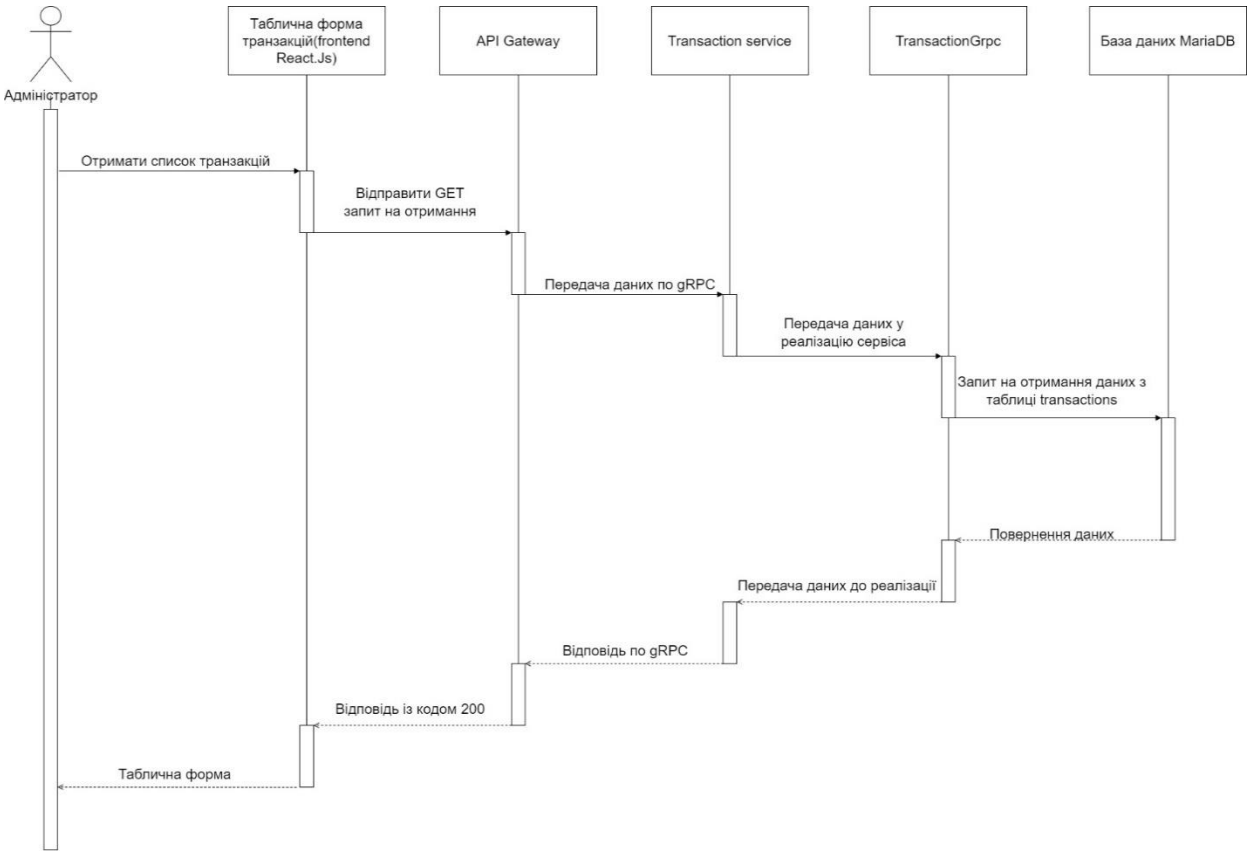

Рисунок 2.5 – Діаграма послідовності для створення транзакції

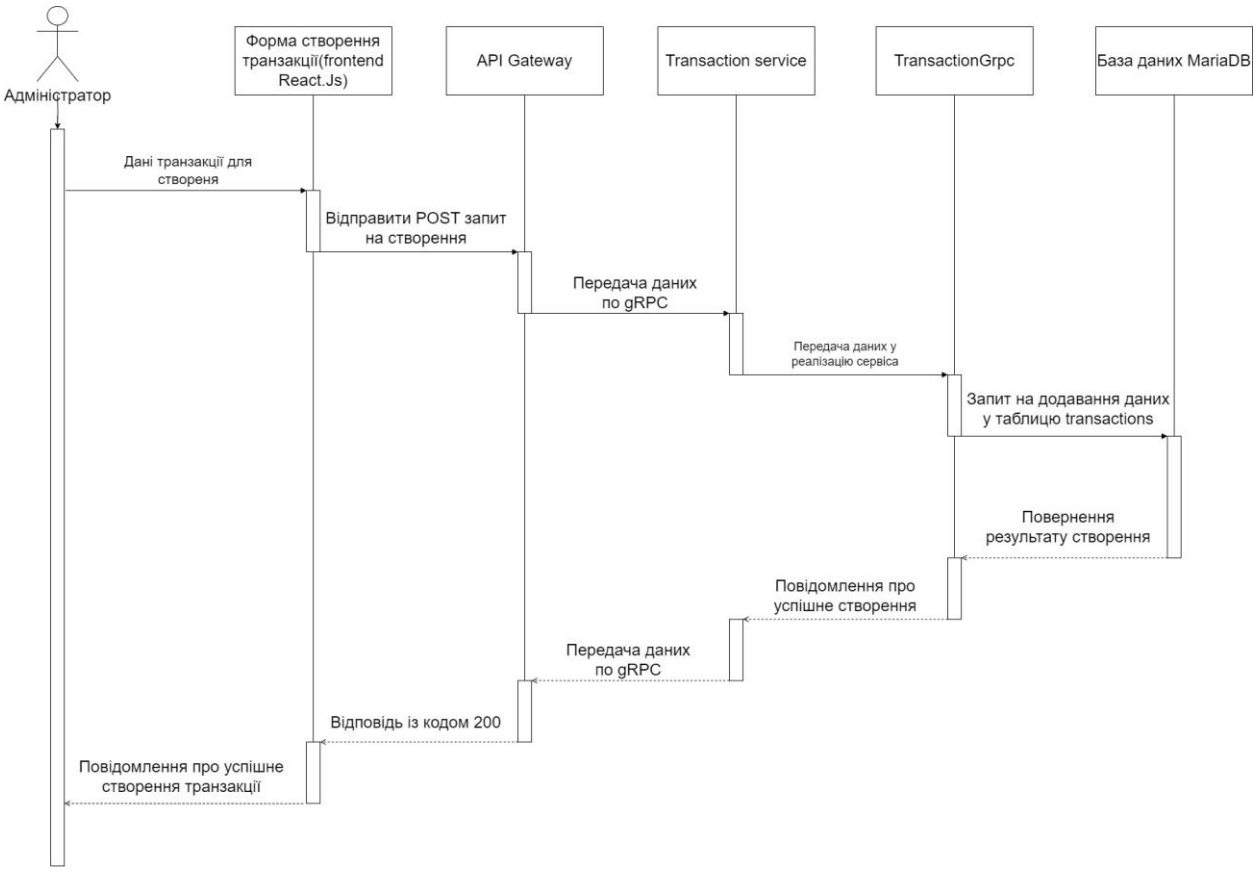

Рисунок 2.6 – Діаграма послідовності для отримання транзакцій

2.2 Етап фізичного проєктування

В цьому розділі розглянемо процес розгортання вебзастосунка за допомогою UML діаграми розгортання, з описом структурних компонентів.

На рисунку 2.7 зображена діаграма розгортання вебдодатку «Система управління особистими фінансами». На діаграмі зображені програмні компоненти «артефакти», які працюють на кожному вузлу наприклад такі як: Application Server, тобто даний вебзастосунок, та база даних з якою вебзастосунок з’єднується. Також вебзастосунок з’єднується з інтерфейсом тобто React.Js засобами REST API.

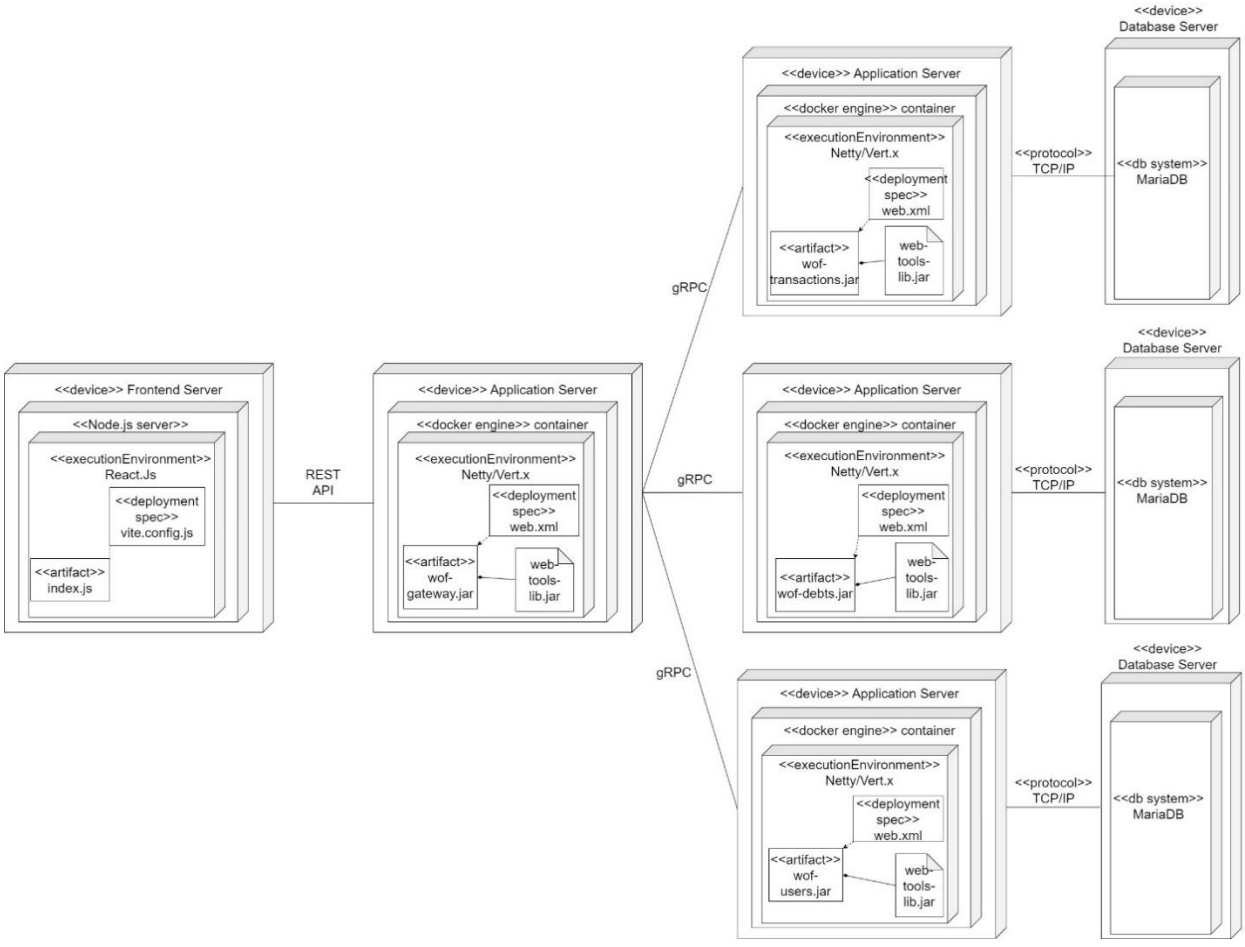

Рисунок 2.7 – Діаграма розгортання

Самі сервіси будуть розгорнені в середовищі Kubernetes з використанням Helm. Для цього потрібно згенерувати helm chart, це буде

зроблено за допомогою бібліотеки quarkus-helm. Далі будуть створені Docker-образи для кожного з них. Ці образи потрібно завантажити в Docker-реєстр (наприклад, Docker Hub або приватний реєстр).

Далі слід налаштувати Kubernetes-кластер. Це можна зробити за допомогою хмарного провайдера (наприклад, Google Kubernetes Engine, Amazon EKS, або Azure Kubernetes Service) або локально з використанням інструментів, таких як Minikube або Kind.

Щоб розгорнути мікросервіси в Kubernetes з використанням Helm, необхідно ініціалізувати Helm у кластері командою helm init, якщо використовується третя версія Helm то ініціалізація не буде потрібна так як і Tiller. Після цього можливо буде використовувати команду helm install, щоб встановити чарт і розгорнути відповідний мікросервіс у кластері. Наприклад, команда helm install wof-transactions ./wof-transactions-chart розгорне мікросервіс транзакцій з чарту.

В процесі розгортання Helm створить необхідні ресурси Kubernetes та налаштує їх відповідно до вказаних параметрів.

Масштабування мікросервісів можна здійснювати за допомогою інструментів Kubernetes, таких як горизонтальний автоскейлінг (Horizontal Pod Autoscaler) для автоматичного масштабування подів на основі навантаження.

Таким чином, використання Kubernetes та Helm значно спрощує процес розгортання, управління та масштабування мікросервісних додатків, забезпечуючи високу гнучкість і масштабованість інфраструктури.

Для подальшого управління розгорнутими мікросервісами важливо мати належні засоби для спостереження та логування. Наприклад, для централізованого логування можна використовувати стек EFK (Elasticsearch, Fluentd, Kibana), який дозволяє збирати, зберігати та аналізувати логи з усіх мікросервісів. Fluentd збирає логи з подів та відправляє їх в Elasticsearch, де вони зберігаються і можуть бути візуалізовані в Kibana.

Також для спостереження за станом кластера та мікросервісів корисно

використовувати такі інструменти, як Prometheus і Grafana. Prometheus збирає метрики з різних компонентів системи, а Grafana дозволяє створювати зручні дашборди для візуалізації цих метрик. Це допомагає вчасно виявляти проблеми з продуктивністю та стабільністю додатків.

Для забезпечення безперервної інтеграції та доставки (CI/CD) можна інтегрувати Helm з інструментами, такими як Jenkins, GitLab CI/CD або GitHub Actions. Це дозволяє автоматизувати процеси збірки, тестування та розгортання мікросервісів. Наприклад, можна налаштувати Jenkins-пайплайн, який буде автоматично збирати нові Docker-образи при кожному коміті в репозиторій, а потім оновлювати розгортання в Kubernetes за допомогою Helm.

Не менш важливим аспектом є управління конфігураціями. Kubernetes ConfigMaps та Secrets дозволяють зберігати конфігураційні дані окремо від коду додатків, що спрощує процес управління конфігураціями та підвищує безпеку. Helm підтримує роботу з цими ресурсами через шаблони, що дозволяє легко змінювати конфігурації під час розгортання.

Також важливо враховувати питання безпеки. Використання RBAC (Role-Based Access Control) в Kubernetes дозволяє налаштовувати ролі та дозволи для користувачів і сервісів, що мінімізує ризики несанкціонованого доступу. Крім того, можна використовувати інструменти для автоматичної перевірки безпеки контейнерів, такі як Aqua Security або Twistlock, для забезпечення відповідності кращим практикам безпеки.

2.3 Схема бази даних

Проєктування бази даних було виконано відповідно до практик роботи з мікросервісами та технічного завдання. Для реалізації схеми було обрано підхід: Database per service.

Database per service – це підхід у розробці програмного забезпечення, де

кожен окремий мікросервіс має свою власну базу даних [6]. На рисунку 2.8 зображений саме такий підхід. Така архітектура надає більшу гнучкість у розгортанні та масштабуванні системи, оскільки окремі компоненти можуть працювати незалежно один від одного і мати свої власні дані.

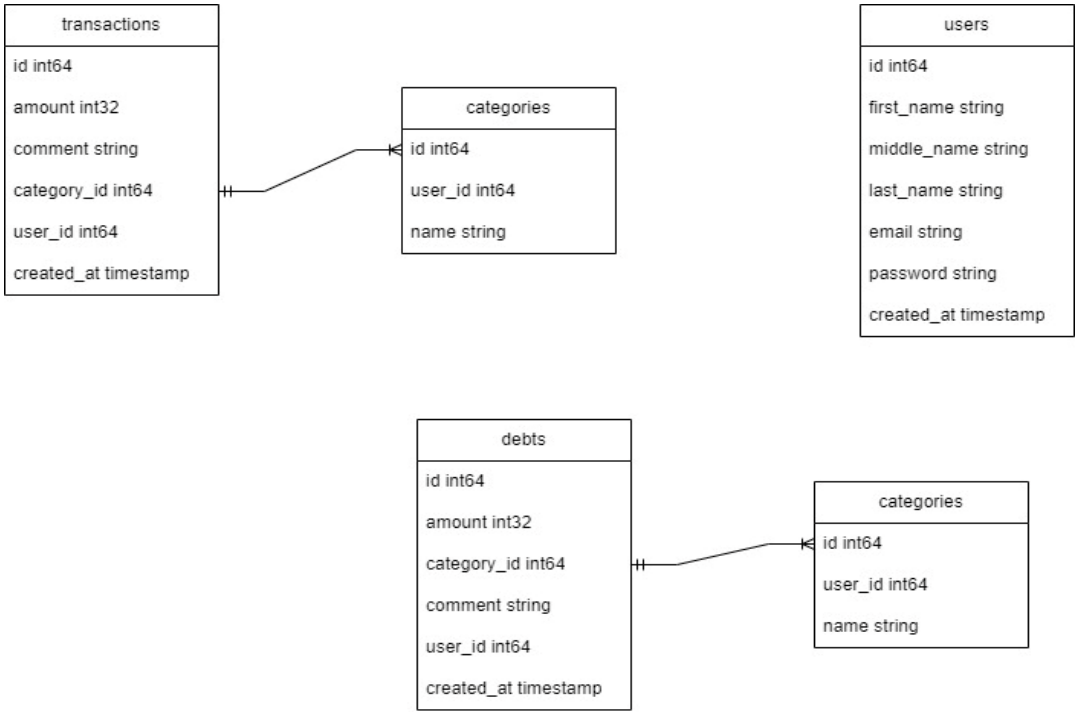

Рисунок 2.8 – ER діаграма бази даних

Переваги цього підходу полягають у зменшенні взаємозалежності між сервісами, забезпеченні високої доступності та швидкості роботи, а також у кращій гнучкості при розгортанні нових функцій та змін у системі. Такий підхід також дозволяє краще ізолювати помилки та проблеми у межах конкретного сервісу, що спрощує виявлення та виправлення неполадок.

Більш того, це дозволяє розробникам працювати над різними частинами системи паралельно, що прискорює процес розробки та впровадження нового функціоналу.

Але, такий підхід може призвести до збільшення складності управління консистентністю даних. Оскільки кожен сервіс має свою власну базу даних, виникає необхідність узгоджувати дані між цими базами, що може стати складним завданням.

Також, він може призвести до зростання витрат на утримання та підтримку, оскільки кожна база даних вимагає окремого нагляду та обслуговування.

Таким чином є 3 бази даних з даними про транзакції, борги та користувачів:

– user – дані користувачів системи;
– transaction – дані транзакцій користувачів;
– debt – дані про борги користувачів;
– category – ця сутність відповідає за збереження категорії.

2.4 Висновки до розділу 2

Отже, в ході опрацювання даного розділу був розглянутий процес перетворення вимог до розробки в послідовність проєктних рішень щодо способів реалізації вимог таких як: формування загальної архітектури програмної системи за допомогою BPMN нотацій, ER діаграми, визначення детального складу модулів кожного з архітектурних компонентів за допомогою діаграм послідовності. Також був розглянутий процес розгортання. Було визначено кроки, необхідні для налаштування серверного середовища, зокрема встановлення необхідного програмного забезпечення та налаштування баз даних. Крім того, було розглянуто питання автоматизації процесу розгортання за допомогою відповідних інструментів та скриптів, що дозволяє забезпечити більш ефективне та надійне розгортання програмної системи. Окрему увагу було приділено питанням безпеки, таким як забезпечення захищеного доступу до серверів та баз даних.

3 ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ ТА РЕЗУЛЬТАТИ ТЕСТУВАННЯ

3.1 Розробка серверної частини додатка

Розробка додатка буде поділятися на дві частини – розробка серверної частини та розробка користувацького інтерфейсу. Серверна частина буде розроблена на Quarkus, використовуючи мікросервісну архітектуру, а користувацький інтерфейс на React.

3.1.1 Створення та ініціалізація проєкту

Проєкт можливо ініціалізувати трьома шляхами. Перший: використати maven. Другий: згенерувати проєкт на code.quarkus. Третій: створити засобами IDE. Для пришвидшення роботи був обраний третій варіант. В результаті було отримано стандартну для java проєкту структуру директорій, також так як при створенні проєкту був обраний maven можливо побачити файл pom.xml. Це файл який зберігає у собі залежності та конфігурацію зборки. При використанні Quarkus налаштовувати зборку не треба, це робить quarkus-maven-plugin, його конфігурація зображена на рисунку 3.1.

```
<plugin>
  <goals>
    <goal>build</goal>
    <goal>generate-code</goal>
     <goal>generate-code-tests</goal>
  </goals>
</plugin>
```

Рисунок 3.1 – Кроки зборки Quarkus проєкту

На рисунку 3.1 зображені кроки зборки проєкту. Ці кроки будуть проходитись незалежно від того який профіль або конфігурація будуть запущені. Як можна побачити першим йде сама збірка потім генерація коду, а потім генерація тестового коду. В етапах генерації коду приймають участь такі бібліотеки як MapStruct та Lombok.

У самого плагіна зборки є 2 основні конфігурації. Перша: prod тобто будування проєкту для подальшого використання в робочому середовищі. Друга: dev для використання в середовищі для розробки. Друга конфігурація має такі переваги для розробника як гаряче перезавантаження, тобто якщо ми змінимо деякий файл та збережемо його то можна буде побачити те що поведінка додатка змінилася.

3.1.2 Схема бази даних

На основі спроєктованих ER діаграм можливо почати створення бази даних, для цього було обрано MariaDB та Liquibase.

Базу даних було запущено використовуючи Quarkus Dev Services. Quarkus підтримує автоматичне надання неналаштованих сервісів у режимі розробки та тестування. Якщо ви включите будь яке розширення яке має залежність наприклад на базу даних і не налаштуєте зовнішній datasource, Quarkus автоматично запустить відповідну службу (зазвичай за допомогою Testcontainers) і підключить вашу програму до цієї служби.

У створеному проєкті було створено директорію changelog у директорії resources. У даній директорії був створений файл корінної міграції, в ньому будуть вказані всі шляхи до інших міграцій.

На рисунках 3.2 та 3.3 зображений приклад такої міграції, як можна побачити тут вказаний includeAll. Цей атрибут вказує, що треба взяти усі файли міграцій із директорії, на рисунку це директорії transactions, categories та debts. Усі міграції будуть виконані при запуску додатка як в режимі розробника так і в робочому режимі.

```
databaseChangeLog:
  - includeAll:
      path: /db/changelog/transactions
  -includeAll:
      path: /db/changelog/categories
  -includeAll:
      path: /db/changelog/debts
```

Рисунок 3.2 – Корінна міграція

```
databaseChangeLog:
  - changeSet:
      id: "transactions-0.1.0"
      author: Nikita Volkov
      changes:
        - createTable:
            tableName: transactions
            columns:
              - column:
                  name: id
                  type: bigint
                  autoIncrement: true
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: amount
                  type: int
                  constraints:
                    nullable: false
              - column:
                  name: comment
                  type: varchar(255)
              - column:
                  name: category_id
                  type: bigint
                  constraints:
                    nullable: true
                    foreignKeyName: fk_transaction_category_id
                    references: categories(id)
```

Рисунок 3.3 – Міграція таблиці транзакцій

3.1.3 Особливості реалізації логіки вебзастосунка

В цьому розділі ми розглянемо процес реалізації основних вимог, використовуючи мікросервісну архітектуру. Головною особливістю розробки вебдодатку для контроля за особистими фінансами є вимога низької зв’язаності модулів та сервісів. Також особливістю даного вебдодатку є аутентифікація за JWT токеном, а не за логіном та паролем (так званий Basic Auth). Для реалізації таких вимог був використаний паттерн API Composition для комунікації між сервісами та шлюзом, для міжсервісної комунікації був використаний підхід Event Driven.

Паттерн API Composition дозволяє об’єднати результати викликів до різних мікросервісів у єдиний API, що спрощує комунікацію між фронтендом та бекендом [7]. Цей підхід дозволяє ефективно керувати запитами та відповідями, забезпечуючи необхідний рівень абстракції для клієнта.

Цей підхід має кілька переваг. По-перше, він зменшує кількість запитів від клієнта до сервера, що покращує продуктивність і зменшує затримки. По-друге, він спрощує логіку на стороні клієнта, зменшуючи складність коду та полегшуючи підтримку. По-третє, API Composition надає можливість централізовано керувати помилками та обробкою даних, що робить систему більш надійною.

Однак, варто враховувати і потенційні проблеми цього підходу. Наприклад, збільшення навантаження на сервер композиції, оскільки він виконує додаткову роботу з об’єднання даних. Також необхідно ретельно проєктувати API-композицію, щоб уникнути проблем із продуктивністю та забезпечити ефективну обробку великих обсягів даних.

Для міжсервісної комунікації використовується підхід Event Driven, що дозволяє мікросервісам обмінюватися повідомленнями в асинхронному режимі. Це забезпечує високу гнучкість та масштабованість системи, оскільки кожен сервіс може обробляти події незалежно від інших. Event Driven підхід також сприяє більш ефективному використанню ресурсів та підвищує надійність додатку, оскільки сервіси можуть автоматично реагувати на події,

що виникають у системі.

Крім того, для ефективного управління подіями та повідомленнями між мікросервісами використовується брокер повідомлень NATS. Такий підхід дозволяє відокремити відправника події від її отримувача, що додає гнучкості та спрощує масштабування. Популярними рішеннями для реалізації брокера повідомлень є Kafka, RabbitMQ або ActiveMQ. Вони забезпечують надійний та масштабований механізм для доставки повідомлень між мікросервісами, зберігаючи високу продуктивність системи.

Для комунікації між сервісами та шлюзом був обраний gRPC. Однією з головних переваг використання gRPC є використання Protobuf а саме його компактність та ефективність. Що дуже позитивно відображається на швидкості серіалізації та десеріалізації.

Крім того, Protobuf підтримує розширюваність структури даних без порушення зворотної сумісності. Це означає, що можна додавати нові поля до повідомлень у майбутніх версіях API без необхідності змінювати існуючі клієнти, що робить його ідеальним вибором для систем, які постійно розвиваються та оновлюються.

Protobuf має чітку схему, яка визначає структуру даних. Це дозволяє уникнути багатьох помилок на етапі компіляції, забезпечуючи високу надійність та безпеку обміну даними. Оскільки структура даних чітко визначена, розробники можуть легко зрозуміти формат даних та працювати з ними, що значно спрощує інтеграцію різних компонентів системи.

Ще однією важливою перевагою Protobuf є його підтримка багатьох мов програмування. Це дозволяє використовувати Protobuf у різних середовищах та платформах, забезпечуючи гнучкість та універсальність у розробці програмних продуктів. Завдяки своїй високій продуктивності та ефективності, Protobuf часто використовується у великих розподілених системах, де швидкість та оптимізація ресурсів є критично важливими.

Protobuf також надає можливість ефективного стискання даних, що може бути корисним у випадках, коли потрібно передавати великі обсяги

інформації через мережу. Це допомагає зменшити навантаження на мережу та покращити загальну продуктивність системи. Завдяки своїй компактності, повідомлення Protobuf займають менше місця у порівнянні з іншими форматами серіалізації, такими як JSON або XML, що особливо важливо для мобільних додатків та пристроїв з обмеженими ресурсами.

Окрім цього, Protobuf має вбудовані механізми для роботи з різними типами даних, включаючи примітивні типи, масиви, та складні структури. Це дозволяє гнучко моделювати різні сценарії використання та адаптувати формат даних під конкретні потреби проєкту.

На рисунках 3.4 та 3.5 продемонстровані .proto файли. Використовуючи Protobuf, можна визначати структури даних і генерувати код для серіалізації та десеріалізації цих даних у різних мовах програмування, включаючи Java. Протокомпілятор (прото-компілятор) є ключовим інструментом в цьому процесі, конвертуючи .proto файли у відповідний код та методи [8].

```
syntax = "proto3";
package transactions;

option java_multiple_files = true;
option java_package = "com.wof.shared";
option java_outer_classname = "TransactionMessages";
import "google/protobuf/timestamp.proto";

message Transaction {
  int64 id = 1;
  int32 amount = 2;
  string comment = 3;
  int64 categoryId = 4;
  int64 userId = 5;
  google.protobuf.Timestamp createdAt = 6;
}

message TransactionsResponse {
  int64 pages = 1;
  repeated Transaction transactions = 2;
}
```

Рисунок 3.4 – Proto файл транзакцій

```
syntax = "proto3";

package transactions.service;
option java_multiple_files = true;
option java_package = "com.wof.shared";
option java_outer_classname = "TransactionService";

import "transactions/message.proto";
import "base.proto";
service Transactions {
  rpc CreateTransaction (Transaction) returns (Transaction) {}
  rpc UpdateTransaction (Transaction) returns (Transaction) {}
  rpc GetTransaction (base.IdRequest) returns (Transaction) {}
  rpc GetTransactions (base.PaginationRequest) returns (TransactionsResponse) {}
  rpc DeleteTransaction (base.IdRequest) returns (Transaction) {}
}
```

Рисунок 3.5 – Proto файл сервіса транзакції

На рисунку 3.6 продемонстрований метод генерації JWT токена при вході в систему користувача. Цей метод повертає згенерований токен який в подальшому буде міститись у куках браузера користувача. Структура JWT токена виглядає так: email; id; дата випуску; той хто виписав; алгоритм шифрування.

JWT (JSON Web Token) має кілька переваг, що робить його популярним вибором для аутентифікації та авторизації. Перш за все, безпека є однією з головних причин використання JWT. Він дозволяє використовувати сучасні алгоритми шифрування для захисту даних, що гарантує, що дані в токені неможливо змінити без виявлення, оскільки будь-яка зміна інвалідізує підпис.

Масштабованість також є важливою перевагою. Оскільки JWT є самодостатнім токеном, він не потребує збереження сесій на сервері. Це дозволяє масштабувати додаток без необхідності синхронізації сесій між серверами. Простота використання також додає вагомих аргументів на користь JWT. Він легко інтегрується в різні платформи і підтримує різноманітні бібліотеки для роботи з токенами, що робить його зручним для

розробників.

Продуктивність є ще однією перевагою JWT. Оскільки перевірка токена відбувається швидко, це зменшує навантаження на сервер і покращує загальну продуктивність системи. Універсальність JWT полягає в тому, що його можна використовувати не лише для аутентифікації, але й для передачі будь-якої іншої інформації, що потребує захисту і перевірки. Це може бути корисним для обміну даними між різними сервісами.

```
public String generateToken(List<String> permissions, String email, Long id) {
    String privateKeyLocation = "/certs/privatekey.pem";
    PrivateKey privateKey = readPrivateKey(privateKeyLocation);
    JwtClaimsBuilder claimsBuilder = Jwt.claims();
    long currentTimeInSecs = currentTimeInSecs();
    Set<String> groups = new HashSet<>(permissions);
    claimsBuilder.issuer(issuer);
    claimsBuilder.claim("id", id);
    claimsBuilder.subject(email);
    claimsBuilder.issuedAt(currentTimeInSecs);
    claimsBuilder.expiresAt(currentTimeInSecs + duration);
    claimsBuilder.groups(groups);
    return claimsBuilder.jws().keyId(privateKeyLocation).sign(privateKey);
}
```

Рисунок 3.6 – Метод генерації JWT токена

3.2 Особливості реалізації інтерфейсу вебзастосунку

В цьому розділі ми розглянемо процес створення інтерфейсу вебзастосунка засобами React.Js.

Інтерфейс взаємодіє з логікою засобами REST API за допомогою json, та пакету RTK Query, методи зображені на рисунках 3.7, 3.8 та 3.9. У сховище Redux зберігається серіалізований результат виконання методів які відправляють об’єкт предмета у backend асинхронно, ці методи в залежності

від відповіді бекенда [2], або викликає оновлення даних, або викликає відображення вікна з помилкою. Також так як більшість маршрутів по котрих відправляється цей запит вимагає авторизованого в системі користувача, то з кожним запитом потрібно передавати в заголовку JWT токен із кукі браузера.

```
createTransaction: builder.mutation<Transaction, Partial< Transaction >>({
    query: (body) => ({
      url: "/transactions",
      method: "POST",
      body: body,
    }),
    invalidatesTags: ["Transaction"],
}),
```

Рисунок 3.7 – Метод створення транзакції

```
getTransactions: builder.query<Transaction[], GetTransactionsRequest>({
    query: (params) => ({
      url: "/transactions",
      method: "GET",
      params: params,
    }),
    providesTags: ["Transaction"],
}),
```

Рисунок 3.8 – Метод отримання транзакцій

```
updateTransaction: builder.query<Transaction, Partial<Transaction> & Pick<Transaction,
“id”>>{
    query: ({ id, …body }) => ({
      url: "/transactions/${id}",
      method: "PUT",
      body: body,
    }),
    providesTags: ["Transaction"],
}),
```

Рисунок 3.9 – Метод оновлення транзакції

На рисунку 3.10 зображена сторінка логіна користувача, для того щоб увійти користувач повинен ввести коректні дані, тоді frontend викликає метод який асинхронно відправляє до бекенду POST запит. У відповідь прийде JWT токен якщо дані вірні, а якщо ні то буде виведена помилка як на рисунку 3.11.

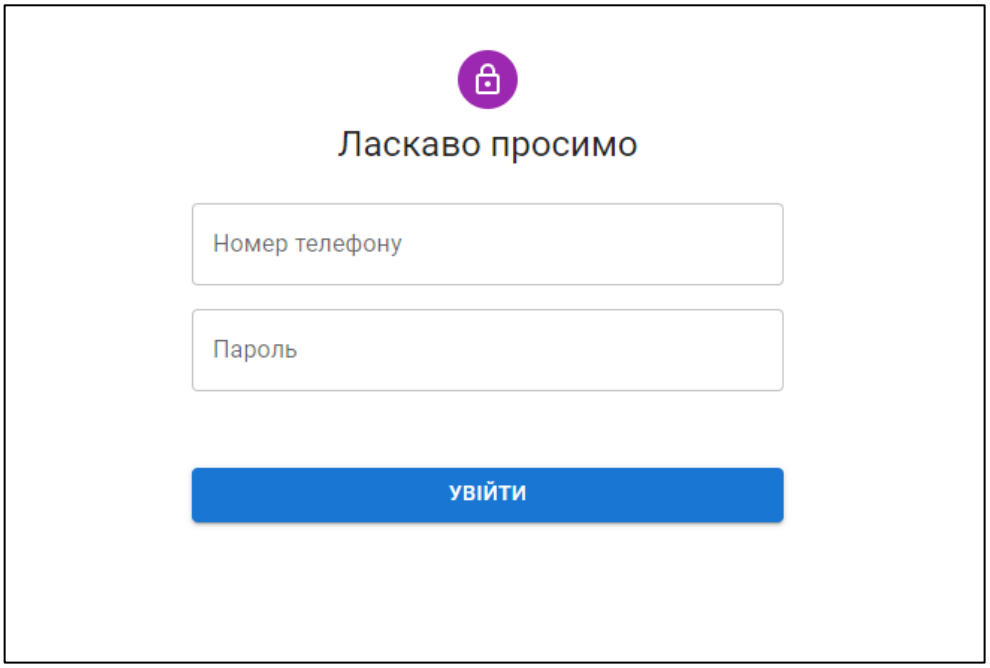

Рисунок 3.10 – Сторінка логіну користувача

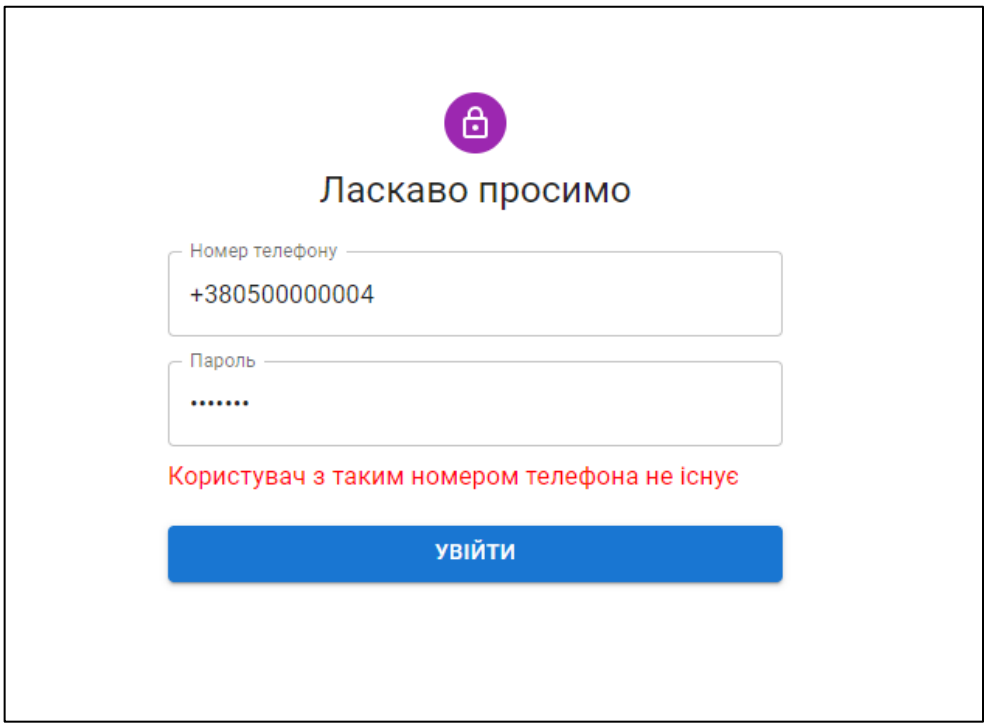

Рисунок 3.11 – Повідомлення про помилку із бекенду

На рисунку 3.12 зображено повідомлення про помилку при перевірці даних. Ця перевірка відбувається перед відправкою запиту на бекенд. Таким чином навантаження на API знижується тому що йому не потрібно додатково перевіряти окрім існування користувача, ще і формат даних у запиті. Для цього були використані такі бібліотеки як react hook form та zod.

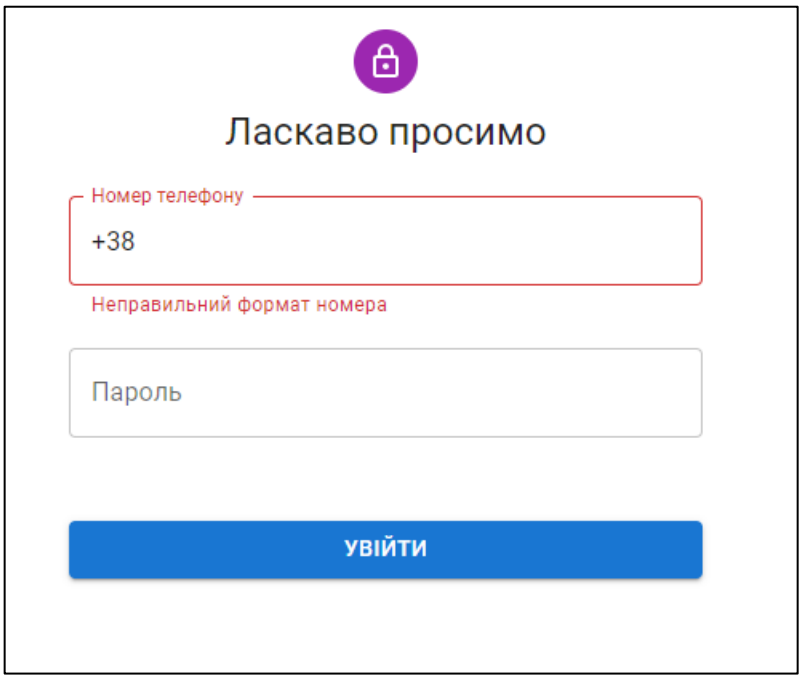

Рисунок 3.12 – Повідомлення про помилку при перевірці даних

React Hook Form забезпечує легку та ефективну інтеграцію з React-компонентами, дозволяючи розробникам створювати складні форми з мінімальним кодом і без використання класових компонентів. Це сприяє кращій продуктивності, оскільки зменшується кількість повторних рендерів і покращується загальна швидкість додатку. Крім того, React Hook Form підтримує асинхронну валідацію та дозволяє легко інтегрувати сторонні бібліотеки.

Zod забезпечує потужний і зручний спосіб для валідації даних. Завдяки своїй декларативній природі, Zod дозволяє розробникам легко створювати схеми для валідації структури даних, що робить код чистішим і зрозумілішим. Поєднання React Hook Form і Zod дозволяє зручно керувати станом форми і

валідацією, підвищуючи надійність і зручність роботи з формами. Разом ці бібліотеки створюють потужний інструментарій для розробки форм, забезпечуючи високу продуктивність, гнучкість і зручність у використанні.

3.3 Перевірка кожного сервіса та способів їхньої інтеграції

В цьому розділі ми розглянемо процес перевірки кожного з модулів вебзастосунку. Тестування було розділено на два етапи: Автоматичне за допомогою unit-тестів із бібліотеки JUnit 5. Мануальне – ручна перевірка працездатності кожного з модулів.

JUnit 5 був обраний через його численні переваги, які роблять процес тестування зручнішим і ефективнішим. Насамперед, JUnit 5 відрізняється своєю модульною архітектурою, яка складається з трьох основних модулів: JUnit Platform, JUnit Jupiter та JUnit Vintage [4]. Це дозволяє розробникам вибирати ті компоненти, які найкраще відповідають їхнім потребам, і з легкістю інтегрувати JUnit 5 у свої проєкти.

JUnit 5 підтримує нові можливості Java, такі як віртуальні потоки, що дозволяє писати більш ефективний і компактний код тестів. Завдяки цьому, тестування стає більш інтуїтивно зрозумілим, а сам код легше підтримувати і розширювати. Крім того, JUnit 5 забезпечує більш гнучке керування життєвим циклом тестів, що дозволяє задавати передумови та дії після тестів на різних рівнях – як для всього класу, так і для окремих методів.

Ще однією важливою перевагою JUnit 5 є розширені можливості асерцій і очікуваних винятків. Завдяки цьому, розробники можуть легко перевіряти результати виконання тестів і переконуватись у правильній роботі коду. Нові методи асерцій дозволяють робити більш точні перевірки і надавати детальніші повідомлення про помилки, що значно полегшує діагностику проблем.

Крім того, JUnit 5 [9] має потужний механізм розширень, який дозволяє

додавати власні функціональні можливості або використовувати вже існуючі розширення, такі як Mockito для мокування або Quarkus Test для інтеграційних тестів. Це робить JUnit 5 дуже гнучким і дозволяє адаптувати його під конкретні потреби проєкту.

Ще одна важлива особливість JUnit 5 – це підтримка паралельного виконання тестів, що дозволяє значно прискорити процес тестування великих проєктів. Це особливо корисно в умовах CI/CD, де швидкість виконання тестів може мати вирішальне значення.

Окрім технічних переваг, JUnit 5 активно підтримується спільнотою розробників, що забезпечує регулярні оновлення та виправлення помилок. Це означає, що використовуючи JUnit 5, розробники можуть бути впевнені в стабільності та надійності цього інструменту.

Таким чином, використання JUnit 5 надає розробникам широкі можливості для ефективного тестування, забезпечує зручність і гнучкість у роботі з тестами, а також дозволяє легко інтегрувати інші інструменти і розширення. Це робить його оптимальним вибором для сучасних Java-проєктів.

Автоматичні тести покривають такі модулі як: користувач, транзакції, борги. Перевірялись усі типи дій такі як: додавання, редагування, видалення, перегляд інформації. Перед кожним тестом за допомогою анотації @BeforeAll були створені залежності в вигляді певних об’єктів. Після кожного тесту за допомогою анотації @AfterAll та метода cleanup будуть видалені всі тестові поля у базі даних. На рисунку 3.13 зображені результати автоматичного тестування за допомогою JUnit.

Для забезпечення точності тестів були використані мок-об’єкти, які дозволяють імітувати поведінку реальних об’єктів у контрольованих умовах. Це дозволяє ізолювати тестований код і перевірити його роботу незалежно від зовнішніх факторів [9]. Крім того, для кожного тестового випадку були створені окремі сценарії, що описують конкретні умови та очікувані результати.

Тести також включають перевірку валідації даних, щоб упевнитись, що

введені користувачем дані відповідають встановленим правилам та обмеженням. Це включає перевірку коректності формату, обов’язковості полів та унікальності значень.

Для аналізу результатів тестів використовується спеціалізований інструмент maven-surefire, який надає детальну інформацію про кожен тестовий випадок, включаючи час виконання, статус та можливі помилки.

Рисунок 3.13 – Результати автоматичного тестування

На рисунках 3.14 та 3.15 зображена реалізація інтеграційних тестів вдалого створення за допомогою gRPC, тобто для того, щоб тест рахувався успішно виконаним, після створення транзакції або боргу, повинен повернутися створений об’єкт з id який не дорівнює 0.

```
@Test
public void transactionCreateTest() {
    Transaction request = Transaction.newBuilder()
            .setAmount(12000)
            .setComment(“Sample transaction”)
            .setCategoryId(1L)
            .build();

    Transaction response = transactionGrpc.createTransaction(request);
    reportId = response.getId();
    Assertions.assertTrue(response.getId() != 0L);
}
```

Рисунок 3.14 – Тест створення транзакції

```
@Test
public void debtCreateTest() {
   Debt request = Debt.newBuilder()
         .setAmount(12000)
         .setComment(“Sample Debt”)
         .setCategoryId(1L)
         .build();

   Transaction response = debtGrpc.createDebt(request);
   reportId = response.getId();
   Assertions.assertTrue(response.getId() != 0L);
}
```

Рисунок 3.15 – Тест створення боргу

Під час такого тесту запускається сервіс та клієнт gRPC, сервіс у свою чергу запускається у робочому режимі, а клієнт у тестовому, тобто без вказання ip адреси. Клієнт робить реальний запит до сервера для створення, сервер в свою чергу звертається до бази даних додаючи новий запис.

Також були проведені unit тести, за допомогою Mockito вийшло імітувати роботу сервера, таким чином справжній запит не був відправлений та опрацьований, а в базу даних не надійшло нового запиту.

Unit тестування за допомогою Mockito дозволило ізолювати тестування бізнес-логіки від залежностей на зовнішні сервіси та базу даних. Це зробило можливим зосередитися на функціональності, яку безпосередньо розробляли, без ризику змінити або пошкодити дані. Імітація серверних запитів також дала змогу значно прискорити тестування, оскільки не було необхідності чекати відповіді від справжнього сервера, та повертати базу даних в стан до проведення тестування.

Мануальне тестування – покриває такий етап, як тестування програмного продукту в цілому, та такі речі котрі неможливо перевірити засобами автоматичного тестування, навіть такими як Selenium, наприклад верстка, або адаптація під мобільні пристрої. Мануальним тестуванням займається розробник, та на більш пізніх етапах QA-Інженер. У даному

вебзастосунку, мануальне тестування було використано на фінальній стадії розробки а саме для тестування інтерфейсу, та таких ситуацій коли була повинна відобразитись помилка. Наприклад на рисунку 3.11 було зроблено мануальне тестування логіна користувача.

3.4 Висновки до розділу 3

Отже, в ході опрацювання даного розділу були перевірені усі модулі за допомогою автоматичного та мануального тестування. Розглянуті особливості реалізації інтерфейсу та логіки вебзастосунку, а саме: відображення помилок, посторінкове відображення великої кількості об’єктів, перевірка вхідних даних, та генерація JWT токена. Особливу увагу було приділено безпеці додатку, зокрема захисту від SQL-ін’єкцій, XSS-атак та інших загроз, що можуть виникнути під час експлуатації. Було реалізовано багаторівневу систему аутентифікації та авторизації користувачів, що забезпечує надійний захист персональних даних.

ВИСНОВКИ

В даній кваліфікаційній роботі було розглянуто принципи роботи з такими фреймворками як Quarkus, та React.Js для створення вебзастосунка «Система управління особистими фінансами».

Під час роботи було реалізовано такі задачі: проаналізовано предметну область її сутності та атрибути. Спроєктовано UML діаграми такі як: діаграма розгортання, діаграма прецедентів. Реалізовано backend вебзастосунка, за допомогою Quarkus. Реалізована система авторизації за допомогою JWT токенів, та систему аутентифікації за допомогою Quarkus-Security. Реалізовано frontend вебзастосунка, за допомогою React.Js. Реалізовано базу даних засобами системи управління базами даних MariaDB. Протестовано базу даних. Протестовано backend вебзастосунка, мануальним методом, та за допомогою такого інструмента як Junit, результати тестування зображені на рисунку 3.13. Протестовано frontend вебзастосунка.

Під час виконання теоретичної частини роботи було досліджено обрану предметну область, після чого було досліджено методи створення, та проєктування вебзастосунка, засобами Quarkus, React.Js, та REST API.

Під час виконання практичної частини роботи було створено діаграму класів, діаграму розгортання, діаграму прецедентів, діаграму послідовності. На основі UML діаграм був створений вебдодаток. Та реалізована база даних, за допомогою міграцій Liquibase.

Була вирішена актуальна проблема контролювання своїх транзакцій розділених на категорії, також можливо керувати своїми боргами.

Крім того, у процесі роботи над проєктом були розглянуті сучасні підходи до безпеки вебзастосунків, зокрема використання методів шифрування даних та захисту від SQL-ін’єкцій. Впроваджено механізми обробки помилок для відслідковування роботи системи та своєчасного реагування на можливі збої.

Для зручності користувачів було розроблено інтуїтивно зрозумілий інтерфейс, що дозволяє швидко і легко здійснювати операції з управління особистими фінансами. Застосовано принципи адаптивного дизайну, що забезпечує коректне відображення інтерфейсу на різних пристроях.

Виконані роботи дозволили створити функціональний та надійний вебзастосунок, який допомагає користувачам ефективно контролювати свої фінанси, планувати витрати та аналізувати фінансову активність. Подальший розвиток проєкту може включати впровадження додаткових функцій, таких як інтеграція з банківськими системами для автоматичного імпорту транзакцій, додаткові аналітичні інструменти для прогнозування фінансового стану та розширені можливості налаштування системи під індивідуальні потреби користувачів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Learn React. URL: https://react.dev (дата звернення: 13.02.2024).
2. RTK Query Quick Start. URL: https://redux-toolkit.js.org/tutorials/rtk-query (дата звернення: 21.02.2024).
3. Hibernate ORM Getting started guide. URL: https://hibernate.org/orm/documentation/6.6/ (дата звернення: 16.02.2024).
4. Фаулер М., Бек К. Рефакторинг поліпшення проекту існуючого коду / перекл. з англ. І. Красикова. Діалектика, 2016. 448 с.
5. Liquibase docs. URL: https://docs.liquibase.com/home.html (дата звернення: 25.03.2024).
6. Макконнелл С. Ідеальний код. США: Microsoft Press, 2004. 960 с.
7. Йенер М., Фідом А. Java EE патерни проектування для професіоналів. США: Wrox, 2015. 264 с.
8. GRPC Reference Guide. URL: https://quarkus.io/guides/grpc-reference (дата звернення: 02.04.2024).
9. Quarkus testing your application. URL: https://quarkus.io/guides/getting-started-testing (дата звернення: 05.04.2024).
10. Quarkus Continuum. URL: https://quarkus.io/continuum/ (дата звернення: 20.04.2024).