Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Пояснювальна записка

до магістерської кваліфікаційної роботи
магістр
(освітньо-кваліфікаційний рівень)

на тему: Розробка методів та засобів підвищення рівня захисту інформаційних систем

Виконав: студент II курсу
групи 2ПІ-18м
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Микитюк Іван Сергійович
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ
Романюк Оксана Володимирівна
(прізвище та ініціали)

ВНТУ – 2019

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Освітньо-кваліфікаційний рівень – магістр
Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.
"____"_____2019 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Микитюку Івану Сергійовичу

1. Тема роботи – розробка методів та засобів підвищення рівня захисту інформаційних систем.

Керівник роботи: Романюк Оксана Володимирівна, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від "__"_____2019 року №___

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи: Операційна система – Windows, MacOS; Мова програмування – JavaScript; Серверний рушій – NodeJS; Фреймворки – VueJS, NestJS.

4. Зміст розрахунково-пояснювальної записки: вступ; аналіз предметної галузі; аналіз проблематики; проектування багатошарового захисту функцій у СКБД; розробка методу відслідковування несанкціонованих змін; програмна реалізація методів та засобів у вигляді спрощеної системи керування базами даних; тестування функцій розробленого програмного засобу та їх захищеності; економічна частина, висновки, перелік посилань, додатки.

5. Перелік графічного матеріалу: структурна схема роботи рівнів захисту; схема роботи алгоритму JWT; структурна схема роботи методу перевірки несанкціонованих змін; інтерфейс системи керування базами даних.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Романюк О.В., к.т.н., доцент кафедри ПЗ		
5	Бальзан М.В., к.е.н., доцент кафедри ЕПВМ		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної Роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної галузі та порівняльна характеристика сучасних СКБД	04.09.2019 – 30.09.2019	Вик.
2	Проектування багатошарового захисту	30.09.2019 – 27.10.2019	Вик.
3	Програмна реалізація методів та засобівв захисту	28.10.2019 – 10.11.2019	Вик.
4	Тестування розробленого програмного засобу	11.11.2019 – 15.11.2019	Вик.
5	Економічна частина	16.11.2019 – 21.11.2019	Вик.

Студент _____ **Микитюк І.С.**
(підпис) (прізвище та ініціали)

Керівник магістерської кваліфікаційної роботи _____ **Романюк О.В.**
(підпис) (прізвище та ініціали)

АННОТАЦІЯ

У магістерській кваліфікаційній роботі розроблено модель розшарування доступу до інформації, що зберігається в сучасних інформаційних системах за рахунок розмежування користувацького доступу до різних рівнів керування базою даних, що дозволило підвищити рівень захищеності даних та функцій сучасних СКБД в цілому. Розроблено метод виявлення несанкціонованих дій при роботі з базою даних, який дозволяє виявити дії зловмисників чи користувачів без належного рівня доступу.

Проведені експериментальні дослідження дають підставу вважати, що розроблені засоби підвищують ефективність захисту сучасних баз даних, та дозволяють зменшити вплив зловмисників на інформацію, яка в них зберігається.

Розроблено програмний засіб (систему керування базами даних) в якій реалізовано модель багатошарового доступу до функцій та інформації у БД та метод виявлення несанкціонованих дій. Програмний засіб розроблявся з використанням мови програмування JavaScript, фреймворків VueJS та серверного рушія NodeJS. Система керування базами даних може працювати під керівництвом операційної системи Windows, MacOS та Linux.

АННОТАЦИЯ

В магистерской квалификационной работе была разработана модель расслоения доступа к информации, хранящейся в современных информационных системах за счет разграничения пользовательского доступа к различным уровням управления базой данных, что позволило повысить уровень защищенности данных и функций современных СУБД в целом. Разработан метод выявления несанкционированных действий при работе с базой данных, который позволяет выявить действия злоумышленников или пользователей без должного уровня доступа.

Проведенные экспериментальные исследования дают основание считать, что разработанные средства повышают эффективность защиты современных баз данных, и позволяют уменьшить влияние злоумышленников на информацию, которая в них хранится.

Разработана программа (система управления базами данных) в которой реализована модель многослойного доступа к функциям и информации в БД и метод выявления несанкционированных действий. Программное средство разрабатывался с использованием языка программирования JavaScript, фреймворка VueJS и серверного двигателя NodeJS. Система управления базами данных может работать под управлением операционной системы Windows, MacOS и Linux.

ABSTRACT

The master's qualification describes the development of a model of bundle access to information and functions of modern information systems by distinguishing user access to different levels of database management. Due to this, the level of security of data and functions of modern DBMS has been improved. A method of detecting unauthorized actions when working with a database has been developed, which allows to detect the actions of malicious users or users without proper access level.

Experimental studies have led to the conclusion that the developed tools increase the efficiency of protection of modern databases, and allow to reduce the influence of attackers on the information stored in them.

Software (database management system) developed. It contains a model of multilayered access to information in the database and a method of detecting unauthorized actions. The software was developed using JavaScript programming language, VueJS framework and NodeJS server engine. The database management system can run Windows, MacOS and Linux.

ЗМІСТ

ВСТУП

Актуальність теми. У ході збільшення та поширення інформації, якою може володіти людина у неї з’явилась необхідність обробки, структуризації та зберігання інформації. Відповідно до області використання даної інформації вона набуває великої цінності і втрата чи несанкціонований доступ до неї можуть нести важкі матеріальні наслідки для власника.

Бази даних, як один з найпопулярніших способів збереження та маніпулювання користувацькою інформацією набули широкого попиту у сучасному суспільстві і з розвитком інформаційних технологій використання баз даних стало доступне не тільки спеціалістам, а й звичайним користувачам. У зв’язку з цим перед адміністраторами сучасних систем керування базами даних постало питання про доопрацювання політики безпеки згідно з напливом низько кваліфікованих користувачів систем керування базами даних.

У додаток до розвитку інформаційних технологій та потреб користувачів йде розвиток методів та технологій, які використовуються зловмисником, і спричиняють зростання ризиків втрати автентифікаційних даних користувачів у СКБД. Сучасні СКБД мають низку криптографічних методів, які можуть шифрувати та гешувати інформацію, в додаток до цього вони мають низку прав, які відповідають за регулювання доступу до інформації, а також налаштування, стосовно рольового доступу, проте вони мають проблему, яка зв’язана з отриманням авторизованим користувачем відповідних йому прав, а саме ті випадки, коли зловмисник отримав необхідну йому автентифікаційну інформацію.

У ході дослідження стану сучасних СКБД, було виявлено, що в більшості з них використовується дискреційна модель надання доступу користувачам та адміністраторам, що збільшує ризик компрометації автентифікаційних даних (підглядування чи підбір паролю). Тому можна виділити, що основною проблемою, яку потрібно вирішити є несанкціонований доступ до даних та функцій у сучасних інформаційних системах через «людський фактор».

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалась відповідно до плану науково-дослідних робіт кафедри програмного забезпечення.

Об'єктом дослідження є процес захисту інформації та доступу до функцій в сучасних інформаційних системах на прикладі систем керування базами даних.

Предметом дослідження є методи та засоби захисту сучасних інформаційних систем.

Мета та задачі дослідження. Метою роботи є підвищення рівня захищеності інформації у сучасних інформаційних системах за рахунок розробки моделі багатошарового доступу до функцій СКБД та методу відслідковування несанкціонованих дій у СКБД.

Основними **задачами дослідження** є:

- провести аналіз проблем сучасних систем керування базами даних;
- розробити моделі багатошарового доступу до функцій СКБД;
- розробити алгоритм функціонування модулів «шарів» програмного засобу;
- розробити метод відслідковування несанкціонованих дій в СКБД.

Методи дослідження. У процесі дослідження застосовувалися: теорія інформації, теорія захисту інформації в автоматизованих системах, реляційна алгебра, криптографія.

У процесі дослідження також були застосовані сучасні методи створення web-додатків, серверних додатків та адміністрування та захисту баз даних.

Наукова новизна одержаних результатів полягає у тому, що:

1. Вперше запропоновано модель розшарування доступу до інформації, що зберігається в сучасних інформаційних системах, особливістю якого є розмежування користувацького доступу до різних рівнів керування базою даних, що дозволило підвищити рівень захищеності даних та функцій сучасних СКБД.

2. Подальшого розвитку отримав метод виявлення несанкціонованих дій при роботі з базою даних, який на відмінну від інших дозволяє виявити несанкціоновані дії зловмисників чи користувачів без належного рівня доступу.

Практичне призначення одержаних результатів полягає у тому, що на основі теоретичних досліжень:

- було розроблено програмний засіб, що містить в собі модель багатошарового доступу до різних рівнів функціоналу в сучасних СКБД;
- розроблено програмну систему відслідковування несанкціонованого доступу до методів СКБД.

Достовірність теоретичних досліджень, наукових положень і висновків, викладених у магістерській кваліфікаційній роботі, підтверджуються збігом аналітично отриманих результатів із результатами експерементальних досліджень.

Апробація матеріалів магістерської кваліфікаційної роботи. Основні положення магістерської кваліфікаційної роботи доповідалися та обговорювалися на конференціях:

- Науково-технічна конференція підрозділів Вінницького національного технічного університету. Факультет інформаційних технологій та комп’ютерної інженерії. Вінниця, 2018.
- Міжнародна науково-практична конференція «Інформаційні технології та комп’ютерне моделювання», Івано-Франківськ, 2018[5].

Публікації. Основні результати досліджень опубліковано в 2 наукових працях, у тому числі 2 – у матеріалах конференцій.

Особистий внесок магістранта. Усі результати, що складають основний текст роботи, отримані автором самостійно. У публікаціях, написаних у співавторстві, автору належитьпорівняльний аналіз сучасних систем керування базами даних та розробка моделі багатошарового доступу до функцій СКБД.

Структура та обсяг роботи.

Магістерська кваліфікаційна робота складається з п’яти розділів. У першому розділі проводи ться техніко-економічне обґрунтування актуальності

та доцільності розробки. Виконується аналіз стану проблеми, досліджуються існуючі аналоги та їх проблеми. Розглядаються методи розв’язання поставленої задачі та формулюються основні завдання розробки.

У другому розділі виконується розробка багатошарової моделі та обгрунтування методів покращення захисту інформації у сучасних СКБД. Зокрема, розробляються та узгоджуються три рівні користувацького доступу, які мають окремі автентифікаційні та криптографічні методи.

У третьому розділі наведено інформацію про процес розробки різних рівнів захисту та використовувані на них криптографічні методи, аналіз та обґрунтування вибору мови програмування та криптографічні методи. Розроблено модель використання шифрування та гешування інформації в комірках, гешування автентифікаційних даних, аналіз та обґрунтування вибору мови програмування та інтегрованого середовища розробки.Також представлено опис створення візуальної та програмної частин СКБД.

У четвертому розділі розглядаються існуючі методики тестування та виконується тестування розробленої системи керування базами даних.

П’ятий розділ містить економічні розрахунки, які підтверджують доцільність розробки.

У додатки винесено технічне завдання, акт впровадження, авторське право на комп’ютерну програму, лістинг вихідного коду та ілюстративний матеріал до захисту магістерської кваліфікаційної роботи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА СУЧАСНИХ СКБД

Перед початком проектування будь-якої системи захисту програмного забезпечення [6] треба цілком чітко собі уявляти, що саме і від кого планується реалізовувати захист. Не існує абсолютно стійких методів захисту. Кваліфіковані системні програмісти, що користуються сучасними засобами аналізу програмного забезпечення (налагоджувачі, дизасемблери, перехоплювачі переривань і т. п.), маючи досить часу, зможуть подолати практично будь-який захист. І тому при проектуванні систем захисту слід передбачати те, що рано чи пізно цей захист буде знято [7].

1.1 Визначення бази даних та системи керування базами даних

База даних (БД) - інформаційна модель, що дозволяє в упорядкованому вигляді зберігати дані про групу об'єктів з однаковим набором властивостей або пойменовану сукупність структурованих даних (пойменована сукупність структурованих даних предметної області) [8]. В загальному випадку база даних містить схеми, таблиці, подання, збережені процедури та інші об'єкти. Дані у базі організовують відповідно до моделі організації даних. Таким чином, сучасна база даних, крім самих даних, містить їх опис та може містити засоби для їх обробки.

В загальному випадку базою даних можна вважати будь-який впорядкований набір даних. Наприклад, паперову картотеку з формулярами про працівників підприємства у відділі кадрів. Але зараз мова йде про використання баз даних в інформаційних системах.

База даних орієнтована на інтегровані запити, а не на одну програму, як у випадку файлового підходу, і використовується для інформаційних потреб багатьох користувачів. В зв’язку з цим бази даних дозволяють в значній мірі скоротити надлишковість інформації. Перехід від структури БД до потрібної структури в програмі користувача відбувається автоматично за допомогою систем керування базами даних (СКБД) [9].

Щоб оперувати даними, які становлять базу, необхідна окрема програма — система керування базами даних.

Керівна програма, призначена для збереження, пошуку й обробки даних у базі, називається системою керування базами даних (СКБД)

Сучасні СКБД — це програмні додатки, які дозволяють виконувати різноманітні завдання. Всі існуючі системи задовольняють, як правило, таким вимогам:

– можливості маніпулювання даними (введення, вибір, вставка, відновлення, видалення тощо). Основні операції з даними виконуються під керуванням СКБД. Важливими показниками є продуктивність СКБД, витрати на збереження і використання даних, простота звертання до бази даних тощо;

– можливість пошуку і формування запитів. За допомогою запитів користувач має змогу швидко одержувати різну інформацію, що зберігається в базі даних;

– забезпечення цілісності (узгодженості) даних. Під час використання даних багатьма користувачами важливо забезпечити коректність операцій, щоб запобігти порушенню узгодженості даних. Порушення узгодженості даних може призвести до їх невідновної втрати;

– забезпечення захисту і конфіденційності. Крім захисту від некоректних дій користувачів, важливо забезпечити захист даних від несанкціонованого доступу і від апаратних збоїв. Проникнення в базу осіб, які не мають на це права, може спричинити руйнацію даних. Конфіденційність бази даних дозволяє визначати коло осіб, що мають доступ до інформації, і порядок доступу.

Сьогодні існує багато СКБД, що відрізняються архітектурою, внутрішньою мовою програмування, операційною системою, якою вони керуються, а також іншими характеристиками. Найпопулярнішими СКБД, що встановлюються в невеликих організаціях і орієнтовані на роботу з кінцевими користувачами, є MySQL, SQL Server, Access. До складніших систем належать

розподілені СКБД, що призначені для роботи з великими базами даних, розподіленими на кількох серверах: Oracle, Sybase, Informix.

1.2 Аналіз проблем сучасних СКБД

1.2.1 Аналіз популярних СКБД

На сьогодні кожна СКБД пропонує своєму користувачеві досить поширений на сьогодні рольовий доступ до даних у БД, деякі з них забезпечують високий рівень захисту цілісності інформації.

MySQL – одна з найпопулярніших СКБД з відкритим кодом сьогодення, яка пропонується користувачеві для створення динамічних веб-сторінок, є складовою багатьох популярних серверів. Але з точки зору захисту у цієї СКБД є низка недоліків. Перший з них – це недостатньо суворий набір вимог щодо паролів користувачів. На сьогодні MySQL не має належного захисту від прямого перебору паролів, вона не примушує користувача створити належний пароль для свого облікового запису. Ця СКБД пропонує користувачеві створити належні привілеї для кожного користувача чи групи користувачів і за належного адміністрування може бути вельми захищеною, але на сьогодні цього недостатньо для того, щоб сказати, що ця СКБД є гарним вибором для інформації, втрата якої спричинить суттєві збитки.

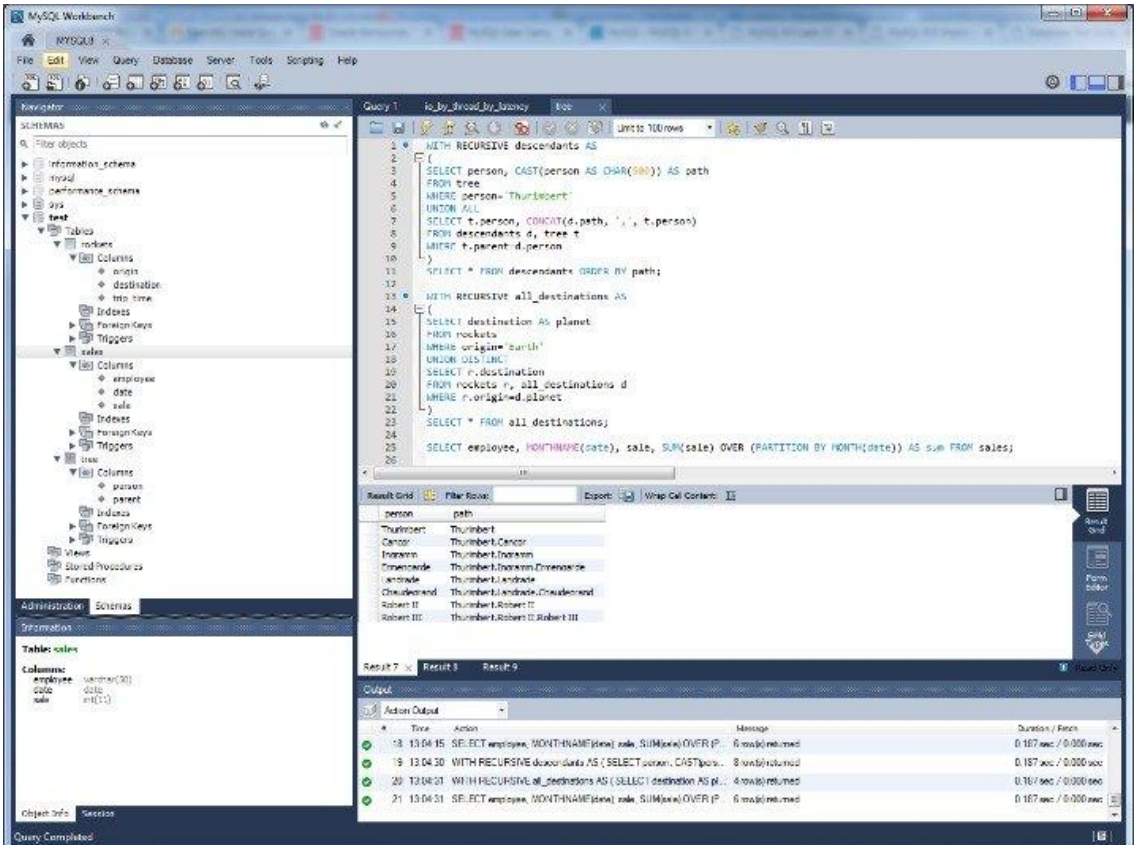


Рисунок 1.1 – Вигляд СКБД MySQL

Oracle – СКБД, розробники якої змогли попіклуватись про те, щоб інформація, яка буде зберігатись у базах даних, створених саме їхньою СКБД буде вважатись «краще захищеною, ніж у конкурентів», тому там вже можна побачити таке доповнення, як Oracle Advanced Security, яке дозволяє шифрувати увесь потік даних. Network encryption теж допоможе користувачеві захистити його дані у мережі. Розробники пропонують користувачеві такі алгоритми, як AES, DES, Triple-DES [10]. Також тут є аудит доступу до даних та жорсткіші вимоги до захисту облікових записів користувачів (захист від перебору паролю, вимоги до створення паролю облікового запису та інше). Але для того, щоб даний захист запрацював у пересічного користувача йому потрібен кваліфікований адміністратор СКБД, який зможе відлагодити рівень захисту інформації на належному рівні та знає про ці всі засоби захисту. Тому слід зауважити, що на сьогодні вся потужність усіх засобів захисту у даній СКБД не використовується на сьогодні через те, що потенційним користувачам простіше заплатити за розробку власної СКБД через те, що більшість замовників не

хочуть сподіватись на підтримку потрібних їм функцій захисту від розробника СКБД, яку вони купують. Так, як стартовий захист у СКБД є звичайним парольним разом з гарними засобами захисту Oracle не може запропонувати користувачеві гарантію захищеності даних без належного налаштування їхньої СКБД.

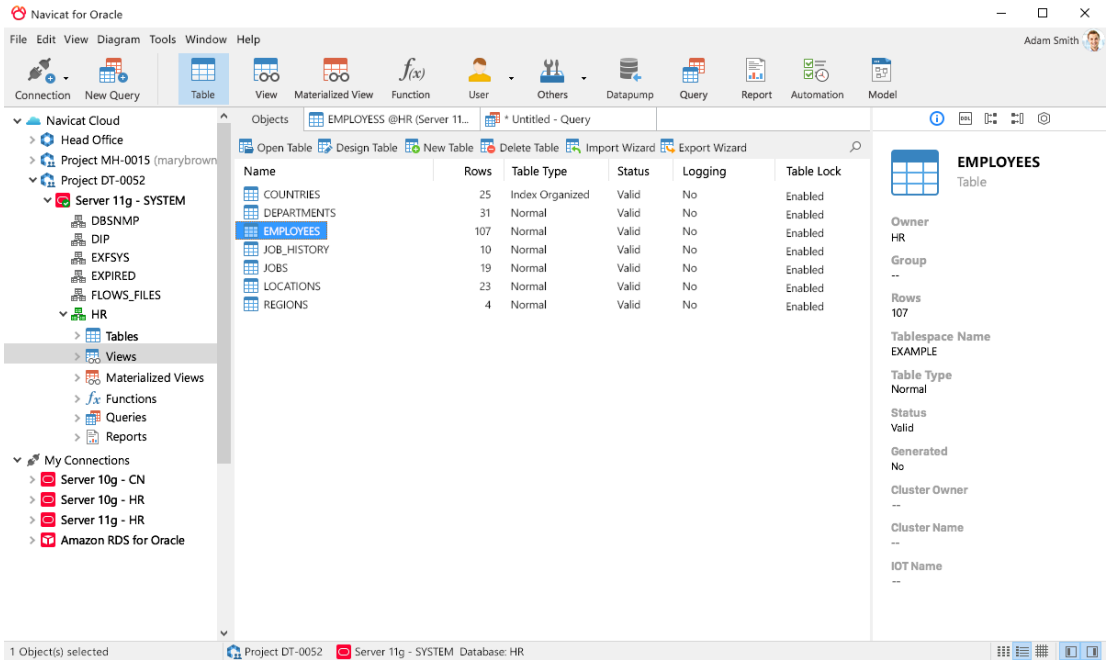


Рисунок 1.2 – Вигляд графічного інтерфейсу СКБД Navicat for Oracle

MS Access – доволі популярна на сьогодні і проста СКБД, яка дозволяє користувачеві на інтуїтивному рівні створити свою базу даних, і захистити її на основі створення паролю чи навіть робочої групи. Має низку налаштувань безпеки, які сильно програють конкурентам. Усі компоненти бази даних такі, як таблиці, звіти, запити, форми та інші об’єкти зберігає у єдиному файлі, що зберігається на диску і має розширення .mdb.

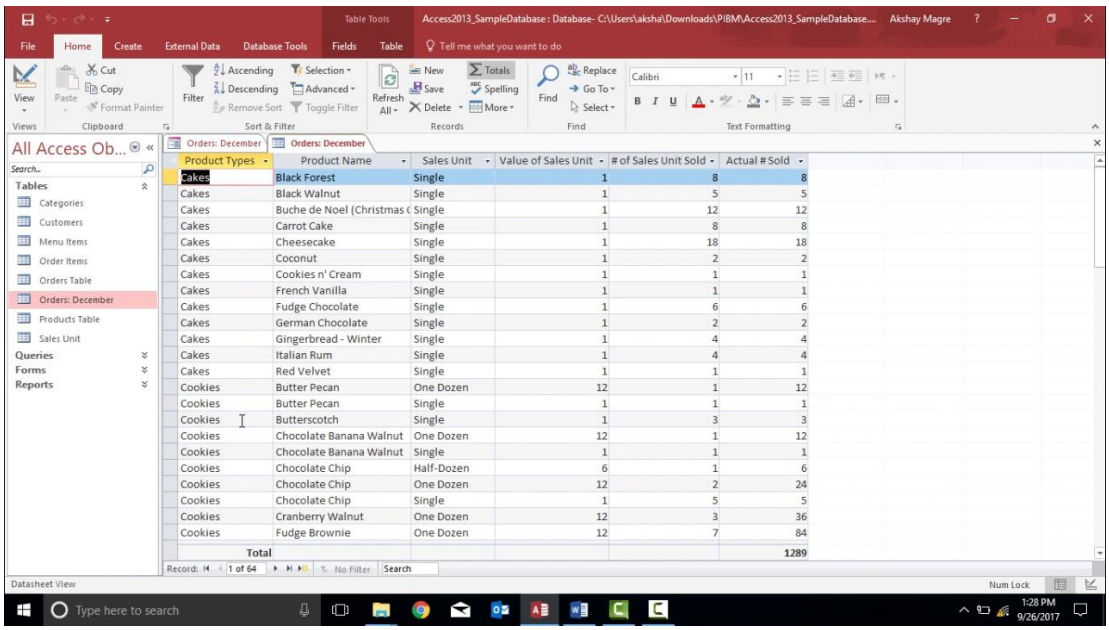


Рисунок 1.3 – Вигляд графічного інтерфейсу СКБД MS Access

Проаналізувавши сьогоднішній стан безпеки систем керування базами даних можна виділити тенденцію розробників з точки зору захисту: кожна СКБД має розмежування доступу до інформації в базах даних за допомогою створення користувачів, ролей, захисту цілісності шляхом створення резервних копій та інших функцій, які можуть бути доступні тільки користувачеві, який має освіченого адміністратора, но жодна з сучасних СКБД не може запропонувати захищеність інформації користувача відразу ж після створення.

1.2.2 Аналіз сучасних загроз безпеці базам даних під управлінням сучасних СКБД

У ході аналізу сучасних систем керування базами даних більшість з них мають в наявності свої проблеми [11], відображені у таблиці 1.1.

Таблиця 1.1 – Рейтинг загроз баз даних

Загроза	Поява (%)	Пояснення
Надмірні дозволи	50	Надання більшості користувачів привілеїв за замовчуванням, або надання ролей, які мають доступ до функціональних можливостей, які є непотрібними
Слабкі паролі	45	Слабкі паролі або паролі за замовчуванням

Продовження таблиці 1.1

Старі оновлення безпеки	35	Нестача оновлень безпеки, або використання старих версій програмного забезпечення
Слабо конфігурований аудит	35	Слабе налаштування відносно реєстрації користувацьких операцій чи аудиту у зв’язку з чим виникають труднощі з виявленням можливих втрат інформації чи зламів
Ім’я облікових записів за замовчуванням	30	Бази даних містять облікові записи за замовчуванням, що дозволяє використати атаки brute-force чи підбору паролю.
Надмірне збереження процедур	25	Бази даних містять надмірну кількість потенційно небезпечних процедур, включаючи ті, які можуть запускати системні команди або інші файли з файлової системи. Це призводить до збільшення доступних функцій, які можуть бути використаними для запуску атак на базу, операційну систиму та ін.
Дозволені порожні паролі	20	Подібно до проблем зі слабким паролем, проте такі бази немають додаткового рівня захисту і є слабозахищеними
Дублікати паролів	15	Декілька профілів бази даних мають схожий пароль, що компрометує ці профілі, та завдає безпеці бази даних чималої шкоди
Звичайний текст в основні назві аккаунту	5	Відсутність захисту для конфіденційної інформації ставить під загрозу збереження поточних користувацьких облікових записів

З табл. 1.1 можна зробити висновок, що сучасні системи керування базами даних у більшості випадків мають стійкий захист, проте частина загроз, яка зв’язана з користувачами і їх обліковими записами навіть у 2018 році постає дуже різко та потребує великої уваги як спеціалістів захисту інформації, адміністраторів баз даних, так і спеціалістів, що розробляли дане програмне забезпечення.

Виходячи з цього було прийнято рішення більшість подальшої роботи приділити увагу аналізу та покращенню стану захищеності баз даних шляхом покращення захисту в області користувацького доступу, використання нових криптографічних методів, розмежування функціонального доступу на підрівні та ін.

1.3 Аналіз моделей політики безпеки

1.3.1 Характеристика моделей політики безпеки

На сьогодні відомі три типи моделей ПБ, які досить детально досліджені та широко використовуються [12]:

- дискреційна;
- мандатна;
- рольова.

Перші дві досить давно відомі і досліджені, а рольова політика є новим досягненням теорії та практики захисту інформації.

Дискреційна політика безпеки має на основі дискреційне управління доступом, що визначається двома властивостями:

- усі суб’єкти і об’єкти повинні бути однозначно ідентифіковані;
- права суб’єкта на доступ до об’єкта системи визначаються на основі деякого зовнішнього правила, що визначається відносно системи яка базується на використанні атрибутів доступу.

Дана політика характеризується наявністю, яка фіксує множину кожного суб'єкта до доступних йому об'єктів та суб'єктів.

Перевагами дискреційної політики є:

- відносна простота в реалізації відповідних механізмів захисту;
- велика економія пам'яті, що зумовлена високою розрідженістю матриці доступів.

До мінусів дискреційної політики належать:

- слабкість до атак типу «Троянський кінь»;
- автоматичне визначення прав (важкість заздалегідь вручну визначити перелік прав кожного суб'єкту на доступ до об'єктів;
- проблема розповсюдження прав доступу;
- питання правил розповсюдження прав доступу і аналізу їх впливу на систему.

Мандатна політика безпеки складає мандатне управління доступом, що має на увазі, що:

- всі суб’єкти і об’єкти повинні бути однозначно ідентифіковані;
- заданий лінійно упорядкований набір міток секретності;
- кожному об’єкту системи привласнена мітка секретності, яка визначає цінність інформації, що міститься в ньому – його рівень секретності в автоматизовані системі;
- кожному суб’єкту системи привласнена мітка секретності, яка визначає рівень довіри до нього в автоматизовані системі – максимальне значення мітки секретності об’єктів, до яких суб’єкт має доступ; мітка секретності суб’єкта називається його рівнем доступу.

Перевагами мандатної політики безпеки є:

- більш високий ступінь надійності, який пов’язаний з тим, що за правилами мандатної політики безпеки відстежуються не тільки правила доступу суб’єктів системи до об’єктів, але і стан самої автоматизованої системи;
- правила мандатної політики безпеки більш ясні і прості для розуміння розробниками і користувачами автоматизованої системи;
- мандатна політика безпеки стійка до атак типу «Троянський кінь».

- мандатна політика безпеки допускає можливість точного математичного доказу, що дана система в заданих умовах підтримує ПБ.

Однак мандатна політика безпеки має серйозні вади вона складна для практичної реалізації і вимагає значних ресурсів обчислювальної системи. Це пов’язано з тим, що інформаційних потоків в системі величезна кількість і їх не завжди можна ідентифікувати. Саме ці вади часто заважають її практичному використанню.

Рольова модель політики безпеки є цілком новим типом політики, яка базується на компромісі між гнучкістю керування доступом, яка є характерною для дискреційною політикою безпеки, і жорсткістю правил контролю доступу, яка притаманна мандатною політикою безпеки.

Рольова політика безпеки розповсюджена досить широко, тому що вона, на відміну від інших більш строгих і формальних політик, є дуже близькою до реального життя. Дійсно, на самій справі, користувачі, що працюють в системі, діють не від свого особистого імені – вони завжди здійснюють певні службові обов’язки, тобто виконують деякі ролі, які аж ніяк не пов’язані з їх особистістю.

Завдяки гнучкості та широким можливостям РПБ суттєво перевершує інші політики, хоча іноді її певні властивості можуть виявитися негативними. Так вона практично не гарантує безпеку за допомогою формального доказу, а тільки визначає характер обмежень, виконання яких і є критерієм безпеки системи.

Перевагами рольової політики безпеки є:

- підхід дозволяє отримати прості і зрозумілі правила контролю доступу
- оперування ролями набагато зручніше за оперування об'єктами
- модель поведінки відповідає розповсюдженим технологіям обробки інформації, які передбачають розподіл обов’язків і сфер відповідальності між користувачами.

Вадою рольової політики безпеки є система позбавлена теоретичної доказової інформації про компрометацію певних користувачів.

У ході виконаного порівняння було прийнято рішення використовувати найсучаснішу та найоптимальнішу у даному випадку модель, а саме – рольову модель політики безпеки, яка дозволить легко масштабувати систему управління доступом і навіть поєднуватись одночасно з іншими ПБ, що не є необхідністю, проте є одним з додаткових плюсів даного вибору.

1.3.2 Аналіз рольової моделі розмежування доступу

Сучасні СКБД можуть бути охарактеризовані, як програмні засоби з високим ступенем захищеності інформації, яка зберігається в базах даних під їх управлінням, проте під час їх аналізу було виявлено один з недоліків [13], а саме використання одного бар'єру (дискреційної, рольової, моделі доступу).

При аналізі проблеми було виявлено, що на сьогодні СКБД надають користувачам певні ролі, які мають свій набір привілеїв [14], основні з них зображені у таблиці 1.2 [15].

Таблиця 1.2 - Ролі користувачів у сучасних СКБД

Роль	**Можливості**	**Загрози**
Власник	Усі дії по налаштуванню та обслуговуванню БД та видалення	Втрата даних у зв'язку з некомпетентністю чи халатністю,
Адміністратор	Адміністрування бази даних, надання привілеїв.	Цілісність даних, ненавмисне надання прав іншим користувачам.
Редактор	Редагування та видалення даних у таблицях	Цілісність та конфіденційність даних
Читач	Зчитування даних	Конфіденційність даних
Користувач без прав	Не може виконувати дії з БД	-

Для аналізу та висування пропозицій слід перелічити деякі з реалізацій загроз, наведених у таблиці 1.2.

Загрози отримання зловмисником невідповідних йому автентифікаційних даних:

- читач бази даних дасть змогу зловмиснику вкрасти інформацію;
- редактор дасть змогу відредагувати інформацію у БД;
- адміністратор дасть змогу приховано надати права користувачам, що не мають відповідного рангу;
- власник ставить під загрозу існування бази даних в цілому.

Проаналізувавши загрози (табл. 1.2) було виявлено ті, що можуть бути реалізовані відповідними користувачами. Сучасні СКБД надають права щодо користування базою даних після автентифікації користувача з необхідним рівнем доступу, тобто зловмиснику стає доступним рівень доступу користувача, в якого він міг отримати автентифікаційні дані. Даних підхід, як і будь-яка дискреційна (рольова) модель доступу, відкриває багато можливостей для несанкціонованого доступу до даних через перехоплення автентифікаційних даних та зловживання повноваженнями. З цього можна зробити висновок, що однорівневий захист є проблемою, яку необхідно вирішити.

В реальних умовах при роботі з СКБД створюється достатня кількість проблем з точки зору безпеки, які зв’язані з користувацькою авторизацією. Це можуть бути, як проблеми звичайного підглядання паролю з боку працівників, що сидять поруч, так і проблеми крадіжок необхідних даних шляхом використання власних користувацьких автентифікаційних даних з невідповідною високою користувацькою роллю. З цього можна зробити висновок, що сучасні СКБД, які мають слабкий парольний захист, чи потребують більшого рівня захисту [16], потребують покращення системи авторизації.

Після проведення досліджень можна зробити висновок, що досліджені СКБД мають низку переваг у вигляді можливості дуже гнучко налаштовувати власний функціонал. Проте з аналізу можна виділити основні недоліки сучасних СКБД – слабкий захист від втрати користувацького доступу та

використання слабких моделей розмежування користувацького доступу до функції з управління.

Відповідно до переваг відомих СКБД запропоновано використовувати криптографічні функції, зокрема гешування [17] та шифрування [18], використовуючи сучасні та стійкі криптоалгоритми, проте на прикладі поточної розробки вони будуть використовуватись дещо іншим способом. Отже до процесу отримання та аналізу інформації про сучасні СКБД було виділено два модулі, які необхідно реалізувати: гешування та шифрування.

Згідно з виконаною роботою над аналізом моделей розмежування доступу та головних сучасних проблем захищеності баз даних було прийнято рішення розробити та реалізувати у вигляді моделі, алгоритм захисту функцій бази даних шляхом розмежування користувацького доступу.

1.4 Постановка задачі

Оскільки завданням дипломної роботи є розробка системи, яка продемонструє підвищення захищеності шляхом поєднання різноманітних методів захисту інформації, таких як криптографія, удосконалення моделі доступу до даних, створення нової моделі розшарування користувацького доступу до інформаційних та функціональних шарів СКБД, можна виділити такі задачі для виконання:

- дослідити існуючі аналоги СКБД та систем з великою кількість функцій та політик безпеки;
- проаналізувати існуючі засоби захисту інформації у СКБД;
- порівняти можливості мов програмування для програмної реалізації моделі доступу у вигляді програмного застосунку;
- розробити інтерфейс системи;
- розробити модель та алгоритми розмежування доступу користувачів до функцій та методів у СКБД;
- провести тестування готового програмного засобу.

1.5 Висновки

Для висвітлення та виокремлення головних проблем сучасних систем керування базами даних було проведено аналіз сучасних аналогів. З проведених аналізів можна зробити висновок, що основною проблемою сучасних СКБД є людський фактор – проблема, яка зв’язана з користувачами СКБД та інших сучасних систем, які потребують наявність політики безпеки та розмежування користувацького доступу.

Було прийнято рішення розробити систему з спроектованим розмежуванням користувацького доступу на «шари», з відповідними методами захисту, які проектуються та налаштовуються для кожного шару окремо.

2 ПРОЕКТУВАННЯ БАГАТОШАРОВОГО ЗАХИСТУ

2.1 Розробка моделі багатошарового доступу до функцій СКБД

Для покращення захисту даних у СКБД запропоновано розшарування доступу до різних функціональних рівнів СКБД шляхом ускладнення доступу користувачів до функцій СКБД залежно від привілеїв, що їм надаються (рис. 2.1).

Для забезпечення належного захисту інформації необхідно комбінувати найкращі існуючі напрацювання та розбивати їх використання на «рівні захисту». Це дасть деякі переваги: користувач, який не володіє необхідним набором автентифікаційних даних отримає доступ тільки до відповідного рівня взаємодії з базою даних.

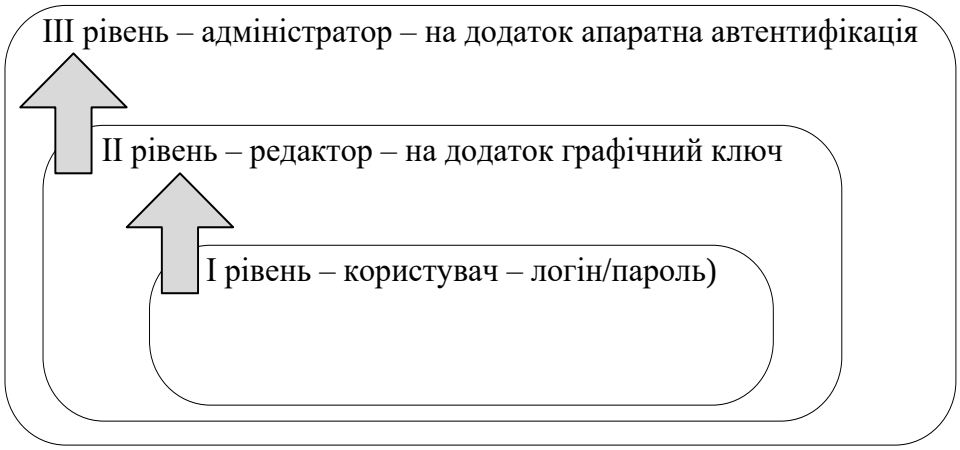


Рисунок 2.1 – Рівні користувацького доступу до функцій СКБД відповідно до запропонованої моделі захисту

Для того щоб реалізувати відповідні функції шарів отримання доступу до функцій СКБД передбачено реалізацію геш-функцій та основних криптографічних алгоритмів, які будуть використовуватись для гешування атрибутів та шифрування інформації, проте ці функції будуть використовуватись на різних «шарах» проходження користувацької автентифікації.

Перший рівень автентифікації користувача передбачає використання зв’язки логіну та паролю, які використовуються для підтвердження наявності

зв’язки введеного ним логіну і паролю у базі користувачів, які мають право на читання вмісту таблиць, що зберігаються в базі даних.

Перший рівень автентифікації використовується для встановлення з’єднання з сервером, отримання ключів доступу JWT та створення нової сесії для користувача (рис. 2.2).

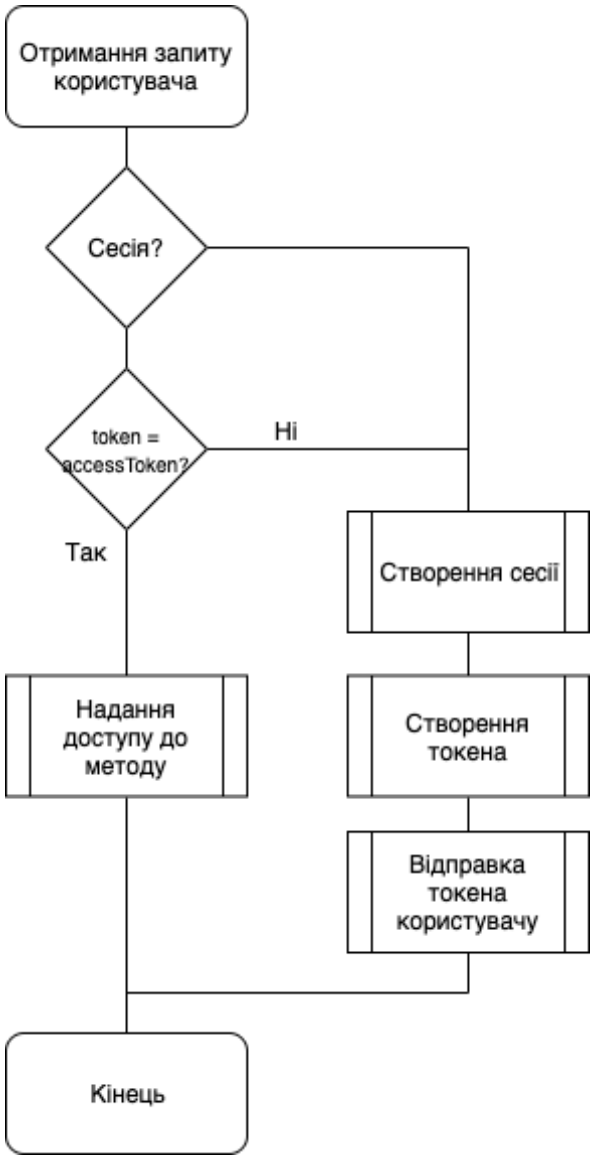


Рисунок 2.2 – Алгоритм створення сесії та перевірки токена

Отримавши перший рівень користувач, який хоче відредагувати інформацію має підтвердити свої права редактора для отримання доступу до функцій, які знаходяться на другому рівні доступу.

Прикладом реалізації автентифікації при отриманні прав другого рівня (функціонального рівня редактора) може бути сутність, яка використовує

алгоритми гешування і зберігає геш-значення з таблицею, до якої хоче отримати доступ, особа з правами користувача. Для переходу на другий рівень користувачеві пропонується ввести графічний пароль, та вибрати геш-функцію, яка повинна до нього примінитись. Після введення, автентифікаційні дані зіставляються з даними, які прив’язані до користувача, і після підтвердження надається доступ.

Другий рівень має в собі алгоритм перевірки доступу користувача до виконуваних ним методів та функцій (рис. 2.3):

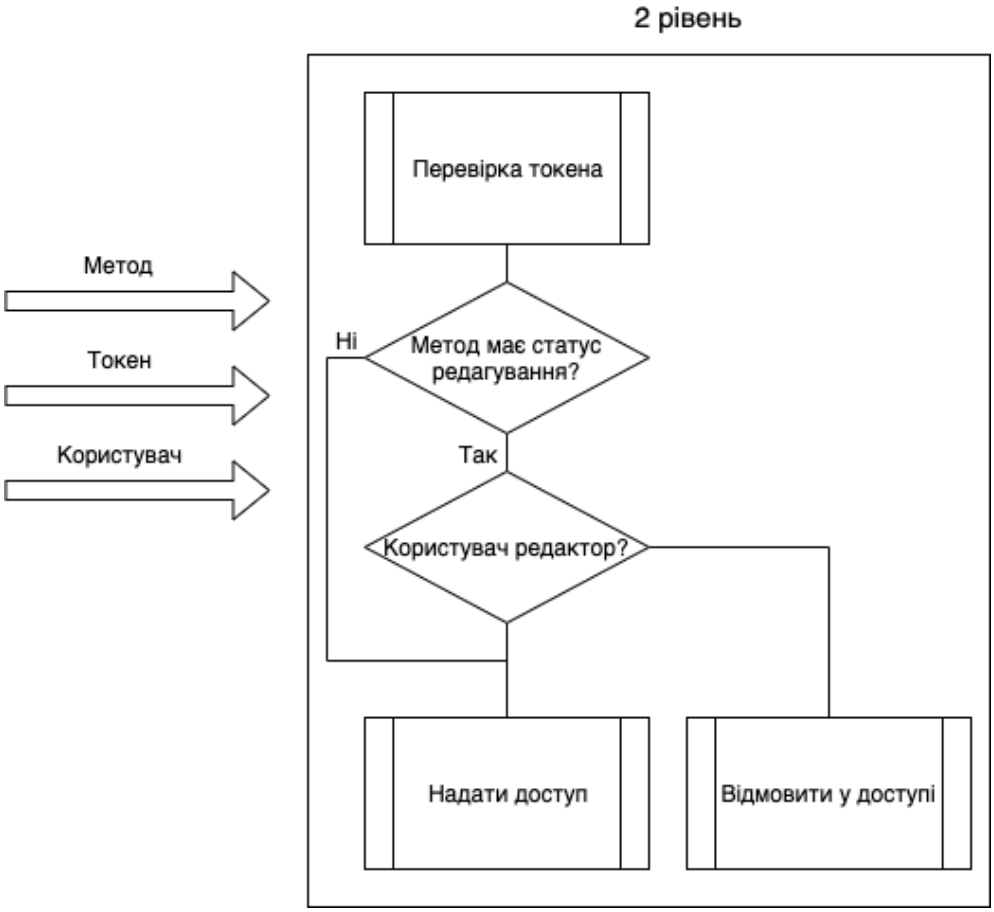


Рисунок 2.3 – Алгоритм отримання та перевірки доступу користувача до функцій редагування

Третій рівень доступу реалізується підтвердженням доступу з другого рівня (рівня редактора), наприклад шляхом наявності флешки-ключа. При запиті на отримання найвищих прав користувачеві виводиться повідомлення, яке пропонує йому вставити флешку з файлом, який містить необхідне для підтвердження прав геш-значення, яке було згенероване під час створення БД і

відвантажене на флешку. При наявності необхідного файлу його дані скануються та зіставляються з наявними даними у БД. Також можлива реалізація з використанням технології одноразових пароль, що генеруються на основі отриманого геш-значення.

Третій рівень містить у собі такі функціональні блоки:

1. Завантаження захищеного файлу-ключа на сервер.
2. Зчитування вмісту файлу-ключа на сервері.
3. Співставлення вмісту файлу-ключа з існуючим на сервері текстовим ключем адміністратора бази даних.
4. При правильній відповідності вмісту файлу ключа – користувачеві на клієнтську частину програмного засобу надходить повідомлення про надання повного адміністраторського доступу до усіх методів та функцій по користуванню СКБД.

Узагальнена схема отримання доступу відповідно до рівнів захисту показана на рис. 2.2.

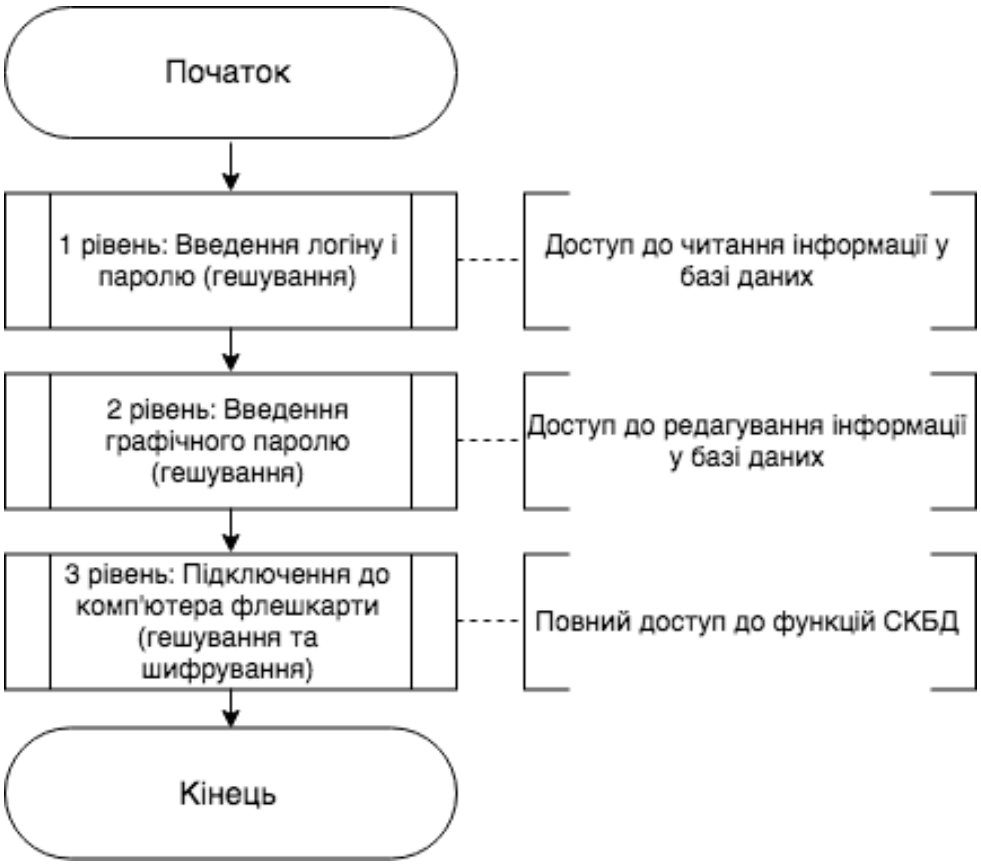


Рисунок 2.2 – Вигляд рівнів захисту з криптографічнми функціями, які на них використовуються

У зв’язку з цим час, потрачений на процес отримання необхідних користувачеві привілеїв збільшиться, але дозволить відслідковувати та керувати доступом користувачів і захищати цілісність та конфіденційність бази даних. Крім того, доступ з правами третього рівня потрібен тільки у виключних ситуаціях, і затримка не сильно впливає на нормальну роботу з базою даних.

Такі три шари захисту дозволяють обмежити можливості зловмиснику, який хоче, наприклад, видалити БД. Зламавши пароль він зупиниться на шарі захисту, який пропонує вибір геш-функції та введення графічного паролю, до якого вона буде застосовуватись. Після цього, зловмиснику необхідно подолати наступний шар – наявність носія з електронним-ключем, і наприкінці співставлення отриманих автентифікаційних даних зі списком ролей та відповідним йому списком користувачів. Таким чином, зловмисник, який отримав автентифікаційні дані одного користувача, графічний пароль та геш-функцію другого користувача, а носій в третього користувача, не зможе отримати доступ.

Отже розмежування доступу користувача має на меті поділення функціоналу на «шари», які будуть доступні тільки конкретним рівням доступу, які стають доступними тільки через низку користувацьких автентифікацій. Таким чином щоб отримати повний, загальний контроль над базою даних користувачеві потрібно відповідно доказати, що він має усі належні права (рис. 2.3).

Рисунок 2.3 – Вигляд шарів функцій баз даних

В доповнення до багатошарової моделі доступу до користувацької інформації у базі даних, використано порівняно нову технологію, а саме blockchain[19]. Дана технологія дозволить базі даних мати додатковий рівень захисту інформації щодо забезпечення цілісності інформації, яка у ній зберігається та структури бази даних в цілому. Дана технологія дозволить зв’язати усю важливу інформацію в один ланцюг даних, які залежатимуть один від одного, і, при неправомірній зміні інформації зловмисником, відразу ж буде відображення помилки та мутації при використанні БД іншими користувачами, і, цим самим, дозволить користувачам розпізнати можливі ситуації, коли базу було змінено без відома інших користувачів.

2.2 Розробка відслідковування несанкціонованих дій

2.2.1 Визначення технології Blockchain

Для перевірки змін, внесених в БД було використано алгоритм перевірки цілісності інформації, який також використовується в технології blockchain. Blockchain — децентралізована система зберігання даних або цифровий реєстр транзакцій, угод, контрактів чи будь якої іншої інформації, що складається з набору записів. Blockchain є ланцюгом блоків даних рис, 3, які створюються і зберігаються на комп'ютерах учасників ланцюжка. Всі учасники мережі

діляться на дві категорії: звичайні користувачі, які створюють нові записи, і «майнери», які створюють блоки. Майнери перевіряють записи, які створюють звичайні користувачі, формують з них блоки, а потім розсилають ці блоки по мережі. Звичайні користувачі отримують ці блоки і зберігають їх у себе в комп'ютері. Учасники Blockchain-мережі мають доступ до інших комп'ютерів мережі, завдяки чому можна обмінюватися даними. Кожен користувач перевіряє коректність нових даних. Якщо вони достовірні, він зберігає їх і передає далі по мережі [21].

Блоки складаються з такої інформації:

- previousHash – геш-значення попереднього блоку для перевірки «валідності» чейну;
- hash – геш-зачення блоку, яке формується на основі усіх інших значень;
- data – дані, які зберігатимуться у блокові.

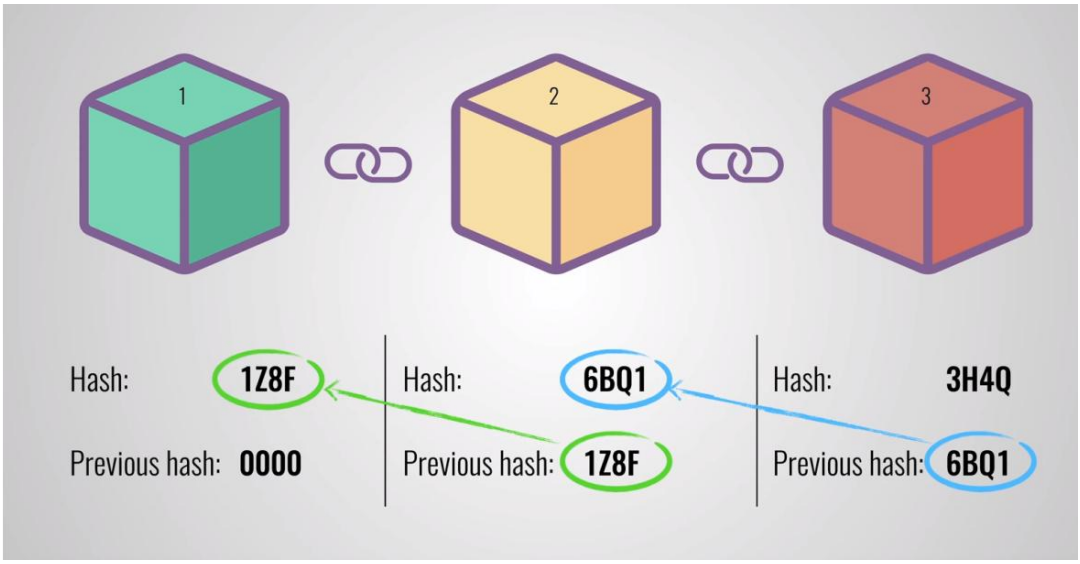


Рисунок 2.3 – Вигляд блоків, що складають ланцюги у технології blockchain

Основні принципи технології «блокчейну»:

- захищеність — через певний час на такому аркуші закінчується місце для записів. Тоді його «опечатують» унікальним зашифрованим кодом, погодженим усіма учасниками. Відтак записи на такому аркуші можна лише передивлятися, але не змінювати.

- теоретична необмеженість – теоретично блокчейн можливо доповнювати записами до нескінченності. Тому його часто порівнюють з суперкомп'ютером.

- гарантія — оскільки кожна наступна сторінка-блок залежить від попередньої, якщо хтось захоче змінити дані вже «опечатаної» сторінки, йому також доведеться змінити зміст і шифр усіх наступних сторінок. А здійснити це самотужки, ураховуючи зростаючу складність обчислення кожного наступного коду, просто неможливо.

2.2.2 Використання технології для досягнення поставлених задач

Дану концепцію можна використати для перевірки цілісності та контролю за можливими несанкціонованим доступом до інформації.

У реляційних базах даних інформація зберігається у таблицях, до яких можна добавити додатковий шар, який відповідатиме за гешування та зв’язок геш-значень полів. Приклад наведено у таблиці 2.1.

Таблиця 2.1 – Приклад таблиці з зв’язками геш-значень полів

№ п.п.	Ключ 1	Ключ 2	Ключ 3	Ключ 4	Ключ 5
1	Значення 1	Значення 2	Значення 3	Значення 4	Значення 5
Геш-зн.	w7dlR	Pjysb	SPpz0	FGxDO	xVb20
Попереднє геш-зн.	0	w7dlR	Pjysb	SPpz0	FGxDO

При змінах авторизованим користувачем з відповідним доступом алгоритм запускає обчислення значень для кожної з комірок, тим самим приводячи кожну комірку в де-факто перевірений стан. Усі комірки, що відповідають за попереднє геш-значення будуть відноситись до відповідних їх попередніх стовпців того ж рядка. При кожному запуску таблиці запускається алгоритм, який перевіряє посилання на комірки та перевіряє чи всі геш-значення відповідають дійсності.

У разі несанкціонованого доступу до БД алгоритм перерахунку геш-значень не буде автоматично запущено і при наступному запуску алгоритм перевірки посилань на комірки (перевірки відповідності геш-значень) видасть помилку через те, що одне з змінених геш-значень не відповідає посиланню з наступної комірки. Приклад таблиці з зміненим значенням у 3 колонці наведено у таблиці 2.2.

Таблиця 2.2 – Приклад таблиці з непідтвердженими «зламаними» зв’язками полів

№ п.п.	Ключ 1	Ключ 2	Ключ 3	Ключ 4	Ключ 5
1	Значення 1	Значення 2	Текст 3	Значення 4	Значення 5
Геш-зн.	w7dlR	Pjysb	***PUGfh***	FGxDO	xVb20
Попереднє геш-зн.	0	w7dlR	Pjysb	SPpz0	FGxDO

У зв’язку з тим, що дані були несанкціоновано змінені у стовпчику 3, а саме «Значення 3» змінено на «Текст 3», при запуску відображення таблиці запускається алгоритм, який перераховує значення для усіх комірок і посилання з стовпчика 4 не відповідає геш-значенню з стовпчика 3, відповідно до цього програмний застосунок робить висновок про несанкціоновані зміни і повідомляє про це користувачеві.

2.3 Використання JSON Web Tokens при користувацькій автентифікації

JWT (JSON Web Token) – відкритий стандарт для створення токенів доступу, оснований на JSON форматі. Як правило використовується для передачі даних авторизації користувачів в клієнт-серверних додатках. Токени створюються сервером та відправляються на клієнтську частину.

JWT-токен у текстовому вигляді складається з трьох частин [20], розділених між собою крапкою, а саме: заголовка (header), інформаційної частини (payload) та підпису (signature). Тобто, маркер має вигляд “header.payload.signature”. Кожна частина маркера кодується за допомогою

методу кодування Base64. Сформований JWT-маркер наведено на такому прикладі:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikhhbm5hIE9saWlueWsiLCJjb21wYW55IjoiTXkgQ29tcGFueSIsImF1dGgiOjQsImV4cCI6MTQ5Mjg4MzM0MX0.KRe-HIQFaQzwn1U-5e_tJkor9F-v-WHM3bG2ZPjgC9A

Зазвичай при використання JSON токенів [21] в клієнт-серверних додатках реалізована слідуюча схема:

1) Клієнт проходить авторизацію в додатку (до прикладу з використанням логіну та паролю).

2) У випадку успішної авторизації сервер створює на основі автентифікаційних даних клієнта токени, та відправляє їх клієнтові.

3) При подальшому зверенню до серверу, клієнт використовує токен, у відповідь.

Алгоритм роботи представлено на рис. 2.5:

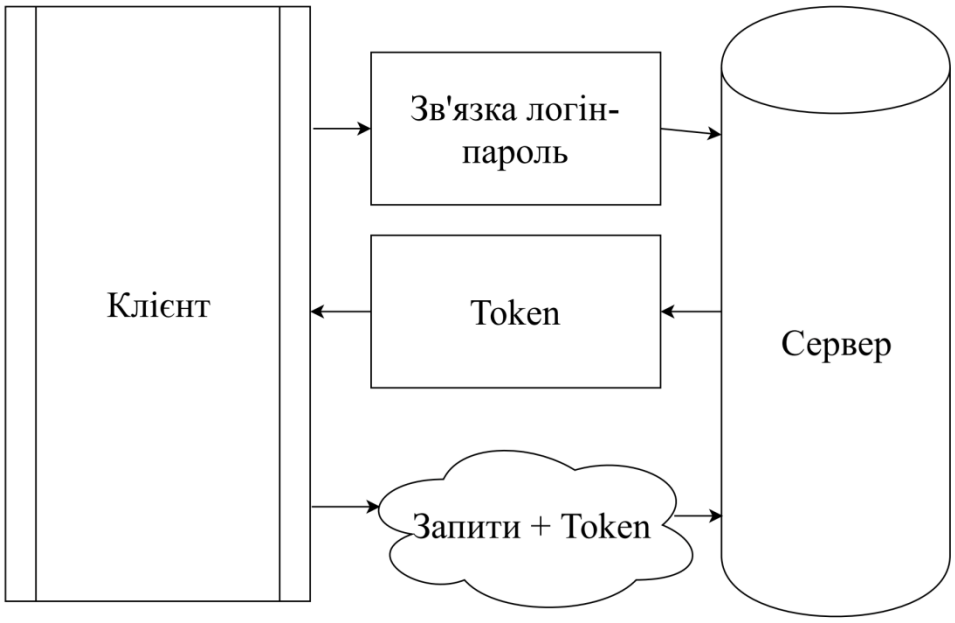


Рисунок 2.5 – Вигляд принципу роботи алгоритму JWT

Відповідно до усіх загальноприйнятих стандартів JWT також має алгоритм таймаутів, які дозволять регламентувати час доступу користувача до певних функцій.

Згідно з проаналізованими методами захисту та аналізом існуючих алгоритмів та методів захисту інформації у сучасних СКБД, побудова системи з багатошаровим користувацьким доступом є актуальним і необхідним рішенням для покращення сучасного захисту баз даних.

2.4 Висновки

Була розроблена модель розмежування користувацького доступу до даних та функцій системи керування базами даних, що використовує сучасні криптографічні методи та нові технології для автентифікації користувача, яка на відміну від сучасних систем керування базами даних розмежовує користувацьку відповідальність за окремі частини функціоналу та доступ до інформації (адміністрування, редагування та ін.) та захищає функціонал вищих рівнів у випадках зламу рівнів, що знаходяться нижче, а також дозволяє впроваджувати багаторівневе розмежування доступу до функцій по керуванню інформаційними системами у більшості сучасних систем, які мають примітивні користувацькі рівні доступу типу: «користувач-редактор-адміністратор».

3 ПРОГРАМНА РЕАЛІЗАЦІЯ МЕТОДІВ ТА ЗАСОБІВ ЗАХИСТУ

3.1 Обґрунтування вибору програмних засобів

Реалізація поставленої задачі, а саме – розробка програмного засобу, який буде складатись з модулів, які будуть основою системи керування бази даних з модульною системою захисту та багатошаровим автентифікаційним доступом, який реалізовуватиметься розмежуванням доступу користувачів до функцій СКБД буде реалізовуватись за допомогою мови програмування JavaScript.

Мова JavaScript – одна з найпопулярніших мов програмування на сьогодні[23] і має найбільшу бібліотеку налаштувань та фреймворків [24], які дають змогу реалізувати програмний засіб максимально зручним, швидким та захищеним.

Однією з можливостей даної мови є гнучке використання різноманітних користувацьких бібліотек, що допоможуть налаштувати та швидко побудувати необхідний програмний засіб. При розробці даного програмного засобу будуть використовуватись такі фреймворки:

- VueJS – JavaScript-фреймворк з відкритим кодом, який використовується для створення користувацьких інтерфейсів в парадигмі реактивного програмування [25]. Він легко інтегрується в JavaScript проекти з іншими бібліотеками завдяки поступово збільшуючісь екосистемі. Даний фреймворк може використовуватись, як допомогу у побудові логіки програмного засобу і для цього він і був обраний. Компонентна основа фреймворку дозволить будувати частини системи захисту модульно та ізольовано один від одного передаючи між ними тільки зашифровану інформацію.

- Програмна платформа NodeJS – платформа, основана на V8 і використовується для розробки високопродуктивних мережевих застосунків. JavaScript використовувався для обробки даних на сторонні користувача, але node.js дозволив обробляти дані на сервері[26]. До того ж ця платформа перетворила JavaScript, на мову загального використання з великою спільнотою розробників. Сервер, написаний на ці платформі буде використовуватись для

обробки великих обсягів інформації, атентифікації користувача та забезпечення правильного відпрацьовування усіх функцій роботи з БД.

- NestJS – фреймворк для роботи з NodeJS, який має в собі зв'язку NodeJS, Express, TypeScript[27], може працювати з WebSockets та JWT, яка необхідна для розробки автентифікаційного рівня . Він використовує прогресивний JavaScript і повністю підтримує TypeScript (але все ще дозволяє розробникам працювати з чистим JavaScript) і поєднує елементи об'єктно-орієнтованого програмування, функціонального програмування та FRP функціонального реактивного програмування.

- TypeORM – технологія програмування, яка дає можливість зв’язати бази даних з концепцією ООП у сучасних мовах програмування, та дає можливість працювати з використовуваним у проекті TypeScript’ом.

- VuetifyJS – компонентна бібліотека, розроблена для фреймворку VueJS та призначена для побудови зручних та зрозумілих користувацьких інтерфейсів[28]. Дана бібліотека зменшить затрати на побудову усього інтерфейсу програми.

- Bootstrap – CSS-бібліотека, що використовуватиметься як допоміжна бібліотека для VuetifyJS.

- PassportJS – бібліотека для роботи з Json Web Token (JWT). Він використовуватиметься для рівня автентифікації користувачів та дозволяє більш комплексно використовувати найсучасніші стратегії автентифікації (використання логіна та пароля, токенів, використання авторизації через сучасні соц. мережі, тощо).

При використанні усіх переваг мови програмування JavaScript та набору вибраних фреймворків побудова захищеного програмного засобу буде проходити на оптимальні швидкості та допоможе побудувати масштабовану компонентну та швидку у розробці кодову базу програми, яку можна буде використати у подальших дослідженнях та покращеннях стану сучасного захисту баз даних.

3.2 Формалізація задач модулів захисту

Задачею реалізації захищеного програмного засобу на даний момент є реалізація модулів, які і будуть основним захистом для СКБД:

- парольна автентифікація користувачів – вікно, яке при запуску додатку буде вимагати від користувача введення своїх автентифікаційних даних;
- компонент визначення користувацької ролі і визначення слідуючих модулів, які відповідатимуть певним рівням (модуль керування авторизацією);
- вікно вибору криптографічних функцій, яке викликатиметься модулем керування та передаватиме інформацію на модуль криптографічних функцій;
- модуль криптографічних функцій, який буде доступним з будь-якого місця програмного застосунку і викликатиметься за необхідністю більшістю «робочими» елементами та фукнціями;
- модуль зв’зку з сервером NodeJS (networking), який дозволить обробляти великі обсяги інформації;
- модуль роботи з файлами бази даних;
- модуль реалізованої технології Blockchain;
- модуль керування рольовим доступом та конфігурацією рівнів автентифікації.

Вигляд зв’язку між модулями програмного засобу зображено на рис. 3.1.

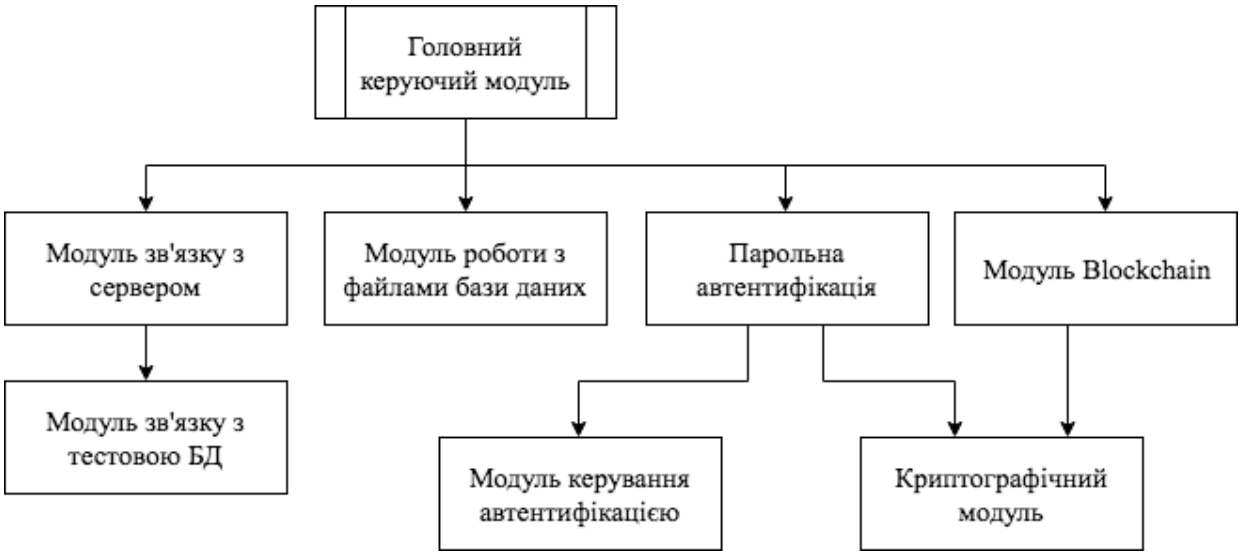


Рисунок 3.1 – Схема модулів програмного засобу

З схеми зв’язку модулів виходить, що застосунок після завершення матиме у собі як необхідний функціонал, так і можливість його розширення.

Велика частина захисту міститься у головному вікні по керуванню самою базою даних. Серед них:

- технологія blockchain;
- гешування атрибутів таблиць;
- шифрування даних у таблиці;
- модуль розподілення доступу до функцій керування базою даних.

Технологія blockchain буде використовуватись задля збереження цілісності атрибутів таблиць у базі даних. Механізм працюватиме на основі ланцюга, який формуватиметься з атрибутів таблиць та унеможливить їх мутацію.

Гешування атрибутів буде частиною технології blockchain, та відповідно до цього поле «data» у blockchain’і буде теж містити геш-значення, сформоване з порядкового номеру та назви атрибуту.

Шифрування даних у таблиці є теж необхідним методом захисту для випадків, коли зловмисник зміг добратись безпосередньо до файлів бази даних і унеможливить доступ зловмисника до інформації, збереженої у базі даних.

Модуль розподілення доступу відповідатиме за активацію або блокування функцій по управлінню базою даних, він буде вимикати та керувати помилками, які показуватимуться у користувацькому вікні.

Для прикладу на рис 3.2 було відображено функціональне схематичне зображення методів, що використовується при формуванні запису, який буде міститись у базі даних.

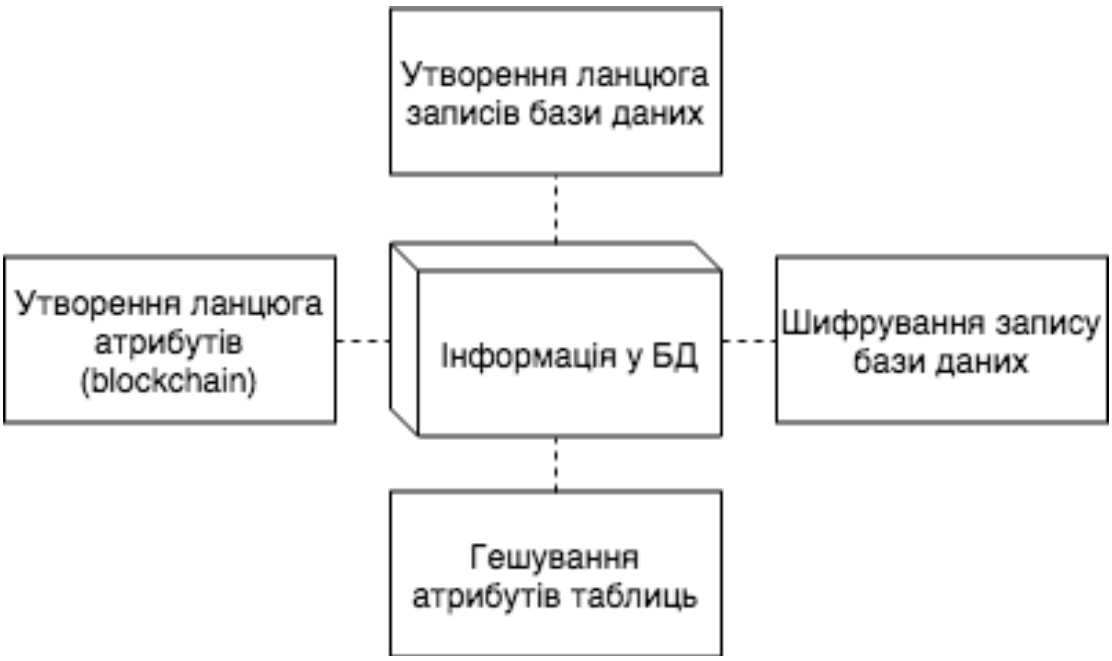


Рисунок 3.2 – Схема методів, що використовуються при формуванні таблиці

3.3 Програмна реалізація І-го кроку автентифікації

Першим кроком користувацької автентифікацію за-замовчуванням є автентифікація користувача за допомогою зв’язки «логін + пароль». Даний крок є обов’язковим кроком автентифікації користувачів у всіх сучасних СКБД і дозволяє системі вірно визначити подальші кроки користувацької автентифікації, тобто визначити набір правил, якими система буде ставити вимоги до користувача.

Оскільки основною метою даної дипломної кваліфікаційної роботи є забезпечення нового рівня захисту при роботі з сучасними базами даних JWT, як один з безпечних способів передачі інформації між двома учасниками (наразі використовується клієнт-серверна модель зв’язку), є беззаперечно найкращим рішенням у сучасних додатках. Для його створення необхідно визначити заголовок (header) з загальною інформацією по токену, корисні дані (payload), такі як id користувача, його роль та ін. і підписи (signature).

Для цього при проходженні рівня авторизації на стороні користувача:

```
signInWithLogPass({ dispatch, commit }, userCredentials) {
  api
    .post(`/auth/signIn/`, {
      login: userCredentials.login,
      password: userCredentials.password
    })
    .then(response => {
      commit(types.SET_CURRENT_USER, response.user);
      commit(types.SET_CURRENT_USER_LOGINED, true);
      api.addHeader("Authorization", "Bearer " + response.accessToken);
      dispatch("getSchema");
    })
    .catch(error => {
      commit(types.SET_CURRENT_USER_LOGINED, false);
      console.error(error);
    });
},
```

Рисунок 3.3 – Вигляд методу авторизації за допомогою логіну та паролю

Рядок:

```
api.addHeader("Authorization", "Bearer " + response.accessToken);
```

Він слугує для збереження на стороні користувача токена, який дозволить в подальшому виконувати функції читання з бази даних.

З іншого боку на стороні сервера використовується перевірка токена:

```
async validate(payload: IJWTAccessPayload, done: (error: Error | null, user: IUser | null) => void) {
    try {
        done(null, await this.sessionsService.validateUser(payload));
    } catch (error) {
        done(error, null);
    }
}
```

Рисунок 3.4 – Вигляд методу авторизації на стороні серверу

З точки зору розробки інтерфейсу користувача даний крок виглядає як звичайна форма введення користувацького логіну та паролю (рис. 3.4).

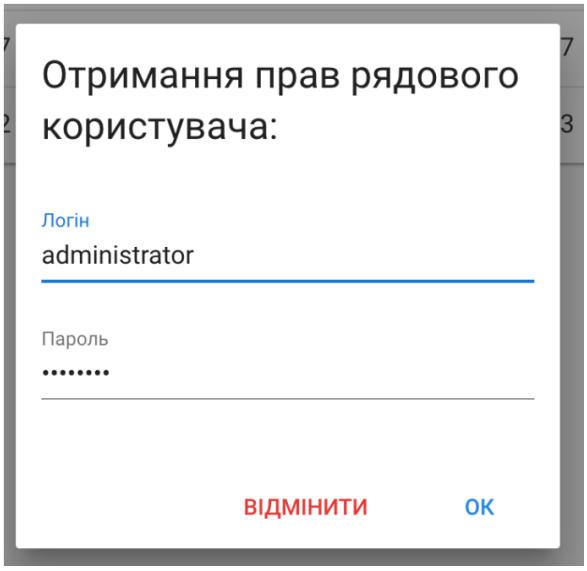


Рисунок 3.5 – Вигляд вікна логінізації користувача

На серверні стороні буде реалізований сервіс, який використовуватиметься для покриття захистом усіх необхідних користувацьких запитів:

```
public async getByPassword(login: string, password: string): Promise<UserEntity> {
    const { connection } = this.connectionManager;
    const repo = (await connection).getRepository(UserEntity);
    const entity = await repo.findOne({ login, password }, { relations: ['role', 'role.rights'] });
    if (entity !== undefined) {
        return entity;
    } else {
        throw new ForbiddenException('Invalid login and/or password');
    }
}
```

Рисунок 3.6 – Вигляд методу отримання користувача за паролем

Даний сервіс налагоджує зв’язок між алгоритмом обробки запитів та алгоритмом генерації токенів, також сервіс використовується для співставлення з існуючею базо даних вже введених адміністратором користувачів.

3.4 Реалізація II-го кроку користувацької автентифікації

Другий крок користувацької автентифікації передбачає в собі використання налаштовуваного модулю з реалізованим методом графічного паролю. Даний модуль використовуватиметься для отримання користувачем доступу до функцій редагування базою даних.

В додаток користувачеві буде пропонуватись вибрати розмірність матриці на якій будується її ключ. Вигляд вікна введення користувацького графічного паролю зображено на рис. 3.7.

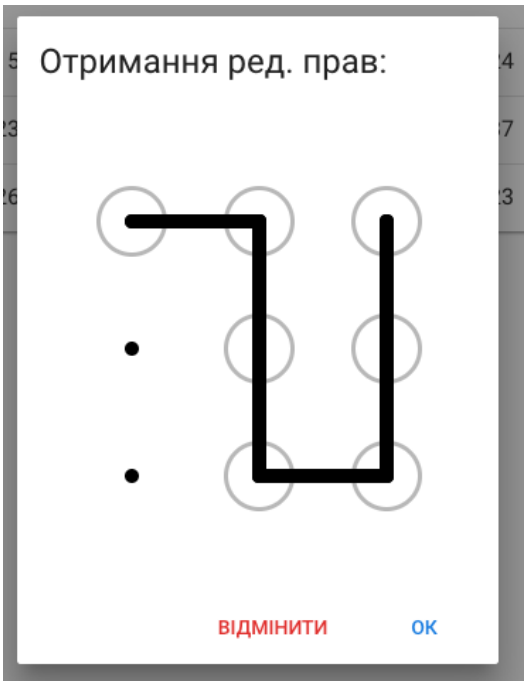


Рисунок 3.7 – Вигляд вікна введення графічного паролю

Для збереження графічного паролю на сервері може використовуватись два варіанти:

- збереження інформації у вигляді масиву типу «ключ: значення» приклад:

```
{1:3,2:1,3:4,4:2,5:9,6:7,7:8,8:5,9:6};
```

- збереження інформації на сервері за допомогою алгоритму patternCaptcha, який використовує збереження графічних паролів у зображеннях.

У зв’язку з тим, що користувацькі автентифікаційні дані повинні зберігатись у захищеному, не відкритому стані будь-який варіант гашується функцією SHS-3 і вже після підставлятись та перевіряти збіжність з введеним графічним ключем.

Даний графічний пароль реалізовується та підключається:

```
<link href=”patternLock.css”  rel=”stylesheet” type=”text/css” />
<script src=”jquery.js”></script>
<script src=”patternLock.js”></script>
```

та викликається у будь якому місці програми за допомогою:

```
var lock = new PatternLock(“#patternContainer”);
```

Розмірність задаватиметься за допомогою:

```
var lock= new PatternLock('#patternHolder',{matrix:[4,4]});
```

На сервері автентифікація для двох перших кроків отримання повного доступу об'єднана у AuthController'і, який відповідає за вхід/вихід користувача, отримання токенів та ін., проте саме авторизація за допомогою графічного ключа проходить за допомогою передачі сформованого на стороні клієнту масиву ключ-значення та виконанні на основі цього методу:

`Public async signInByVisualKey(credentials: IvisualKeyCredentials): Promise<IauthResponse>`, в якому використовується метод, зображений на рис. 3.8:

```
public async getByVisualKey(login: string, visualKey: string): Promise<UserEntity> {
    const { connection } = this.connectionManager;
    const repo = (await connection).getRepository(UserEntity);
    const users = await repo.createQueryBuilder('user')
        .innerJoinAndSelect('user.role', 'role')
        .innerJoinAndSelect('role.rights', 'right')
        .innerJoinAndSelect('user.visualKeys', 'key')
        .where('user.login = :login AND key.value = :visualKey', { login, visualKey })
        .getMany();
    switch (users.length) {
        case 0:
            throw new ForbiddenException('Invalid login and/or visual key');
        case 1:
            {
                const user = users[0];
                const canUseVisualKeyRight = (user.role.rights || []).find((right) => right.policy ===
                ACCESS_RIGHT.CAN_USE_VISUAL_KEY);
                if (canUseVisualKeyRight ? canUseVisualKeyRight.value : user.role.defaultPolicy) {
                    return user;
                } else {
                    throw new ForbiddenException('Can not use visual keys');
                }
            }
        default:
            throw new ForbiddenException('Too many users found');
    }
}
```

Рисунок 3.8 – Вигляд методу отримання користувачів за графічним ключем

Таким чином проходячи даний етап авторизації користувач підтверджує свої права на використання методів, які доступні тільки редакторам.

Цей метод є частиною `sessionService`, який розміщений на сервері та використовується для роботи з БД, створення таблиць, внесення редагувань та перевірки користувацьких прав на використання функцій по керуванню вже створеними таблицями.

3.5 Реалізація функціоналу по керуванню базою даних

Після проходження автентифікації графічним паролем, що розміщена на другому рівні користувач отримує доступ до базових функцій по редагуванню таблиць та бази даних вцілому (відкривається сесія користувача).

Сесія користувача – захищене з'єднання яке існує поки користувач не викличе функцію logout чи розірве з'єднання з сервером. Особливістю сесії є логування кожного запиту та перевірка кожного користувацького запиту на можливість виклику функції (відповідність функції ролі користувача, який її викликає):

```
@UseGuards(AuthGuard('jwt'), CanModifyDataGuard)
```

Запити які надходять на сервер після автентифікації бувають:

- створення таблиці;
- видалення таблиці;
- очистка таблиці;
- добавлення рядка в таблицю;
- зчитування таблиці;
- видалення рядка з таблиці.

Перед виконанням кожного з методів також перевіряється токен користувача, який відправляє запит.

Методи редагування/створення/видалення таблиці перевіряються за допомогою `AccessGuard`, який перевіряє доступність методів для користувача (читання-редагування-адміністрування).

```
public canActivate(this: AccessGuard, context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest() as IAuthRequest;
    return request && request.user && all(map((x) => request.user.role.rights[x], this._rights));
}
```

Самі методи є викликаними в `ConnectionProvider` SQL-запитами, наприклад – очистка таблиці (рис 3.9):

```
public async truncateTable(this: PublicDataProvider, tableName: string): Promise<void> {
    const databaseSchema = await this.schema;
    const tableSchema = databaseSchema[tableName];
    if (tableSchema) {
        const connection = await this.connectionProvider.connection;
        await connection.query(`TRUNCATE public.${escapeIdentifier(tableName)}`);
    } else {
        throw new BadRequestException(`Invalid table ${escapeIdentifier(tableName)}`);
    }
}
```

Рисунок 3.9 – Вигляд методу очистки таблиці

ConnectionProvider є модулем серверної частини додатку, яка відповідає за підключення безпосередньо до бази даних та виконання різноманітних SQL-запитів.

```
public readonly connectionOptions = {
    type: 'postgres',
    host: '127.0.0.1',
    port: 5432,
    username: 'postgres',
    database: 'test',
    synchronize: true,
    entities: [
        EncryptedTokenEntity,
        RoleAccessEntity,
        SessionEntity,
        UserEntity,
        UserRoleEntity,
        VisualKeyEntity,

        SchemaTableEntity,
        SchemaColumnEntity,
        SchemaConstraintColumnUsageEntity,
        SchemaTableConstraintEntity,
    ],
    namingStrategy: new UnderscoreNamingStrategy('underscore'),
} as ConnectionOptions;
```

Рисунок 3.10 – Вигляд створення підключення на стороні серверу

3.6 Реалізація III-го кроку користувацької автентифікації

Третій крок передбачає в собі реалізацію модулю, який надаватиме додатковий рівень захисту для отримання користувачем доступу до адміністраторських функцій по керуванню таблицями у БД. Даний крок реалізовуватиметься за допомогою алгоритму перевірки наявності у користувача необхідного файлу.

Для правильного підтвердження свої адміністраторської ролі користувачеві відвантажуватиметься спеціально згенерований файл, який у собі містить загешовані дані про користувацький профіль та зашифровані спеціальним секретним кодом-ключем, який формується на основі усіх даних, що використовувались користувачем при проходженні двох перших етапів.

Використання необхідного компоненту бібліотеки Bootstrap 4 у коді програми має такий вигляд:

```
<input id="input-id" type="file" class="file" data-preview-file-type="text">
```

Для завантаження необхідного файлу було реалізовано модуль відвантаження його на сервер та розшифрування його вмісту. У зв’язку з цим користувацький інтерфейс має вікно, зображене на рис 3.11.

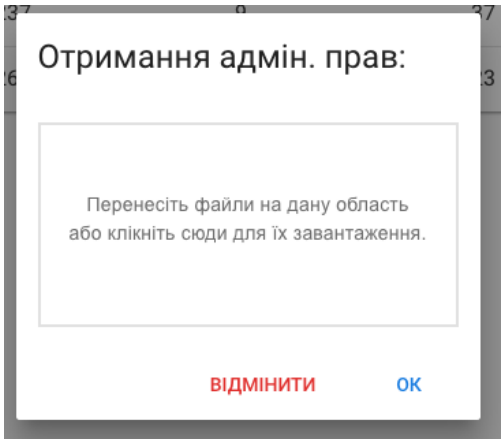


Рисунок 3.11 – Вигляд вікна вибору файлу-ключа

Сервером буде проводитись повне розшифрування та гешування поточних даних і співставлення їх з розшифрованим значенням. Слід зауважити, що дані на сервері зберігаються у захищеному вигляді та формуються «на льоту» для більшої захищеності.

3.6.1 Реалізація алгоритму перевірки відповідності даних у таблицях

У ході розробки тестового проекту, зображеного на рис. 3.12 було протестовано можливості технології blockchain та визначено можливі варіанти використання даної технології.

Рисунок 3.12 – Вигляд головного вікна програми тестового проекту

Дана технологія може бути використана для забезпечення максимального захисту цілісності інформації, яка будь-яким чином відноситься до бази даних. Насамперед вона використовуватиметься для зв'язування та підтвердження цілісності даних у таблицях бази даних. Крім атрибутів передбачається налаштування збереження інформації про користувачів, ролі бази даних і персональні налаштування кожного з рівнів, передбачених адміністратором бази даних.

Дані у таблиці зазвичай є дуже важливою інформацією, яку необхідно захистити від зламу та небажаної зміни. Для захисту таблиці було реалізовано алгоритм формування двозв'язного ланцюга, який формується з сусіднік комірок та використовує алгоритм перевірки валідності такої ж як і у популярні на сьогодні технології Blockchain.

Вигляд ланцюгу (рис. 3.8) є схематичним виглядом моделей інформації, які є зв'язаними між собою та містять поля з геш-значеннями атрибутів (prevHash, currentHash), порядковим номером (number) та назвою атрибуту таблиці (data).

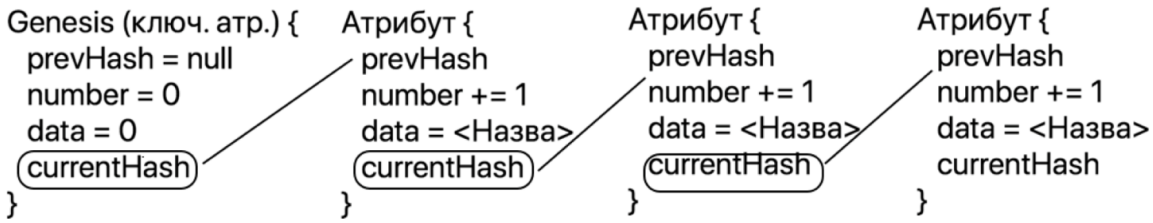


Рисунок 3.13 – Вигляд двозв'язного ланцюгу атрибутів таблиці

Для перевірки валідності інформації було передбачено алгоритм:

```
SET_VALIDITY(state) {
    for (let i = 1; i < state.chain.length; i++) {
      const currentBlock = state.chain[i];
      const previousBlock = state.chain[i - 1];
      if (
        currentBlock.hash !==
        currentBlock.calculateHash()) {
        state.chain[i].isValid = false;
      } else if (
        currentBlock.previousHash !==
        previousBlock.calculateHash()) {
        state.chain[i - 1].isValid = false;
      } else {
        state.chain[i - 1].isValid = true;
        state.chain[i].isValid = true;
      }
    }
  }
```

Рисунок 3.14 – Вигляд «хука» перевірки правильності введених даних

Реалізація даного алгоритму дозволяє получити контроль над змінами у базах даних і використати перевірку відповідності як компрометування зловмисника.

Весь алгоритм можна представити у такому вигляді (рис. 3.15):

1) база заповнюється новими даними і відповідно до цього формується ланцюг даних;

2) зловмисник будь-яким способом отримавши доступ до файлів бази даних змінює атрибути у базі;

3) алгоритм перевірки відповідності та перерахунку геш-значень запускається тільки у випадку зміни інформації авторизованими користувачами з відповідними ролями, які мають доступ до функцій редагування;

4) відповідно до цього робиться висновок, що вузли ланцюгу, які «вилітають» з нього не є довіреними і пройшли неавторизовані зміни бази даних.

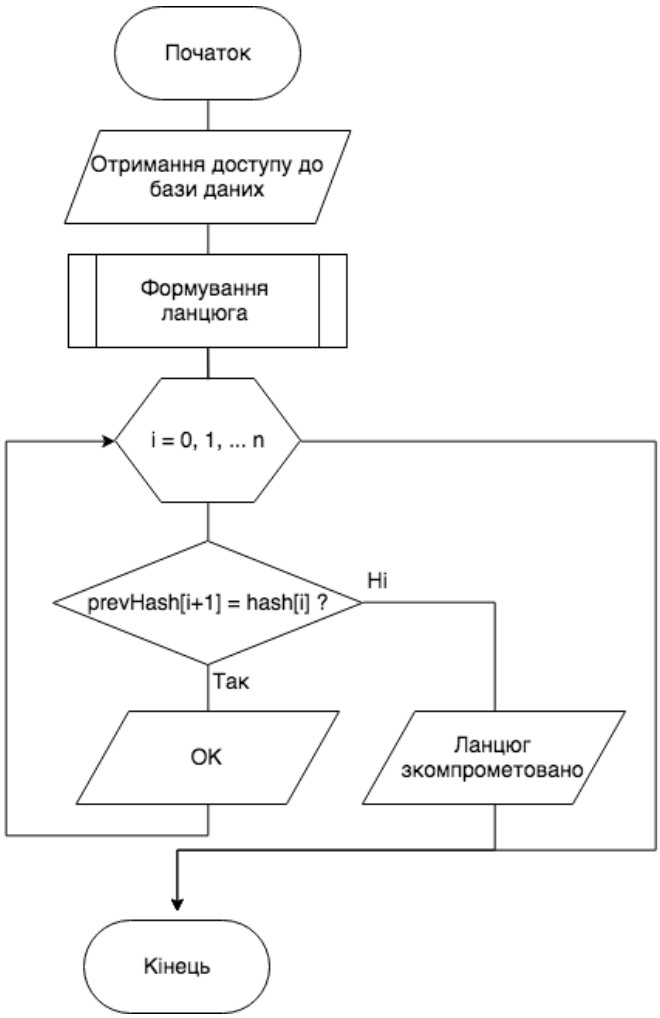


Рисунок 3.15 – Вигляд алгоритму перевірки цілісності атрибутів у БД

При навігації по базі даних будь-який користувач, що має доступ до функцій читання таблиць може побачити, що зміни зловмисника відобразились.

3.6.2 Реалізація основного інтерфейсу керування таблицями

Основним інтерфейсом додатку є інтерфейс керування таблицями у базі даних. Даний інтерфейс дозволяє:

- редагувати інформацію;
- добавляти інформацію;
- редагувати атрибути;
- видаляти таблиці.

Функції зміни БД в розробці є настільки ж необхідними і важливими, як і методи автентифікації, тому розміщення методів автентифікації знаходиться в ті ж частині, що і інші методи. Загальний вигляд інтерфейсу програмного засобу зображено на рис 3.16.

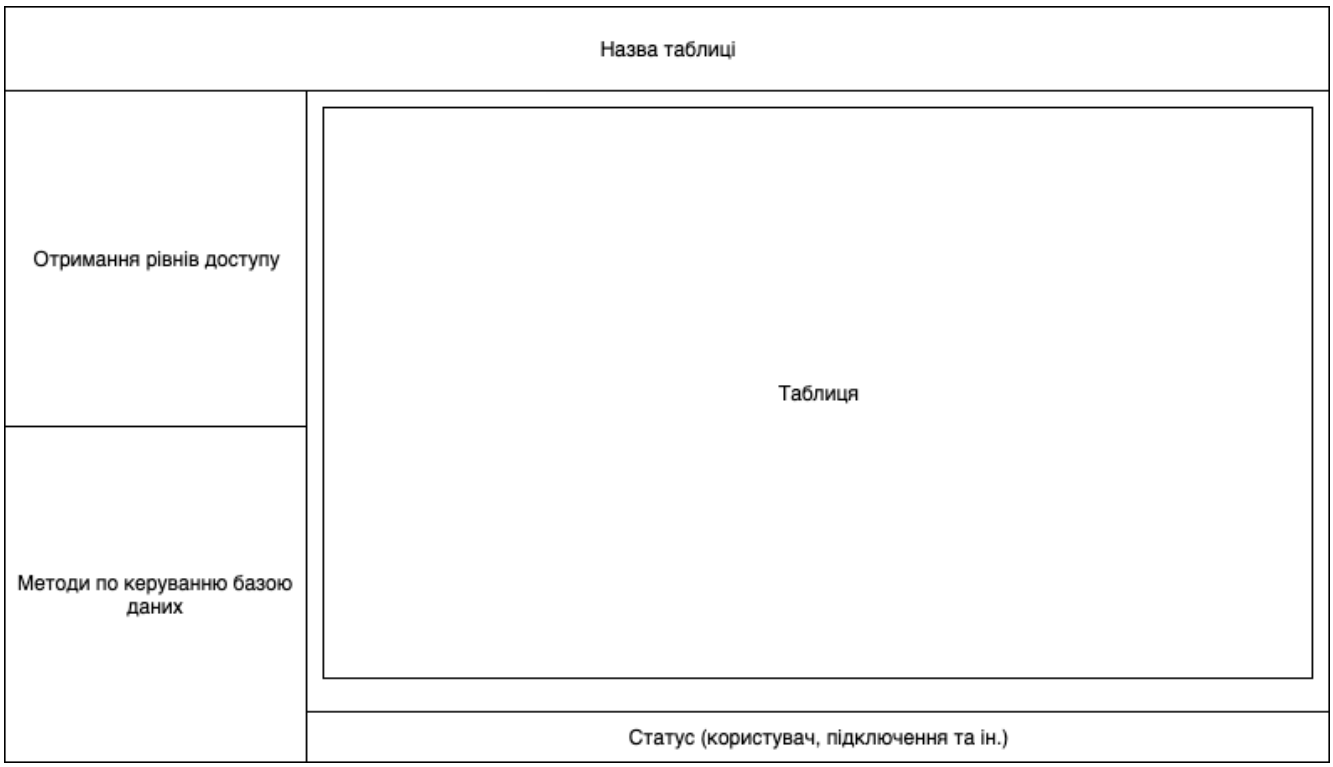


Рисунок 3.16 – Схематичний вигляд основного інтерфейсу програмного засобу

Вибір функцій, методи виклику модулів підтвердження доступу до нових рівнів зображені на рис. 3.17.

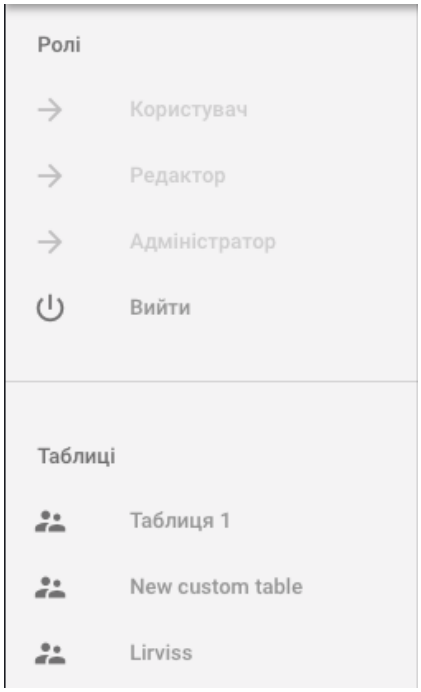


Рисунок 3.17 – Вигляд головного меню управління додатком

Для відображення інформації у вигляді таблиці було реалізовано простий табличний інтерфейс (рис. 3.18).

Dessert (100g serving)	Calories	Fat (g)	Carbs (g)	Protein (g)	Iron (%)
Frozen Yogurt	159	6	24	4	1%
Ice cream sandwich	237	9	37	4.3	1%
Eclair	262	16	23	6	7%

Рисунок 3.18 – Вигляд користувацької таблиці

Реалізація інтерфейсу таблиці відбувається за допомогою бібліотеки Vuetify та відповідного використання компонентів (рис. 3.19):

```
<thead>
  <tr>
    <th class="column text-xs-left" style="font-size: 16px;" v-for="item in headers">{{ item.text }}</th>
  </tr>
</thead>
<tbody>
  <tr v-for="item in table">
    <td v-for="(column, key, index) in item" v-if="index != 0" style="width: 217px;">
      <p v-if="!isEditModeEnabled" style="font-size: 16px;"> {{ column }}</p>
    </td>
  </tr>
</tbody>
```

Рисунок 3.19 – Вигляд відображення таблиці на частині клієнта

Відповідно до функцій, які вибирає користувач у полях таблиці вмикаються та вимикаються поля редагування та після формується запит на захищений сервер зі зміненою інформацією. Для найкращого захисту інформація відправляється на сервер відразу у захищеному стані.

3.6.3 Низькорівневий механізм розмежування користувацького доступу

Інтерфейсна частина програмного засобу має ряд змінних, які дозволяють вимикати та вмикати використання деяких функцій програмного засобу. Більшість функціоналу програмного засобу можна виконати з будь-якої точки інтерфейсу, проте дане виконання може бути доступне тільки користувачам з відповідними правами.

Дане розмежування доступу реалізовано на нижчому рівні програмного засобу, а саме сервері, який відповідає за криптографічні алгоритми та зберігає усю користувацьку інформацію. Вигляд зв’язків користувацького простору імен бази даних зображено на рис. 3.20.

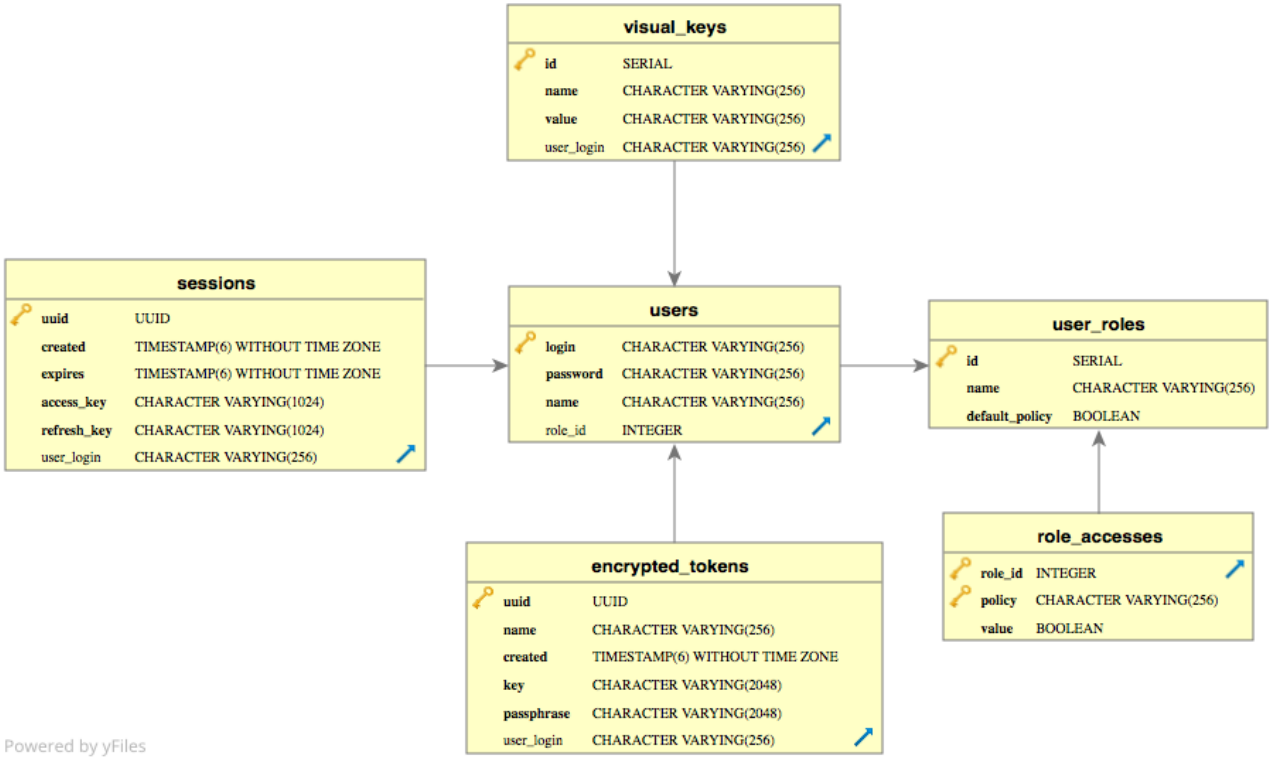


Рисунок 3.20 – Вигляд користувацького простору імен

Даний простір імен відповідає за збереження, отримання та видачу певних прав користувачам (як новим, так і вже існуючим).

Користувацьку інформацію, яку отримує інтерфейсна частина ПЗ формує дана схема. Для додатку вона відображається у такому вигляді:

```
export interface IUser {
  id: number;
  name: string;
  rights: IUserRights;
}
```

Рисунок 3.21 – Вигляд інтерфейсу користувача

Користувацькі права реалізовані у такому вигляді (інтерфейс):

```
export interface IUserRights {
  canUseVisualKey: boolean;
  canUseEncryptedToken: boolean;
  canCreateTables: boolean;
  canDropTables: boolean;
  canInsert: boolean;
  canSelect: boolean;
  canUpdate: boolean;
  canDelete: boolean;
}
```

Рисунок 3.22 – Вигляд користувацьких доступів

Об’єкт метаданих користувача, що зберігається у користувацькому просторі імен формується при створенні користувача та на льоту використовуєься при перевірці відповідності інших запитів:

```
public async buildPayload(this: SessionsManager, session: SessionEntity): Promise<IUser> {
    const role: IUserRole = {
        id: session.user.role.id,
        name: session.user.role.name,
        rights: {},
    };
    for (let key in ACCESS_RIGHT) {
        const name = ACCESS_RIGHT[key] as ACCESS_RIGHT;
        role.rights[name] = session.user.role.defaultPolicy;
    }
    for (let right of session.user.role.rights || []) {
        role.rights[right.policy] = right.value;
    }
    return {
        session: session.uuid,
        login: session.user.login,
        name: session.user.name,
        role: role,
    };
}
```

Рисунок 3.23 – Вигляд перевірки користувацьких доступів

Відповідно до отриманої інформації програмний застосунок блокує виконання певних функцій для користувача. Оскільки дана інформація зберігається на основні обчислювальні частині програмного застосунку, а саме сервері, то було обрано написати функцію-відслідковувача [28] користувацьких запитів, а саме функцію, що буде відслідковувати запити на управління таблицями у базі даних.

```
export class AccessGuard implements CanActivate {
    private readonly _rights: ACCESS_RIGHT[];

    public constructor(rights: ACCESS_RIGHT[]) {
        this._rights = rights;
    }

    public canActivate(this: AccessGuard, context: ExecutionContext): boolean {
        const request = context.switchToHttp().getRequest() as IAuthRequest;
        return request && request.user && all(map((x) => request.user.role.rights[x], this._rights));
    }
}
```

Рисунок 3.24 – Вигляд відслідковувача доступів

Дана функція отримує на вхід роль користувача та список ролей, які існують у базі даних. При отриманні доступних для користувача ролей функція

повертає вверх контексту результат перевірки, а саме можливість користувачем використання даної функції.

3.7 Портування програмного коду на різні операційні системи

Фреймворк для розробки клієнтських частин Electron дозволяє писати один код застосунку і запускати на різних операційних системах. Для цього web-додаток поміщається в захищене середовище Chromium останніх версій та «обвертається» у відповідний фрейм (Windows, MacOS, Linux).

Для цього використовується допоміжний пакет electron-builder, який відповідає за створення готового до розповсюдження додатку, який підтримує автематичне оновлення. Він дозволяє зкомпілювати нативні для кожної ОС додатки (у випадку, коли немає «залежностей» від нативних бібліотек платформи), підписати сертифікатом, специфічним для операційної системи та вподальшому дозволить автоматично оновлювати програмний додаток новими змінами у функціоналі.

Для початку слід визначити конфігурацію вікна (розмір області відображення клієнтського вікна):

```
mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js')
    }
  })
```

Рисунок 3.25 – Створення вікна

Та підвантажити головний файл з залежностями:

```
mainWindow.loadFile('index.html')
```

Після цього основним в конфігурації є визначення поведінки закриття вікна:

```
  mainWindow.on('closed', function () {
    mainWindow = null
  })
}
```

та специфічна для MacOS перевірка для повного закриття:

```
app.on('window-all-closed', function () {
  if (process.platform !== 'darwin') app.quit()
})
```

Після цього при встановленні electron-builder[]:

```
yarn add electron-builder --dev
```

слід налаштувати середовище та id додатку для правильного отримання дозволів та розпізнавання системою:

```
"build": {
  "appId": "mkr.id",
  "mac": {
    "category": "public.app-category.developer-tools"
  }
}
```

Рисунок 3.26 – Налаштування середовища

Після цього для того щоб зформувати готовий для запуску застосунок слід використати в терміналі будь-якої ОС команду:

```
npm run electron:build
```

Також додатоково треба запустити серверну частину програмного застосунку за допомогою:

```
npm run serve
```

Результатом буде готовий для запуску додаток, який може працювати на будь-якій сучасній операційній системі.

3.8 Висновки

Було проведено аналіз та обґрунтування вибору мови та фреймворків для реалізації системи керування базами даних, як користувацької частини, так і серверної частини. Було описано процес програмної реалізації додатку, з точним описанням функцій системи. Була розроблена схема роботи алгоритму перевірки змін у базі даних, розроблений простір імен для керування користувацьким доступом, графічна схема інтерфейсу програмного продукту.

4 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАСОБУ

Тестування проаграмного забезпечення – перевірка відповідності фактичних результатів роботи програми очікуваним результатам та надання висновків про відсутність програмних помилок чи дефектів в роботі програмного засобу.

4.1 Аналіз методик тестування

Тестування допомагає виявити помилки, прогалини логіки чи відсутність необхідних вимог при розробці функціональних та логічних рівнів застосунку. Зазвичай це робиться за допомогою ручних або автоматизованих інструментів.

Тестування за допомогою ручних інструментів це зазвичай процес оцінки окремого елемента програмного засобу, який дозволяє виявити відмінності між заданими вхідними даними і очікуваним результатом. Найчастіше тестування використовується під час розробки і дозволяє перевіряти модулі програмного засобу по ходу розробки та на льоту виправляти знайдені помилки. Завдяки такому підходу можна виявити помилки на ранніх стадіях розробки. Між іншим тестування програмного засобу це не просто знаходження помилок, а також процес верифікації (перевірки відповідності вимогам, які були встановлені на початковому етапі роботи) та валідації (гарантує, що продукт відповідає вимогам наприкінці етапу розробки).

Тестування виконує дві основні задачі:

- демонстрація якості функціонування ПЗ;
- знаходження і усунення помилок в ПЗ.

Тестування програмного продукту вважається однією з основних задач по розробці ПЗ. Існує безліч варіантів вирішення завдання такого типу та верифікації ПЗ, проте слід зазначити, що ефективне тестування комплексних систем, які складаються з клієнтської та серверної частини – трудомісткий процес, який не може закінчись на виконанні шаблонних завдань та процедур або лише їх створенні.

До основних методів тестування відносять:

- тестування «білого ящика» та «чорного ящика»;
- статистичне та динамічне тестування;
- регресивне тестування;
- стресове тестування[29].

Тестування білої скрині передбачає огляд структури коду. Воно проходить за розуміння тестувальником внутрішньої структури програмного засобу і дозволяє використати тести, які можуть забезпечити повне покриття та відповідність специфікації програмного засобу, та всі внутрішні компоненти були реалізовані відповідно усім правилам.

Тестування чорної скриньки використовується у випадках, коли тестувальник не має знати внутрішню роботу «чорної скриньки» чи програми. Чорна скринька постає перед ним у вигляді функціональної одиниці, яка може виконувати якісь дії, проте достовірно про це невідомо. Основна увага в тестуванні чорної скриньки приділяється функціональним можливостям системи в цілому.

Статичне тестування – це тип тестування, який виконується без коду. Воно виконується на стадії документації під час етапу тестування. Таке тестування включає в себе аудит покрокової інструкції на відповідність з результатами проекту. Статичне тестування не виконує код, але перевіряються конфігурації назв. Таке тестування застосовується для тестів, планів тестів, дизайнерських документів. Група випробувань повинна проводити статичне тестування, оскільки дефекти, виявлені під час цього типу випробувань, є економічно вигідними з точки зору проекту.

Стрес тестування проводиться, коли система піддавалася стресовому навантаженню за своїми технічними характеристиками, щоб перевірити, як і коли вона не працює. Це виконується під великим навантаженням при доступі до бази даних, безперервного введення в систему або завантаження бази даних.

Регресія передбачає повторне тестування частин програми, які не мали явних змін у ході розробки. Регресійне тестування – це тип тестування, в якому

тестові випадки повторно виконуються, щоб перевірити, чи працює попередня функціональність програми, а зміни не спричиняють нових помилок у виконанні. Цей тест зазвичай виконують на з новою збіркою продукту, коли існує значна зміна початкового функціоналу.

Тестувальники використовують функціональне тестування при отриманні доступу до нової версії. Метою цього тесту є перевірка змін, що були внесені в існуючу функцію та нещодавно додану функціональність. При проходженні цього тесту, тестувальник повинен перевірити, чи існуюча функціональність працює, як очікувалося і нові зміни не вносять ніяких недоліків.

В результаті проведеного аналізу, для тестування розробленого продукту було застосовано метод «білого ящика» та регресивного тестування, оскільки вони дають змогу за короткий проміжок часу «покрити» роботу всієї системи, і підтримувати її надійність після внесення покращень до вже існуючого її функціоналу.

4.2 Перевірка правильності роботи додатку

Програмний засіб реалізований мовою програмування JavaScript та зібраний для виконання у будь-якій операційній системі за допомогою фреймворку Electron. Попередня тестування додатку буде проводитись на MacOS 10.15.1.

Програмний засіб має інтуїтивний інтерфейс та надає користувачеві інформацію про функції, які йому доступні шляхом вимкнення та увімкнення кнопок керування. Вимкнуті керуючі елементи надають користувачеві інформацію про те, що необхідні йому функції не є доступними на даний момент чи користувацькі права його облікового запису не дозволяють йому виконувати певні операції. Приклад вимкнутих кнопок отримання користувацьких прав у авторизованого користувача з правами адміністратора, який пройшов усі 3 рівні автентифікації зображено на рис. 3.10.

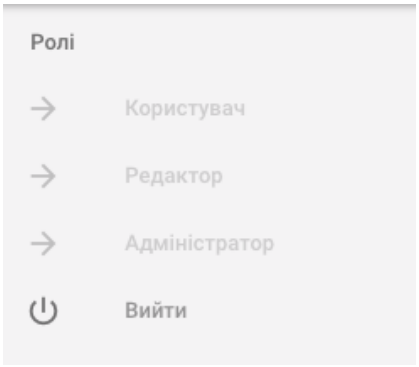


Рисунок 3.10 – Вигляд вимкнених кнопок отримання нових рівнів користувацької автентифікації

У разі розблокування зловмисником необхідних кнопок додатково реалізований захист на редагування та видалення таблиці, шляхом перевірки користувацьких прав нальоту. Відповідно коли користувач розблоковує собі доступ до певної функції, до якої він не повинен мати доступ – він не зможе виконати будь-яких операцій вище свого рівня доступу.

Рівні отримання користувацьких прав відображаються у випадаючих вікнах. Після введення своїх автентифікаційних даних формується запит з геш-значеннями введених даних та відправляється на сервер у якому зіставляється з загешованою відповідною користувацькою автентифікаційною інформацією. Приклад вікна введення користувацьких даних зображено на рис 3.11.

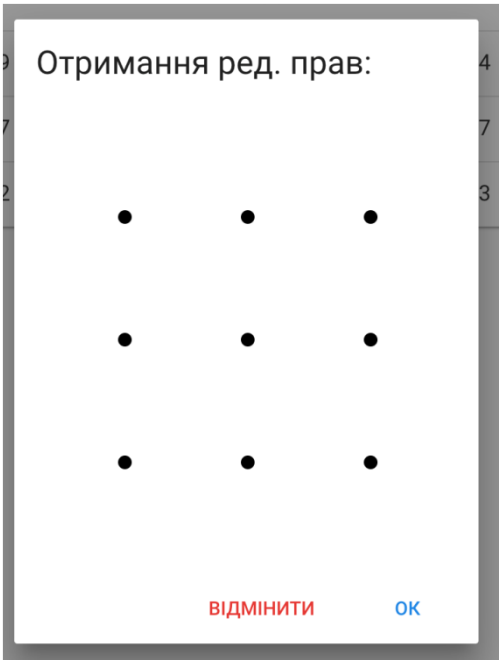


Рисунок 3.11 – Вигляд випадаючого вікна

Доступ до таблиць створеної бази даних надається тільки після авторизації користувача за логіном та паролем (отримання прав на читання). Завантаження таблиць проходить шляхом отримання списку таблиць з серверу. Редагування таблиць може увімкнутись тільки при умові користувацького рівня доступу типу редактор. Вигляд користувацької таблиці з увімкненою функцією редагування зображено на рис. 3.12.

Dessert (100g serving)	Calories	Fat (g)	Carbs (g)	Protein (g)	Iron (%)
Frozen Yogurt	159	6	24	4	1%
Ice cream sandwich	237	9	37	4.3	1%
Eclair	262	16	23	6	7%

Рисунок 3.12 – Вигляд таблиці бази даних під час редагування

Так, як користувацька таблиця може редагуватись на льоту передбачено механізм, що відправляє інформацію тільки після завершення інформації задля передбачення конфліктів між користувацькими діями.

Слід зауважити, що усі запити, які йдуть на сервер програмного засобу добавляються у чергу і виконуються послідовно. Через це можливість появи конфліктів при редагуванні таблиць різними користувачами зводиться до нуля.

4.3 Аналіз результатів роботи механізму розмежування доступу

Згідно з реалізованою клієнт-серверною архітектурою програмного засобу усі користувацькі зміни, автентифікацйні методи та криптографічні алгоритми виконуються на серверній частині додатку.

У зв’язку з цим все керування БД відбувається через користувацькі запити, які перевіряються розробленим модулем перевірки користувацької ролі.

Після правильного проходження методу авторизації користувачеві приходить набір даних, що надає клієнтські частині програми команду провести налаштування для доступних користувачеві викликів функцій:

```
{
    "accessExpiresIn": 3599,
    "refreshExpiresIn": 7199,
    "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. ... ",
    "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. ... ",
    "user": {
```

```
        "session": "01d8c822-fff1-4f6b-8d1e-37e0dcb2e8d0",
        "login": "root",
        "name": "root",
        "role": {
            "id": 1,
            "name": "Root",
            "rights": {
                "CAN_USE_VISUAL_KEY": true,
                "CAN_USE_ENCRYPTED_TOKEN": true,
                "CAN_CREATE_TABLES": true,
                "CAN_DROP_TABLES": true,
                "CAN_MODIFY_TABLES": true,
                "CAN_INSERT": true,
                "CAN_SELECT": true,
                "CAN_UPDATE": true,
                "CAN_DELETE": true
            }
        }
    }
}
```


Клієнтська частина програми налаштовує користувацький інтерфейс, відключаючи використання відповідних функцій для передбачення запитів на серверну частину додатку, що не відповідають користувацькій ролі.

У зв’язку з можливістю зламу клієнтської частини додатково реалізовано функції-відслідковувачі, які у разі недоступності виконання певних функцій відповідають на запити інформацією про неможливість користувача виконувати певні дії:

1. Користувач не авторизований.


```
{
    "statusCode": 401,
    "error": "Unauthorized"
}
```


2. Помилка виконання запиту.


```
{
    "statusCode": 404,
    "error": "Not Found",
    "message": "Cannot POST /data/select/row"
}
```


У більшості випадків дані функції дають майже моментальну відповідь на користувацькі дії і не навантажують зв’язок з БД для інших користувачів – в середньому відповідь з помилкою про виконання йде ~5-10 мс та має вагу 250-350 байт. Згідно з такими показниками можна зробити висновок, що серверна

частина додатку є стійкою для можливих атак, спрямованих на відмову обладнання.

4.4 Висновки

Під час виконання четвертого розділу було представлено та описано методики тестування програмного забезпечення. Після аналізу методів тестування було обрано найоптимальніші у випадку розробки даної системи керування базами даних, а саме метод «білого ящика» та регресивного тестування.

Під час виконання четвертого розділу було проведено тестування системи на різних операційних системах, перевірка основних методів по розмежуванню користувацького доступу, тестування серверної частини додатку.

В результаті тестування можна зробити висновок, що система повністю відповідає вхідним вимогам, які були поставлені на початку розробки СКБД.

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Оцінювання комерційного потенціалу розробки

Метою проведення технологічного аудиту є оцінювання комерційного потенціалу розробки. Для проведення технологічного аудиту було залучено 2-х незалежних експертів. Такими експертами будуть керівник магістерської роботи – к.т.н., доц. каф. ПЗ Романюк Оксана Володимирівна, к.т.н., доц. каф. ПЗ – Ракитянська Ганна Борисівна, к.т.н..

Здійснюємо оцінювання комерційного потенціалу розробки за 12-ма критеріями за 5-ти бальною шкалою.

Результати оцінювання комерційного потенціалу розробки наведено в таблиці 5.1.

Таблиця 5.1 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта	
	1. Експерт 1	2. Експерт 2
	Бали, виставлені експертами:	
1	4	4
2	3	3
3	3	4
4	4	3
5	3	4
6	4	4
7	3	3
8	4	4
9	4	3
10	4	3
11	3	4
12	3	4
Сума балів	$СБ_1 = 43$	$СБ_2 = 43$
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_{1}^{3} СБ_i}{2} = 43$	

Отже, з отриманих даних таблиці 5.1 видно, що нова розробка має високий рівень комерційного потенціалу.

5.2 Прогнозування витрат на виконання науково-дослідної роботи та конструкторсько–технологічної роботи.

Для розробки нового програмного продукту необхідні такі витрати.

Основна заробітна плата для розробників визначається за формулою (5.1):

$$З_о = \frac{М}{Т_р} \cdot t, \quad (5.1)$$

де М- місячний посадовий оклад конкретного розробника;

$Т_р$ - кількість робочих днів у місяці, $Т_р$ = 21 день;

t - число днів роботи розробника, t = 45 днів.

Розрахунки заробітних плат для керівника і програміста наведені в таблиці 5.2.

Таблиця 5.2 – Розрахунки основної заробітної плати

Працівник	Оклад М, грн.	Оплата за робочий день, грн.	Число днів роботи, t	Витрати на оплату праці, грн.
Науковий керівник	5500	261,90	10	2619
Інженер-програміст	4000	190,47	45	8571,15
Всього:				11190,15

Розрахуємо додаткову заробітну плату:

$$З_{дод} = 0,1 \cdot 11190,15 = 1119,01 (грн.)$$

Нарахування на заробітну плату операторів НЗП розраховується як 37,5…40% від суми їхньої основної та додаткової заробітної плати:

$$Н_{зп} = (З_о + З_р) \cdot \frac{\beta}{100}, \quad (5.2)$$

$$Н_{зп}= (11190,15+1119,01)\cdot \frac{36,3}{100} = 4468,22 \text{ (грн.)}.$$

Розрахунок амортизаційних витрат для програмного забезпечення виконується за такою формулою:

$$А=\frac{Ц\cdot Н_а}{100}\cdot\frac{Т}{12}, \quad (5.3)$$

де Ц – балансова вартість обладнання, грн;

$Н_а$ – річна норма амортизаційних відрахувань % (для програмного забезпечення 25%);

Т – Термін використання (Т=3 міс.).

Таблиця 5.3 – Розрахунок амортизаційних відрахувань

Найменування програмного забезпечення	Балансова вартість, грн.	Норма амортизації, %	Термін використання, міс.	Величина амортизаційних відрахувань,грн
Персональний комп'ютер	9000	25	2	375
Всього:				375

Розрахуємо витрати на комплектуючі. Витрати на комплектуючі розрахуємо за формулою:

$$K = \sum_1^n Н_і \cdot Ц_і \cdot К_і, \quad (5.4)$$

де n – кількість комплектуючих;

Ні - кількість комплектуючих і-го виду;

Ці – покупна ціна комплектуючих і-го виду, грн;

Кі – коефіцієнт транспортних витрат (приймемо Кі = 1,1).

Таблиця 5.4 - Витрати на комплектуючі, що були використані для розробки ПЗ.

Найменування матеріалу	Одиниці виміру	Ціна, грн.	Витрачено	Вартість витрачених матеріалів, грн.
Флешка	шт.	180	1	180
Пачка паперу	уп.	130	1	130
Ручка	шт.	10	1	10
Всього з урахуванням транспортних витрат				352

Витрати на силову електроенергію розраховуються за формулою:

$$В_е=В\cdot П\cdot Ф\cdot К_п\ ; \quad (5.5)$$

де В – вартість 1кВт-години електроенергії (В=1,7 грн/кВт);

П – установлена потужність комп'ютера (П=0,6кВт);

Ф – фактична кількість годин роботи комп'ютера (Ф=180 год.);

$К_п$ – коефіцієнт використання потужності ($К_п< 1$, $К_п = 0,8$).

$$В_е= 1,7 \cdot 0,6 \cdot 180 \cdot 0,8 = 146,88 \text{ (грн.)}$$

Розрахуємо інші витрати $В_{ін}$.

Інші витрати $І_в$ можна прийняти як (100…300)% від суми основної заробітної плати розробників та робітників, які були виконували дану роботу, тобто:

$$В_{ін} = (1..3) \cdot (З_о + З_р). \quad (5.6)$$

Отже, розрахуємо інші витрати:

$$В_{ін} = 1 * (11190,15 + 1119,01) = 12309,16 \text{ (грн).}$$

Сума всіх попередніх статей витрат дає витрати на виконання даної частини роботи:

$$В = З_о + З_д + Н_{зп} + А + К + В_е + І_в$$

В = 11190,15+1119,01+4468,22+375+352+146,88+12309,16= 29960,42 (грн.)

Розрахуємо загальну вартість наукової роботи $B_{заг}$ за формулою:

$$В_{заг} = \frac{В_{ін}}{\alpha} \quad (5.7)$$

де α – частка витрат, які безпосередньо здійснює виконавець даного етапу роботи, у відн. одиницях = 1.

$$В_{заг} = \frac{29960,42}{1} = 29960,42$$

Прогнозування загальних витрат ЗВ на виконання та впровадження результатів виконаної наукової роботи здійснюється за формулою:

$$ЗВ = \frac{В_{заг}}{\beta} \quad (5.8)$$

де β – коефіцієнт, який характеризує етап (стадію) виконання даної роботи.

Отже, розрахуємо загальні витрати:

$$ЗВ = \frac{29960,42}{0,9} = 33289,35 \text{ (грн.)}$$

5.3 Прогнозування комерційних ефектів від реалізації результатів розробки.

Спрогнозуємо отримання прибутку від реалізації результатів нашої розробки. Зростання чистого прибутку можна оцінити у теперішній вартості грошей. Це забезпечить підприємству (організації) надходження додаткових коштів, які дозволять покращити фінансові результати діяльності .

Оцінка зростання чистого прибутку підприємства від впровадження результатів наукової розробки. У цьому випадку збільшення чистого прибутку підприємства $\Delta П_i$ для кожного із років, протягом яких очікується отримання

позитивних результатів від впровадження розробки, розраховується за формулою:

$$\Delta П_i = \sum_1^n (\Delta П_я \cdot N + П_я \Delta N)_i \quad (5.9)$$

де $\Delta П_я$ – покращення основного якісного показника від впровадження результатів розробки у даному році;

N – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

ΔN – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки;

$П_я$ – основний якісний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки.

В результаті впровадження результатів наукової розробки витрати на виготовлення інформаційної технології зменшаться на 20 грн (що автоматично спричинить збільшення чистого прибутку підприємства на 20 грн), а кількість користувачів, які будуть користуватись збільшиться: протягом першого року – на 250 користувачів, протягом другого року – на 200 користувачів, протягом третього року – 150 користувачів. Реалізація інформаційної технології до впровадження результатів наукової розробки складала 600 користувачів, а прибуток, що отримував розробник до впровадження результатів наукової розробки – 300 грн.

Спрогнозуємо збільшення чистого прибутку від впровадження результатів наукової розробки у кожному році відносно базового.

Отже, збільшення чистого продукту $\Delta П_1$ протягом першого року складатиме:

$$\Delta П_1 = 20 \cdot 600 + (300 + 20) \cdot 250 = 92000 \text{ грн.}$$

Протягом другого року:

$$\Delta П_2 = 20 \cdot 600 + (300 + 20) \cdot (250 + 200) = 156000 \text{ грн.}$$

Протягом третього року:

$$\Delta П_3 = 20 \cdot 600 + (300 + 20) \cdot (250 + 200 + 150) = 204000 \text{ грн.}$$

5.4 Розрахунок ефективності вкладених інвестицій та період їх окупності

Визначимо абсолютну і відносну ефективність вкладених інвестором інвестицій та розрахуємо термін окупності.

Абсолютна ефективність $Е_{абс}$вкладених інвестицій розраховується за формулою:

$$Е_{абс} = (ПП - PV), \qquad (5.10)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т – період часу, протягом якого виявляються результати впровадженої НДДКР, 3 роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,1;

t – період часу (в роках) від моменту отримання чистого прибутку до точки 2, 3,4.

Рисунок, що характеризує рух платежів (інвестицій та додаткових прибутків) буде мати вигляд, рисунок 5.1.

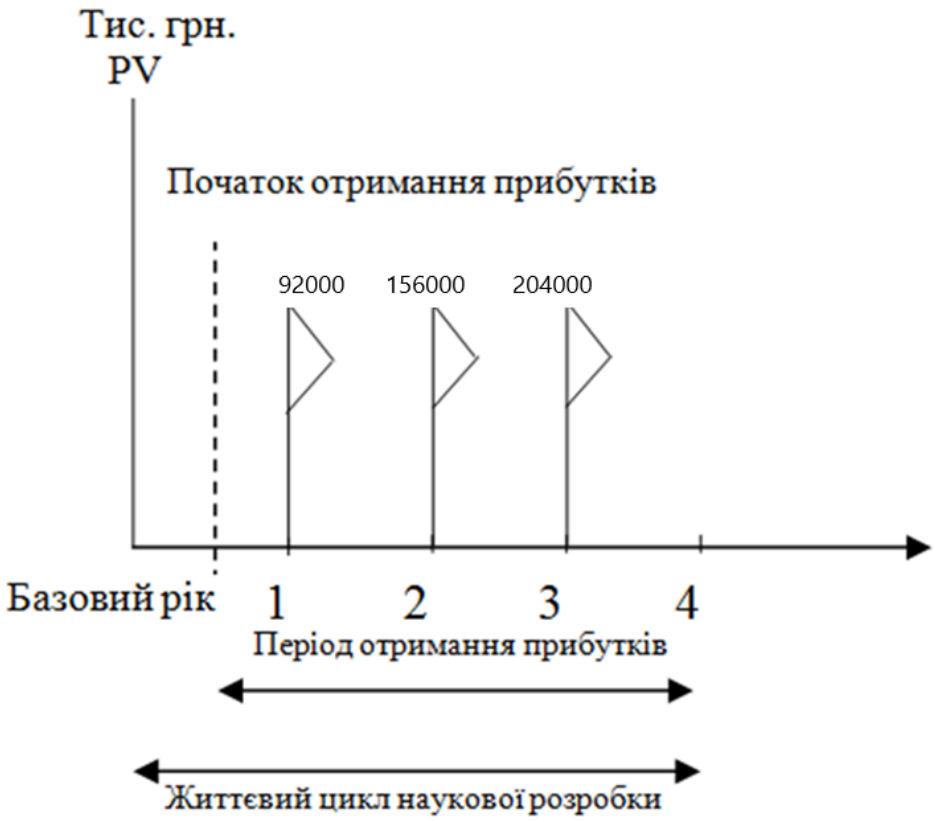


Рисунок 5.1 – Вісь часу з фіксацією платежів, що мають місце під час розробки та впровадження результатів НДДКР

Розрахуємо вартість чистих прибутків за формулою:

$$ПП = \sum_1^m \frac{\Delta П_i}{(1+\tau)^t} \quad (5.11)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн;

т – період часу, протягом якого виявляються результати впровадженої НДДКР, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,1;

t – період часу (в роках) від моменту отримання чистого прибутку до точки.

Отже, розрахуємо вартість чистого прибутку:

$$ПП = \frac{33289,35}{(1+0,1)^0} + \frac{92000}{(1+0,1)^2} + \frac{156000}{(1+0,1)^3} + \frac{204000}{(1+0,1)^4} = 365863,9 \text{ (грн.)}$$

Тоді розрахуємо $E_{абс}$:

$$E_{абс} = 365863,9 - 33289,35 = 332574,55 \text{ грн.}$$

Оскільки $E_{абс} > 0$, то вкладання коштів на виконання та впровадження результатів НДДКР буде доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$ за формулою:

$$E_в = \sqrt[T]{1 + \frac{E_{абс}}{PV}} - 1 \qquad (5.12)$$

де $E_{абс}$ – абсолютна ефективність вкладених інвестицій, грн;

PV – теперішня вартість інвестицій PV = ЗВ, грн;

$T_ж$ – життєвий цикл наукової розробки, роки.

Тоді будемо мати:

$$E_в = \sqrt[3]{1 + \frac{332574,55}{33289,35}} - 1 = 1,22 \text{ або } 122\ \%$$

Далі, розраховану величина $E_в$ порівнюємо з мінімальною (бар'єрною) ставкою дисконтування $\tau_{мін}$, яка визначає ту мінімальну дохідність, нижче за яку інвестиції вкладатися не будуть. У загальному вигляді мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau = d + f,$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2019 році в Україні d = 0,2;

f – показник, що характеризує ризикованість вкладень, величина f = 0,1.

$$\tau = 0{,}2 + 0{,}1 = 0{,}3$$

Оскільки $E_в$ = 122% > $\tau_{мін}$ = 0,3 = 30%, то у інвестор буде зацікавлений вкладати гроші в дану наукову розробку.

Термін окупності вкладених у реалізацію наукового проекту інвестицій. Термін окупності вкладених у реалізацію наукового проекту інвестицій $T_{ок}$ розраховується за формулою:

$$T_{ок} = \frac{1}{E_в}$$

$$T_{ок} = \frac{1}{1{,}22} = 0{,}81 \text{ року}$$

Обрахувавши термін окупності даної наукової розробки, можна зробити висновок, що фінансування даної наукової розробки буде доцільним.

5.5 Висновки

В даному розділі було проведено оцінювання комерційного потенціалу розробки програмного засобу.

Проведено технологічний аудит з залученням двох незалежних експертів та визначено, що рівень комерційного потенціалу розробки вище середнього.

Аналіз комерційного потенціалу розробки показав, що програмний продукт за своїми характеристиками вносить новизну в предметну галузь та випереджає аналогічні програмні продукти і є перспективною розробкою. Він має кращі цільові функціональні показники, а тому є конкурентоспроможним товаром на ринку. Існуючі переваги нової розробки дозволять швидко її поширити та популяризувати.

Згідно із розрахунками всіх статей витрат на виконання науково-дослідної, дослідно-конструкторської та конструкторсько-технологічної роботи загальні витрати на розробку складають 33289,35 грн.

Розрахована абсолютна ефективність вкладених інвестицій в сумі 332574,55 грн свідчить про отримання прибутку інвестором від комерціалізації програмного продукту.

Щорічна ефективність вкладених в наукову розробку інвестицій складає 122 %, що вище за мінімальну бар’єрну ставку дисконтування, яка складає 90%. Це означає потенційну зацікавленість інвесторів у фінансуванні розробки.

Термін окупності вкладених у реалізацію проекту інвестицій становить 0,81 року, що також свідчить про доцільність фінансування нової розробки.

ВИСНОВКИ

В ході виконання магістерської кваліфікаційної роботи був проведений аналіз стану захищеності сучасних систем керування базами даних та була проведена оцінка основних загроз. За результатами аналізу загроз було визначено, що сучасні бази даних мають потужний захист з точки зору криптографії чи передачі даних, проте є інша низка загроз: загрози надмірного доступу, поганої адміністрації бази даних, які загалом підпадають під людський фактор є основною проблемою сучасних СКБД.

Проблема користувацького доступу може бути вирішена тільки прикладними методами розмежування доступу та строгими обмеженнями доступів користувачів до функцій, введеням нових політик безпеки. Проте був представлений новий засіб для захисту бази даних від несанкціонованого доступу до інформації та функцій, які в ній зберігаються, а саме модель розшарування доступу до інформації, що зберігається в сучасних інформаційних системах, особливістю якого є розмежування користувацького доступу до різних рівнів керування базою даних, що дозволило підвищити рівень захищеності даних та функцій сучасних СКБД. Додаткового захисту подальшого розвитку отримав метод перевірки користувацьких змін, який дозволяє виявити несанкціоновані дії у таблицях бази даних.

Для демонстрації правильності роботи та перевірки дієздатності моделі було розроблено програмний засіб, який складається з клієнтської частини (використовувалась мова програмування JavaScript та фреймворк VueJS) та серверної частини (мова прогрмування JavaScript та рушій NodeJS). В результаті чого програмний додаток було протестовано та перевірено доцільність використаних програмних засобів та розроблених методів захисту.

Проведено аналіз комерційного потенціалу розробки, який довів, що програмний продукт за своїми характеристиками дає порівняно-інший рівень захисту, аніж аналогічні програмні продукти і є перспективною розробкою. Він має кращу модель доступу до функцій СКБД, функціональні показники, а тому

є конкурентоспроможним товаром на ринку. Існуючі переваги нової розробки дозволять швидко її поширити та популяризувати на усіх існуючих платформах.

Задачу магістерської кваліфікаційної роботи виконано в повному обсязі: проаналізовано наявні ризики та загрози, розроблено та описано алгоритми основних блоків роботи програмного продукту, розроблено програмну реалізацію системи та проведено ряд тестів для підтвердження коректності роботи програмного додатку та подальшої підтримки системи.

ПЕРЕЛІК ПОСИЛАНЬ

1. Микитюк І.С., Войтович О.П. Захист баз даних шляхом фрагментування користувацького доступу // Матеріали XLVII Науково-технічної конференції факультету інформаційних технологій та компютерної інженерії (2018).

2. Зрюмов, Е. А. Базы данных для инженеров: навчальний посібник / Е. А. Зрюмов, А. Г. Зрюмова; Алт. держ. техн. ун-т им. И. И. Ползунова. – Барнаул : Видав-во АлтГТУ, 2010. – 131 с.

3. Kupershtein L. M. The database-oriented approach to data protection in Android operation system / Kupershtein L. M., Voytovych O. P., Procopcuk S.O., Kaplun V.A. // Вісник ХНУ : серія Технічні науки. - №1. -2018. - С. 18-22

4. Баришев Ю. В., Каплун В. А., Неуйміна К. В. Дискреційна модель та метод розмежування прав доступу до розподілених інформаційних ресурсів // Наукові паці ВНТУ. – 2017. – №2. – 8 с.

5. Войтович О. П., Микитюк І.С. Метод захисту баз даних шляхом багатошарового користувацького доступу // «Інформаційні технології та комп'ютерне моделювання»; матеріали статей Мажнародної науково-практичної конференції, м.Івано-Франківськ, 14-19 травня 2018р. – Івано-Франківськ: п.Голіней О.М., 2018 – С. 182-185

6. В. А. Каплун, О. В. Дмитришин, Ю. В. Баришев. Захист програмного забезпечення, частина 2 – Вінниця, ВНТУ, 2014 – 105 с..

7. Каплун В.А., Дудатьєв А.В., Семеренко В.П., Захист програмного забезпечення, частина 1 – Вінниця, ВНТУ, 2005 – 140 с..

8. Кузин А. В. Базы данных: навчальний посібник для студентів вищих навчальних закладів / А. В.Кузин, С.В.Левонисова. — 5-те вид., випр. — М. : Видавничий центр «Академия», 2012. — 320 с.

9. Зрюмов, Е. А. Базы данных для инженеров: навчальний посібник / Е. А. Зрюмов, А. Г. Зрюмова; Алт. держ. техн. ун-т им. И. И. Ползунова. – Барнаул : Видав-во АлтГТУ, 2010. – 131 с.

10. Сучасні криптографічні системи: Навч. посібник. – Одеса: ВЦ ОНАЗ ім. О.С. Попова, 2007. – 152 стор.

11. Top 10 Common Database security issues [Електронний ресурс]. Режим доступу: URL : https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/july/top-10-common-database-security-issues/ – Назва з екрану.

12. Види політик безпеки [Електронний ресурс]. Режим доступу: URL : http://helpiks.org/6-26932.html – Назва з екрану.

13. Шайтанова Н. Ж., Туленгалиева М.Г. Защита информации в базах данных [Електронний ресурс]. Режим доступу: URL : http://www.rusnauka.com/10_DN_2014/Informatica/3_165120.doc.htm – Назва з екрану.

14. Микитюк І.С., Баришев Ю.В. Підхід до захисту баз даних: тези на наукову конференцію // Матеріали XLVII Науково-технічної конференції факультету інформаційних технологій та компютерної інженерії (2017).

15. Microsoft Docs. Роли уровня баз даных. [Електронний ресурс]. Режим доступу: URL: https://docs.microsoft.com/ru-ru/sql/relational-databases/security/authentication-access/database-level-roles - Назва з екрану.

16. Полтавцева М. А., Хабаров А. Р. Безопасность баз данных: проблемы и перспективы //Программные продукты и системы. – 2016. – №. 3 (115).

17. Сучасні криптографічні системи: Навч. посібник. – Одеса: ВЦ ОНАЗ ім. О.С. Попова, 2007. – 152 стор.

18. Євсеєв С.П. Гешування даних в інформаційних системах : монографія / С.П.Євсеєв, О.Ю.Йохов, О.Г.Король – Х. : Вид. ХНЕУ, 2013. – 312с.

19. Щербань Е. Что такое блокчейн, и как он работает [Електронний ресурс]. Режим доступу: URL : https://revolverlab.com/how-its-works-blockchain-6d0355c43bfc – Назва з екрану.

20. А.М. Луцків, А.М. Калинюк. Інтеграція підсистем біометричної аутентифікації у вебсервісів // Матеріали VI Міжнародної науково-технічної

конференції молодих учених та студентів. Тернопільський національний технічний університет імені Івана Пулюя, Україна

21. Олійник Г. В. Використання JWT-маркерів для аутентифікації та авторизації користувачів у web-додатках / Г. В. Олійник, С. В. Грибков // Вісник Національного університету «Львівська політехніка». Серія: Автоматика, вимірювання та керування. — Львів : Видавництво Львівської політехніки, 2017. — № 880. — С. 86–93.

22. Олійник Г. В., Грибков С. В., Литвинов В. А. Обрання програмної платформи для побудови модуля безпеки web-орієнтованої системи підтримки прийняття рішень / Г. В. Олійник, С. В. Грибков, В. А. Литвинов // Вісник Нац. ун-ту “Львівська політехніка”, серія “Автоматика, вимірювання та керування”. – 2016. – №852. – С. 137–142.

23. Darren Jones. Javascript: Novice to Ninja / by Darren Jones. — 2nd end. — SitePoint. — 2017. — 666. с.

24. MDN. Поширені питання - Вчимо веб-розробку [Електронний ресурс]. Режим доступу: URL : https://developer.mozilla.org/uk/docs/Learn/Common_questions – Назва з екрану.

25. VueJS. (офіційний сайт) [Електронний ресурс]: Режим доступу: URL : https://vuejs.org/ – Назва з екрану.

26. Node.JS. (офіційний сайт) [Електронний ресурс]: Режим доступу: URL : https://nodejs.org/uk/ – Назва з екрану.

27. VuetifyJS. (офіційний сайт) [Електронний ресурс]: Режим доступу: URL : https://vuetifyjs.com/ru/ – Назва з екрану.

28. Node.js в действии / М. Кантелон , М. Хартер, Т. Головайчук, Н. Райлих. — СПб.: Питер. — 2014. — 548 с. — ISBN 978-5-496-01079-5.

29. Тестування програм/В.В. Липа. - М.: Радіо і зв'язок, 1986. - 437 с.

ДОДАТОК А

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
д.т.н., проф. О. Н. Романюк
"____ "_____2019 р.

Технічне завдання
на магістерську кваліфікаційну роботу «Розробка методів та засобів підвищення рівня захисту інформаційних систем»
за спеціальністю
121 – Інженерія програмного забезпечення

Керівник магістерської кваліфікаційної роботи:
_____ к.т.н., доц. О.В.Романюк
"____"_____2019 р.
Виконав:
_____студент гр.2ПІ-18м І.С. Микитюк
"____"_____2019 р.

Вінниця – 2019 року

1. Найменування та галузь застосування

Магістерська кваліфікаційна робота: «Розробка методів та засобів підвищення рівня захисту інформаційних систем».

Галузь застосування – системи керування базами даних.

2. Підстава для розробки.

Підставою для виконання магістерської кваліфікаційної роботи (МКР) є індивідуальне завдання на МКР та наказ №____ ректора по ВНТУ про закріплення тем МКР.

3. Мета та призначення розробки.

Метою роботи є підвищення рівня захищеності інформації у сучасних інформаційних системах за рахунок розробки моделі багатошарового доступу до функцій СКБД та методу відслідковування несанкціонованих дій у СКБД.

3 Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись МКР.

1. Зрюмов, Е. А. Базы данных для инженеров: навчальний посібник / Е. А. Зрюмов, А. Г. Зрюмова; Алт. держ. техн. ун-т им. И. И. Ползунова. – Барнаул : Видав-во АлтГТУ, 2010. – 131 с.
2. Сучасні криптографічні системи: Навч. посібник. – Одеса: ВЦ ОНАЗ ім. О.С. Попова, 2007. – 152 стор.
3. Шайтанова Н. Ж., Туленгалиева М.Г. Защита информации в базах данных [Електронний ресурс]. Режим доступу: URL : http://www.rusnauka.com/10_DN_2014/Informatica/3_165120.doc.htm – Назва з екрану.
4. Каплун В.А., Дудатьєв А.В., Семеренко В.П., Захист програмного забезпечення, частина 1 – Вінниця, ВНТУ, 2005 – 140 с..

4. Технічні вимоги

Операційна система – Windows, MacOS;

Мова програмування – JavaScript;

Серверний рушій – NodeJS;

Середовище розробки – Visual Studio Code.

5. Конструктивні вимоги.

Конструкція пристрою повинна відповідати естетичним та ергономічним вимогам, повинна бути зручною в обслуговуванні та керуванні.

Графічна та текстова документація повинна відповідати діючим стандартам України.

6. Перелік технічної документації, що пред'являється по закінченню робіт:

– пояснювальна записка до МКР;

– технічне завдання;

– лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз предметної галузі та порівняльна характеристика сучасних СКБД	04.09.2019 – 30.09.2019
2	Проектування багатошарового захисту	30.09.2019 – 27.10.2019
3	Програмна реалізація методів та засобівв захисту	28.10.2019 – 10.11.2019
4	Тестування розробленого програмного засобу	11.11.2019 – 15.11.2019
5	Економічна частина	16.11.2019 – 21.11.2019

10. Порядок контролю та прийняття.

Виконання етапів магістерської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи.

Прийняття магістерської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком

ДОДАТОК Б

Лістинг програми

Файл auth.controller.ts

```
import {Body, Controller, ForbiddenException, Get, Post, Request, UseGuards} from '@nestjs/common';
import {AuthGuard} from '@nestjs/passport';
import {SignInDto} from '../dto/auth/sign.in.dto';
import {IAuthRequest} from '../interfaces/auth/auth.request.interface';
import {IAuthResponse} from '../interfaces/auth/auth.response.interface';
import {IUser} from '../interfaces/user.interface';
import {SessionsService} from '../providers/sessions.service';

@Controller('auth')
export class AuthController {
    public constructor(private readonly sessionsService: SessionsService) {}

    @Post('signIn')
    public async signIn(@Body() body: SignInDto): Promise<IAuthResponse> {
        const {login, password, visualKey} = body;
        if (password !== undefined) {
            return this.sessionsService.signInByPassword({ login, password });
        } else if (visualKey !== undefined) {
            return this.sessionsService.signInByVisualKey({ login, visualKey });
        } else {
            throw new ForbiddenException('No password nor visual key provided');
        }
    }

    @Get('user')
    @UseGuards(AuthGuard('jwt'))
    public async getUser(@Request() request: IAuthRequest): Promise<IUser> {
        const {session, login, name, role} = request.user;
        return {session, login, name, role};
    }

    @Post('logOut')
    @UseGuards(AuthGuard('jwt'))
    public async logOut(@Request() request: IAuthRequest): Promise<void> {
        await this.sessionsService.logOut(request.user);
    }

    @Post('token')
    public async refresh(@Body('token') token: string): Promise<IAuthResponse> {
        return this.sessionsService.refresh(token);
    }
}
```

Файл public.data.controller.ts

```
import {
    Body,
    Controller,
    Get,
    HttpException,
    InternalServerErrorException,
    Param,
    Post,
    UseGuards,
} from '@nestjs/common';
import {AuthGuard} from '@nestjs/passport';
import {AlterTableDto} from '../dto/public.data/alter.table.dto';
import {DropTableDto} from '../dto/public.data/drop.table.dto';
import {InsertRowDto} from '../dto/public.data/insert.row.dto';
import {SchemaTableDto} from '../dto/public.data/schema.table.dto';
import {SelectRowDto} from '../dto/public.data/select.row.dto';
import {SelectTableDto} from '../dto/public.data/select.table.dto';
import {UpdateRowDto} from '../dto/public.data/update.row.dto';
import {
    CanAlterTablesGuard,
    CanCreateTablesGuard,
    CanDropTablesGuard,
    CanModifyDataGuard,
    CanReadDataGuard,
} from '../guards/role.guards';
import {IDatabaseSchema} from '../interfaces/public.data/schema.table.interface';
import {PublicDataProvider} from '../providers/public.data.provider';

@Controller('data')
export class PublicDataController {
    public constructor(private readonly publicDataProvider: PublicDataProvider) {}

    @Get('schema')
    @UseGuards(AuthGuard('jwt'))
    public getSchema(): Promise<IDatabaseSchema> {
        return this.publicDataProvider.schema;
    }

    @Post('select')
    @UseGuards(AuthGuard('jwt'), CanReadDataGuard)
    public selectTable(@Body() dto: SelectTableDto): Promise<any> {
        return this.publicDataProvider
            .selectTable(dto.tableName, dto.orderBy || [])
            .catch((exception) => {
                if (exception instanceof HttpException) {
                    throw exception;
                } else {
                    throw new InternalServerErrorException(exception.message);
                }
            })
    }


    @Post('row')
    @UseGuards(AuthGuard('jwt'), CanReadDataGuard)
    public selectRow(@Body() dto: SelectRowDto): Promise<any> {
        return this.publicDataProvider
            .selectRow(dto.tableName, dto.filter)
            .catch((exception) => {
                if (exception instanceof HttpException) {
                    throw exception;
                } else {
                    throw new InternalServerErrorException(exception.message);
                }
            });
    }
```

```
@Post('update')
@UseGuards(AuthGuard('jwt'), CanModifyDataGuard)
public updateRow(@Body() dto: UpdateRowDto): Promise<any> {
    return this.publicDataProvider.updateRow(dto.tableName, dto.filter, dto.values)
        .catch((exception) => {
            if (exception instanceof HttpException) {
                throw exception;
            } else {
                throw new InternalServerErrorException(exception.message);
            }
        });
}

@Post('insert')
@UseGuards(AuthGuard('jwt'), CanModifyDataGuard)
public insertRow(@Body() dto: InsertRowDto): Promise<any> {
    return this.publicDataProvider.insertRow(dto.tableName, dto.values)
        .catch((exception) => {
            if (exception instanceof HttpException) {
                throw exception;
            } else {
                throw new InternalServerErrorException(exception.message);
            }
        });
}

@Post('delete')
@UseGuards(AuthGuard('jwt'), CanModifyDataGuard)
public deleteRow(@Body() dto: SelectRowDto): Promise<any> {
    return this.publicDataProvider.deleteRow(dto.tableName, dto.filter)
        .catch((exception) => {
            if (exception instanceof HttpException) {
                throw exception;
            } else {
                throw new InternalServerErrorException(exception.message);
            }
        });
}

@Post('truncate')
@UseGuards(AuthGuard('jwt'), CanModifyDataGuard)
public truncateTable(@Body() dto: SelectTableDto): Promise<any> {
    return this.publicDataProvider.truncateTable(dto.tableName)
        .catch((exception) => {
            if (exception instanceof HttpException) {
                throw exception;
            } else {
                throw new InternalServerErrorException(exception.message);
            }
        });
}

@Post('create')
@UseGuards(AuthGuard('jwt'), CanCreateTablesGuard)
public async createTable(@Body() dto: SchemaTableDto): Promise<any> {
    try {
        await this.publicDataProvider.createTable(dto);
        return await this.publicDataProvider.refreshSchema();
    } catch (exception) {
        if (exception instanceof HttpException) {
            throw exception;
        } else {
            throw new InternalServerErrorException(exception.message);
        }
    }
}
```

```
            @Post('drop')
            @UseGuards(AuthGuard('jwt'), CanDropTablesGuard)
            public async dropTable(@Body() dto: DropTableDto): Promise<any> {
                try {
                    await this.publicDataProvider.dropTable(dto.tableName);
                    return await this.publicDataProvider.refreshSchema();
                } catch (exception) {
                    if (exception instanceof HttpException) {
                        throw exception;
                    } else {
                        throw new InternalServerErrorException(exception.message);
                    }
                }
            }

            @Post('alter')
            @UseGuards(AuthGuard('jwt'), CanAlterTablesGuard)
            public async alterTable(@Body() dto: AlterTableDto): Promise<any> {
                try {
                    await this.publicDataProvider.alterTable(dto.tableName, dto.drop || [], dto.add
|| []);
                    return await this.publicDataProvider.refreshSchema();
                } catch (exception) {
                    if (exception instanceof HttpException) {
                        throw exception;
                    } else {
                        throw new InternalServerErrorException(exception.message);
                    }
                }
            }
}
```

Файл role.gurds.ts

```
        import {CanActivate, ExecutionContext, Injectable} from '@nestjs/common';
        import {ACCESS_RIGHT} from '../enum/e.access.right';
        import {IAuthRequest} from '../interfaces/auth/auth.request.interface';
        import {all, map} from '../util/func.util';

        export class AccessGuard implements CanActivate {
            private readonly _rights: ACCESS_RIGHT[];

            public constructor(rights: ACCESS_RIGHT[]) {
                this._rights = rights;
            }

            public canActivate(this: AccessGuard, context: ExecutionContext): boolean {
                const request = context.switchToHttp().getRequest() as IAuthRequest;
                return request && request.user && all(map((x) => request.user.role.rights[x],
this._rights));
            }
        }

        @Injectable()
        export class CanUseVisualKeysGuard extends AccessGuard {
            public constructor() {
                super([ACCESS_RIGHT.CAN_USE_VISUAL_KEY]);
            }
        }

        @Injectable()
        export class CanUseEncryptedTokenGuard extends AccessGuard {
            public constructor() {
                super([ACCESS_RIGHT.CAN_USE_ENCRYPTED_TOKEN]);
            }
        }
```

```
@Injectable()
export class CanCreateTablesGuard extends AccessGuard {
    public constructor() {
        super([ACCESS_RIGHT.CAN_CREATE_TABLES]);
    }
}

@Injectable()
export class CanDropTablesGuard extends AccessGuard {
    public constructor() {
        super([ACCESS_RIGHT.CAN_DROP_TABLES]);
    }
}

@Injectable()
export class CanAlterTablesGuard extends AccessGuard {
    public constructor() {
        super([ACCESS_RIGHT.CAN_ALTER_TABLES]);
    }
}

@Injectable()
export class CanModifyDataGuard extends AccessGuard {
    public constructor() {
        super([ACCESS_RIGHT.CAN_DELETE, ACCESS_RIGHT.CAN_INSERT, ACCESS_RIGHT.CAN_UPDATE]);
    }
}

@Injectable()
export class CanReadDataGuard extends AccessGuard {
    public constructor() {
        super([ACCESS_RIGHT.CAN_SELECT]);
    }
}
```

Файл connection.provider.ts

```
import {Injectable} from '@nestjs/common';
import {Connection, ConnectionOptions, createConnection, DefaultNamingStrategy} from 'typeorm';
import {EncryptedTokenEntity} from '../orm/entity/encrypted.token.entity';
import {RoleAccessEntity} from '../orm/entity/role.access.entity';
import {SchemaColumnEntity} from '../orm/entity/schema.column.entity';
import {SchemaConstraintColumnUsageEntity} from '../orm/entity/schema.constraint.column.usage.entity';
import {SchemaTableConstraintEntity} from '../orm/entity/schema.table.constraint.entity';
import {SchemaTableEntity} from '../orm/entity/schema.table.entity';
import {SessionEntity} from '../orm/entity/session.entity';
import {UserEntity} from '../orm/entity/user.entity';
import {UserRoleEntity} from '../orm/entity/user.role.entity';
import {VisualKeyEntity} from '../orm/entity/visual.key.entity';

function stringifyArgs(...args: any[]): string {
    return args.map((arg) => JSON.stringify(arg)).join(', ');
}

function underscoreCase(name: string): string {
    return name
        .replace(/\.?([A-Z]+)/g, (x,y) => "_" + y.toLowerCase())
        .replace(/^_/, "");
}

class UnderscoreNamingStrategy extends DefaultNamingStrategy {
    public name: string;

    public constructor(name: string) {
        super();
        this.name = name;
    }
```

```
            columnName(propertyName: string, customName: string, embeddedPrefixes: string[]): string
{
                // console.info(`columnName(${stringifyArgs(...arguments)})`);
                return  super.columnName(propertyName,  customName  ||  underscoreCase(propertyName),
embeddedPrefixes);
            }

            tableName(targetName: string, userSpecifiedName: string | undefined): string {
                // console.info(`tableName(${stringifyArgs(...arguments)})`);
                return super.tableName(targetName, userSpecifiedName || underscoreCase(targetName));
            }

            joinColumnName(relationName: string, referencedColumnName: string): string {
                // console.info(`joinColumnName(${stringifyArgs(...arguments)})`);
                return underscoreCase(relationName) + '_' + underscoreCase(referencedColumnName);
            }
        }

        @Injectable()
        export class ConnectionProvider {
            private _connection?: Promise<Connection>;

            public readonly connectionOptions = {
                    type: 'postgres',
                    host: '127.0.0.1',
                    port: 5432,
                    username: 'postgres',
                    database: 'test',
                    synchronize: true,
                    entities: [
                        EncryptedTokenEntity,
                        RoleAccessEntity,
                        SessionEntity,
                        UserEntity,
                        UserRoleEntity,
                        VisualKeyEntity,

                        SchemaTableEntity,
                        SchemaColumnEntity,
                        SchemaConstraintColumnUsageEntity,
                        SchemaTableConstraintEntity,
                    ],
                    namingStrategy: new UnderscoreNamingStrategy('underscore'),
            } as ConnectionOptions;

            public constructor() {
                this.connection.catch((error) => {
                    console.error(error);
                });
            }

            public get connection(this: ConnectionProvider): Promise<Connection> {
                if (!this._connection) {
                    this._connection = createConnection(this.connectionOptions);
                }
                return this._connection;
            }
}
```

Файл public.data.provider.ts

```
        import {BadRequestException, Injectable, InternalServerErrorException} from '@nestjs/common';
        import {HttpException} from '@nestjs/common/exceptions/http.exception';
        import {ColumnFilterDto} from '../dto/public.data/column.filter.dto';
        import {OrderByDto} from '../dto/public.data/select.table.dto';
        import {DATA_TYPE} from '../enum/e.data.type';
        import          {IDatabaseSchema,          ISchemaColumn,          ISchemaTable}          from
'../interfaces/public.data/schema.table.interface';
```

```
        import {SchemaTableEntity} from '../orm/entity/schema.table.entity';
        import {all, map} from '../util/func.util';
        import {defineColumn, escapeIdentifier, escapeLiteral, escapeSet, escapeValue, escapeWhere}
from '../util/query.util';
        import {ConnectionProvider} from './connection.provider';

        @Injectable()
        export class PublicDataProvider {
            private _schema?: Promise<IDatabaseSchema>;

            private async _getSchema(this: PublicDataProvider): Promise<IDatabaseSchema> {
                const {connection} = this.connectionProvider;
                const tableRepo = (await connection).getRepository(SchemaTableEntity);
                //              const              constraintsRepo              =              (await
connection).getRepository(SchemaTableConstraintEntity);

                const tablesRaw = await tableRepo.createQueryBuilder('table')
                    .leftJoinAndSelect('table.columns', 'column')
                    .leftJoinAndSelect('table.constraints', 'constraint')
                    .leftJoinAndSelect('constraint.usedColumns', 'usedColumn')
                    .where(
                        'table.tableCatalog = :tableCatalog AND table.tableSchema = :tableSchema AND
table.tableType    =    :tableType    AND    (constraint.constraintType    =    :constraintType    OR
constraint.constraintType IS NULL)',
                        {
                            tableCatalog: this.databaseName,
                            tableSchema: this.schemaName,
                            tableType: 'BASE TABLE',
                            constraintType: 'PRIMARY KEY',
                        },
                    )
                    .orderBy('table.tableName', 'ASC')
                    .orderBy('column.ordinalPosition', 'ASC', 'NULLS FIRST')
                    .getMany();

                const result: IDatabaseSchema = {};
                for (let table of tablesRaw) {
                    const       constraint       =       table.constraints!.find((constraint)       =>
constraint.constraintType === 'PRIMARY KEY');
                    result[table.tableName] = {
                        tableName: table.tableName,
                        columns: table.columns!.map((column) => ({
                            defaultValue:  column.columnDefault  !==  null  ?  column.columnDefault  :
undefined,
                            valueNullable: column.isNullable === 'YES',
                            valueType: column.dataType,
                            maximumLength:     column.characterMaximumLength     !==     null     ?
column.characterMaximumLength : undefined,
                            columnName: column.columnName,
                        })),
                        primaryKey:    constraint    ?    constraint.usedColumns!.map((column)    =>
column.columnName) : [],
                    };
                }

                return result;
            }

            public readonly schemaName: string = 'public';

            public get databaseName(this: PublicDataProvider): string {
                return this.connectionProvider.connectionOptions.database as string;
            }

            public constructor(private readonly connectionProvider: ConnectionProvider) {

            }
```

```
            public get schema(this: PublicDataProvider): Promise<IDatabaseSchema> {
                if (!this._schema) {
                    this._schema = this._getSchema();
                }
                return this._schema;
            }

            public refreshSchema(this: PublicDataProvider): Promise<IDatabaseSchema> {
                return this._schema = this._getSchema();
            }

            public async selectTable(this: PublicDataProvider, tableName: string, orders:
OrderByDto[]): Promise<any> {
                const databaseSchema = await this.schema;
                const tableSchema = databaseSchema[tableName];
                if (tableSchema) {
                    for (let orderBy of orders) {
                        if (!tableSchema.columns.find((column) => column.columnName ===
orderBy.columnName)) {
                            throw new BadRequestException(`Invalid column
${escapeIdentifier(tableName)}.${escapeIdentifier(orderBy.columnName)}`);
                        }
                    }
                    const ordering = orders.length
                        ? 'ORDER BY ' + orders.map((orderBy) =>
`${escapeIdentifier(orderBy.columnName)} ${orderBy.order || 'ASC'} NULLS ${orderBy.nulls ||
'FIRST'}`).join(', ')
                        : '';
                    const connection = await this.connectionProvider.connection;
                    return await connection.query(`SELECT * FROM
public.${escapeIdentifier(tableName)} ${ordering}`);
                } else {
                    throw new BadRequestException(`Invalid table ${escapeIdentifier(tableName)}`);
                }
            }

            public async truncateTable(this: PublicDataProvider, tableName: string): Promise<void> {
                const databaseSchema = await this.schema;
                const tableSchema = databaseSchema[tableName];
                if (tableSchema) {
                    const connection = await this.connectionProvider.connection;
                    await connection.query(`TRUNCATE public.${escapeIdentifier(tableName)}`);
                } else {
                    throw new BadRequestException(`Invalid table ${escapeIdentifier(tableName)}`);
                }
            }

            public async selectRow(this: PublicDataProvider, tableName: string, filter:
ColumnFilterDto[]): Promise<any> {
                const databaseSchema = await this.schema;
                const tableSchema = databaseSchema[tableName];
                if (tableSchema) {
                    const primaryKey = new Set(tableSchema.primaryKey);
                    const filterColumns = new Set(filter.map((column) => column.columnName));
                    for (let key of primaryKey) {
                        if (!filterColumns.has(key)) {
                            throw new BadRequestException(`Filter for column
${escapeIdentifier(tableName)}.${escapeIdentifier(key)} is not specified`);
                        }
                    }
                    for (let key of filterColumns) {
                        if (!primaryKey.has(key)) {
                            throw new BadRequestException(`Invalid column
${escapeIdentifier(tableName)}.${escapeIdentifier(key)} specified`);
                        }
                    }
```

```
                    const connection = await this.connectionProvider.connection;
                    const filterQuery = filter
                        .map((filterColumn)          =>          escapeWhere(filterColumn.columnName,
filterColumn.columnValue))
                        .join(' AND ');
                    const      data      =      await      connection.query(`SELECT      *      FROM
public.${escapeIdentifier(tableName)} WHERE ${filterQuery ? filterQuery : 'true'}`) as any[];
                    switch (data.length) {
                    case 0:
                        return null;
                    case 1:
                        return data[0];
                    default:
                        throw new BadRequestException('Data is not unique!');
                    }
                } else {
                    throw new BadRequestException(`Invalid table ${escapeIdentifier(tableName)}`);
                }
            }

            public   async   deleteRow(this:   PublicDataProvider,   tableName:   string,   filter:
ColumnFilterDto[]): Promise<void> {
                const databaseSchema = await this.schema;
                const tableSchema = databaseSchema[tableName];
                if (tableSchema) {
                    const primaryKey = new Set(tableSchema.primaryKey);
                    const filterColumns = new Set(filter.map((column) => column.columnName));
                    for (let key of primaryKey) {
                        if (!filterColumns.has(key)) {
                            throw       new       BadRequestException(`Filter       for       column
${escapeIdentifier(tableName)}.${escapeIdentifier(key)} is not specified`);
                        }
                    }
                    for (let key of filterColumns) {
                        if (!primaryKey.has(key)) {
                            throw          new          BadRequestException(`Invalid          column
${escapeIdentifier(tableName)}.${escapeIdentifier(key)} specified`);
                        }
                    }
                    const connection = await this.connectionProvider.connection;
                    const filterQuery = filter
                        .map((filterColumn)          =>          escapeWhere(filterColumn.columnName,
filterColumn.columnValue))
                        .join(' AND ');
                    await  connection.query(`DELETE  FROM  public.${escapeIdentifier(tableName)}  WHERE
${filterQuery ? filterQuery : 'true'}`);
                } else {
                    throw new BadRequestException(`Invalid table ${escapeIdentifier(tableName)}`);
                }
            }

            public   async   updateRow(this:   PublicDataProvider,   tableName:   string,   filter:
ColumnFilterDto[], values: ColumnFilterDto[]): Promise<any> {
                const databaseSchema = await this.schema;
                const tableSchema = databaseSchema[tableName];
                if (tableSchema) {
                    const primaryKey = new Set(tableSchema.primaryKey);
                    const filterColumns = new Set(filter.map((column) => column.columnName));
                    const    expectedColumns    =    new    Set(tableSchema.columns.map((column)    =>
column.columnName));
                    for (let key of primaryKey) {
                        if (!filterColumns.has(key)) {
                            throw       new       BadRequestException(`Filter       for       column
${escapeIdentifier(tableName)}.${escapeIdentifier(key)} is not specified!`);
                        }
                    }
                    for (let key of filterColumns) {
```

```
                    if (!primaryKey.has(key)) {
                        throw new BadRequestException(`Invalid column ${escapeIdentifier(tableName)}.${escapeIdentifier(key)} specified!`);
                    }
                }
                for (let column of values) {
                    if (!expectedColumns.has(column.columnName)) {
                        throw new BadRequestException(`Unexpected column ${escapeIdentifier(tableName)}.${escapeIdentifier(column.columnName)}`);
                    }
                }
                const connection = await this.connectionProvider.connection;
                const filterQuery = filter
                    .map((filterColumn) => escapeWhere(filterColumn.columnName, filterColumn.columnValue))
                    .join(' AND ');

                const updateQuery = values
                    .map((column) => escapeSet(column.columnName, column.columnValue))
                    .join(', ');

                const data = await connection.query(`UPDATE public.${escapeIdentifier(tableName)} SET ${updateQuery} WHERE ${filterQuery ? filterQuery : 'true'} RETURNING *`) as any[];
                switch (data.length) {
                case 0:
                    return null;
                case 1:
                    return data[0];
                default:
                    throw new BadRequestException('Data is not unique!');
                }
            } else {
                throw new BadRequestException(`Invalid table ${escapeIdentifier(tableName)}`);
            }
        }

        public async insertRow(this: PublicDataProvider, tableName: string, values: ColumnFilterDto[]): Promise<any> {
            const databaseSchema = await this.schema;
            const tableSchema = databaseSchema[tableName];
            if (tableSchema) {
                const expectedColumns = new Set(tableSchema.columns.map((column) => column.columnName));
                const columns = new Set(values.map((column) => column.columnName));
                for (let column of values) {
                    if (!expectedColumns.has(column.columnName)) {
                        throw new BadRequestException(`Unexpected column ${escapeIdentifier(tableName)}.${escapeIdentifier(column.columnName)}`);
                    }
                }
                for (let column of tableSchema.columns) {
                    if (!column.valueNullable && column.defaultValue === undefined && !columns.has(column.columnName)) {
                        throw new BadRequestException(`Column ${escapeIdentifier(tableName)}.${escapeIdentifier(column.columnName)} is required!`);
                    }
                }
                const connection = await this.connectionProvider.connection;

                const columnNames = values.map((column) => escapeIdentifier(column.columnName)).join(', ');
                const columnValues = values.map((column) => escapeValue(column.columnValue)).join(', ');

                const data = await connection.query(`INSERT INTO public.${escapeIdentifier(tableName)} (${columnNames}) VALUES (${columnValues}) RETURNING *`) as any[];
```

```
                    switch (data.length) {
                    case 0:
                        return null;
                    case 1:
                        return data[0];
                    default:
                        throw new BadRequestException('Data is not unique!');
                    }
                } else {
                    throw new BadRequestException(`Invalid table ${escapeIdentifier(tableName)}`);
                }
            }

            public async createTable(this: PublicDataProvider, tableSchema: ISchemaTable): Promise<void> {
                const databaseSchema = await this.schema;
                const existingTableSchema = databaseSchema[tableSchema.tableName];
                if (!existingTableSchema) {
                    const primaryKeyColumns = new Set(tableSchema.primaryKey);
                    const columns = tableSchema.columns
                        .map(
                            (columnDefinition) =>
                                defineColumn(columnDefinition, primaryKeyColumns.has(columnDefinition.columnName)),
                        );
                    await (
                        (await this.connectionProvider.connection)
                            .query(`CREATE TABLE ${escapeIdentifier(tableSchema.tableName)} (${columns.join(', ')});`)
                    );
                } else {
                    throw new BadRequestException(`Table ${escapeIdentifier(tableSchema.tableName)} is already exists`);
                }
            }

            public async dropTable(this: PublicDataProvider, tableName: string): Promise<void> {
                const databaseSchema = await this.schema;
                const existingTableSchema = databaseSchema[tableName];
                if (existingTableSchema) {
                    await (
                        (await this.connectionProvider.connection)
                            .query(`DROP TABLE ${escapeIdentifier(tableName)};`)
                    );
                } else {
                    throw new BadRequestException(`Table ${escapeIdentifier(tableName)} does not exist`);
                }
            }

            public async alterTable(this: PublicDataProvider, tableName: string, dropColumns: string[], addColumns: ISchemaColumn[]): Promise<void> {
                const databaseSchema = await this.schema;
                const existingTableSchema = databaseSchema[tableName];
                if (existingTableSchema) {
                    const primaryKeyColumns = new Set(existingTableSchema.primaryKey);
                    const existingColumns = new Set(existingTableSchema.columns.map((columnDefinition) => columnDefinition.columnName));

                    const actions: string[] = [];

                    actions.push(
                        ...dropColumns.map((columnName) => {
                            if (primaryKeyColumns.has(columnName)) {
                                throw new BadRequestException(`Can't drop primary key column ${escapeIdentifier(tableName)}.${escapeIdentifier(columnName)}`);
                            }
```

```
                            if (existingColumns.has(columnName)) {
                                return `DROP COLUMN ${escapeIdentifier(columnName)}`;
                            } else {
                                throw                 new                 BadRequestException(`Column
${escapeIdentifier(tableName)}.${escapeIdentifier(columnName)} does not exist`);
                            }
                        }),
                    );

                    actions.push(
                        ...addColumns.map((columnDefinition) => {
                            if (!existingColumns.has(columnDefinition.columnName)) {
                                if (columnDefinition.defaultValue === undefined) {
                                    switch (columnDefinition.valueType) {
                                    case DATA_TYPE.SERIAL:
                                    case DATA_TYPE.UUID:
                                        break;
                                    default:
                                        throw             new             BadRequestException(`Column
${escapeIdentifier(tableName)}.${escapeIdentifier(columnDefinition.columnName)} must be nullable or
have default value`);
                                    }
                                }
                                return `ADD COLUMN ${defineColumn(columnDefinition, false)}`;
                            } else {
                                throw                 new                 BadRequestException(`Column
${escapeIdentifier(tableName)}.${escapeIdentifier(columnDefinition.columnName)} already exists`)
                            }
                        }),
                    );

                    if (actions.length) {
                        await (
                            (await this.connectionProvider.connection)
                                .query(`ALTER TABLE ${escapeIdentifier(tableName)} ${actions.join(',
')};`)
                        );
                    }
                } else {
                    throw  new  BadRequestException(`Table  ${escapeIdentifier(tableName)}  does  not
exist`);
                }
            }
}
```

Файл session.manager.ts

```
        import {SelectQueryBuilder} from 'typeorm';
        import * as uuid from 'uuid';
        import {Injectable, UnauthorizedException} from '@nestjs/common';
        import {ACCESS_RIGHT} from '../enum/e.access.right';
        import {IJWTAccessPayload, IJWTRefreshPayload} from '../interfaces/jwt.payload.interface';
        import {IUser} from '../interfaces/user.interface';
        import {IUserRole} from '../interfaces/user.role.interface';
        import {SessionEntity} from '../orm/entity/session.entity';
        import {UserEntity} from '../orm/entity/user.entity';
        import {isProductionEnv} from '../util/env.util';
        import {ConnectionProvider} from './connection.provider';

        @Injectable()
        export class SessionsManager {
            public static get SessionTimeOut() {
                return isProductionEnv() ? (2 * 60 * 60 * 1000) : (10 * 60 * 60 * 1000);
            }

            public constructor(private readonly connectionManager: ConnectionProvider) {}

            private              async              _selectQuery(this:              SessionsManager):
Promise<SelectQueryBuilder<SessionEntity>> {
```

```
            const {connection} = this.connectionManager;
            const repo = (await connection).getRepository(SessionEntity);
            return repo.createQueryBuilder('session')
                .innerJoinAndSelect('session.user', 'user')
                .innerJoinAndSelect('user.role', 'role')
                .leftJoinAndSelect('role.rights', 'right');
        }

        public async get(id: string): Promise<SessionEntity> {
            const sessions = await (await this._selectQuery())
                .where('session.uuid = :session AND session.expires > NOW()', { session: id })
                .getMany();
            switch (sessions.length) {
            case 0:
                throw new UnauthorizedException('Invalid session');
            case 1:
                return sessions[0];
            default:
                throw new UnauthorizedException('Too many sessions found!');
            }
        }

        public async buildPayload(this: SessionsManager, session: SessionEntity): Promise<IUser>
{
            const role: IUserRole = {
                id: session.user.role.id,
                name: session.user.role.name,
                rights: {},
            };
            for (let key in ACCESS_RIGHT) {
                const name = ACCESS_RIGHT[key] as ACCESS_RIGHT;
                role.rights[name] = session.user.role.defaultPolicy;
            }
            for (let right of session.user.role.rights || []) {
                role.rights[right.policy] = right.value;
            }
            return {
                session: session.uuid,
                login: session.user.login,
                name: session.user.name,
                role: role,
            };
        }

        public    async    validate(this:    SessionsManager,    payload:    IJWTAccessPayload):
Promise<SessionEntity> {
            const sessions = await (await this._selectQuery())
                .where(
                    'session.uuid = :session AND user.login = :login AND session.accessKey =
:accessKey AND session.expires > NOW()',
                    payload,
                )
                .getMany();
            switch (sessions.length) {
            case 0:
                throw new UnauthorizedException('Invalid user login and/or token');
            case 1:
                if (sessions[0].accessKey === payload.accessKey) {
                    return sessions[0];
                } else {
                    throw new UnauthorizedException('Hacked!');
                }
            default:
                throw new UnauthorizedException('Too many sessions found!');
            }
        }
```

```
            public    async    refresh(this:    SessionsManager,    payload:    IJWTRefreshPayload):
Promise<SessionEntity> {
                const {connection} = this.connectionManager;
                const repo = (await connection).getRepository(SessionEntity);
                const sessions = await (await this._selectQuery())
                    .where('session.uuid = :session AND user.login = :login AND session.expires >
NOW()', payload)
                    .getMany();
                switch (sessions.length) {
                case 0:
                    throw new UnauthorizedException('Invalid user login and/or token');
                case 1:
                    if (sessions[0].refreshKey === payload.refreshKey) {
                        return await repo.save(
                            Object.assign(
                                sessions[0],
                                {
                                    accessKey: uuid.v4(),
                                    refreshKey: uuid.v4(),
                                    expires: new Date(Date.now() + SessionsManager.SessionTimeOut),
                                } as Partial<SessionEntity>,
                            )
                        );
                    } else {
                        throw new UnauthorizedException('Hacked!');
                    }
                default:
                    throw new UnauthorizedException('Too many sessions found!');
                }
            }

            public async create(this: SessionsManager, user: UserEntity): Promise<SessionEntity> {
                const {connection} = this.connectionManager;
                const repo = (await connection).getRepository(SessionEntity);
                const session = repo.create({
                    accessKey: uuid.v4(),
                    refreshKey: uuid.v4(),
                    user: user,
                    expires: new Date(Date.now() + SessionsManager.SessionTimeOut),
                });
                return await repo.save(session);
            }

            public async drop(this: SessionsManager, session: SessionEntity): Promise<void> {
                const {connection} = this.connectionManager;
                const repo = (await connection).getRepository(SessionEntity);
                Object.assign(
                    session,
                    {
                        expires: new Date(),
                    } as Partial<SessionEntity>,
                );
                await repo.save(session);
            }
}
```

Файл users.manager.ts

```
        import * as crypto from 'crypto';

        import {ForbiddenException, Injectable, UnauthorizedException} from '@nestjs/common';
        import {ACCESS_RIGHT} from '../enum/e.access.right';
        import {IUser} from '../interfaces/user.interface';
        import {IClearUserToken, IEncryptedUserToken} from '../interfaces/auth/user.token.interface';
        import {EncryptedTokenEntity} from '../orm/entity/encrypted.token.entity';
        import {UserEntity} from '../orm/entity/user.entity';
        import {ConnectionProvider} from './connection.provider';

        @Injectable()
```

```
        export class UsersManager {
            public constructor(private readonly connectionManager: ConnectionProvider) {}

            public async getByPassword(login: string, password: string): Promise<UserEntity> {
                const {connection} = this.connectionManager;
                const repo = (await connection).getRepository(UserEntity);
                const entity = await repo.findOne({ login, password }, { relations: ['role',
'role.rights' ] });
                if (entity !== undefined) {
                    return entity;
                } else {
                    throw new ForbiddenException('Invalid login and/or password');
                }
            }

            public async getByVisualKey(login: string, visualKey: string): Promise<UserEntity> {
                const {connection} = this.connectionManager;
                const repo = (await connection).getRepository(UserEntity);
                const users = await repo.createQueryBuilder('user')
                    .innerJoinAndSelect('user.role', 'role')
                    .innerJoinAndSelect('role.rights', 'right')
                    .innerJoinAndSelect('user.visualKeys', 'key')
                    .where('user.login = :login AND key.value = :visualKey', { login, visualKey })
                    .getMany();
                switch (users.length) {
                case 0:
                    throw new ForbiddenException('Invalid login and/or visual key');
                case 1:
                {
                    const user = users[0];
                    const canUseVisualKeyRight = (user.role.rights || []).find((right) =>
right.policy === ACCESS_RIGHT.CAN_USE_VISUAL_KEY);
                    if (canUseVisualKeyRight ? canUseVisualKeyRight.value : user.role.defaultPolicy)
{
                        return user;
                    } else {
                        throw new ForbiddenException('Can not use visual keys');
                    }
                }
                default:
                    throw new ForbiddenException('Too many users found');
                }
            }

            public async getByEncryptedToken(encryptedToken: IEncryptedUserToken):
Promise<UserEntity> {
                const {connection} = this.connectionManager;
                const tokensRepo = (await connection).getRepository(EncryptedTokenEntity);
                const usersRepo = (await connection).getRepository(UserEntity);
                const tokenEntity = await tokensRepo.findOne(encryptedToken.uuid);
                if (tokenEntity) {
                    const {key, created} = tokenEntity;
                    const clearToken = JSON.parse(crypto.publicDecrypt({ key },
Buffer.from(encryptedToken.payload, 'hex')).toString('utf8')) as IClearUserToken;
                    if (clearToken.created === created.valueOf()) {
                        const user = await usersRepo.findOne(clearToken.login, { relations: ['role',
'role.rights' ]});
                        if (user) {
                            const canUseTokens = (user.role.rights || []).find((right) =>
right.policy === ACCESS_RIGHT.CAN_USE_ENCRYPTED_TOKEN);
                            if (canUseTokens ? canUseTokens.value : user.role.defaultPolicy) {
                                return user;
                            } else {
                                throw new ForbiddenException('Can not use encrypted tokens');
                            }
                        }
                    }
```

```
                }
                throw new ForbiddenException('Invalid token');
            }
}
```

ДОДАТОК В

ІЛЮСТРАТИВНИЙ МАТЕРІАЛ ДО ЗАХИСТУ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Завідувач кафедри ПЗ, д. т. н., професор _____ О. Н. Романюк

Науковий керівник, к.т.н., доц. кафедри ПЗ_____О. В. Романюк

Рецензент, д.т.н, проф. кафедри КН_____ А. А. Яровий

Нормоконтроль, к.т.н., доц. кафедри ПЗ_____О. В. Романюк

Виконавець, студент групи 2ПІ-18м _____ І. С. Микитюк

Магістерська кваліфікаційна робота на тему:

«Розробка методів та засобів захисту інформаційних систем»

Виконав: **Микитюк І.С.**
студент групи 2ПІ-18м
Керівник: **Романюк О.В.**
к.т.н., доц каф. ПІ

1

Основні проблеми захисту баз даних

Загроза	Пояснення
Надмірні дозволи	Надання більшості користувачів привілеїв за замовчуванням, або надання ролей, які мають доступ до функціональних можливостей, які є непотрібними
Слабкі паролі	Слабкі паролі або паролі за замовчуванням
Старі оновлення безпеки	Нестача оновлень безпеки, або використання старих версій програмного забезпечення
Слабо конфігурований аудит	Слабе налаштування відносно реєстрації користувацьких операцій чи аудиту у зв'язку з чим виникають труднощі з виявленням можливих втрат інформації чи зламів
Ім'я облікових записів за замовчуванням	Бази даних містять облікові записи за замовчуванням, що дозволяє використати атаки brute-force чи підбору паролю.

2

Актуальність

Необхідність створення нових способів захисту інформації у сучасних інформаційних системах. Впровадження нових технологій у області захисту інформації у базах даних.

Мета

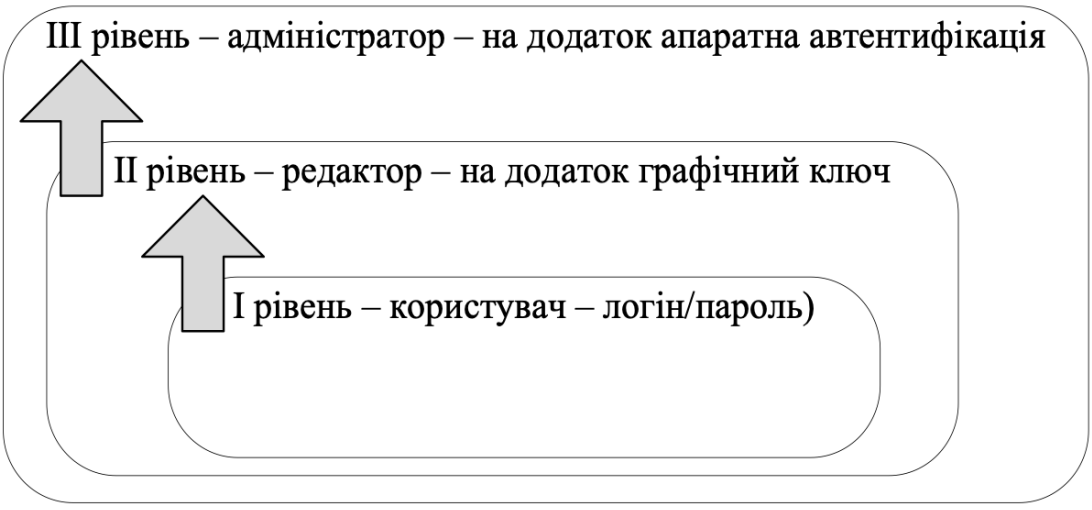
Метою роботи є підвищення рівня захищеності інформації у сучасних інформаційних системах за рахунок розробки моделі багатошарового доступу до функцій СКБД

Задачі

- Аналіз проблем сучасних систем керування базами даних;
- Розробка моделі багатошарового доступу до функцій СКБД;
- Реалізація програмного засобу і аналіз результатів.

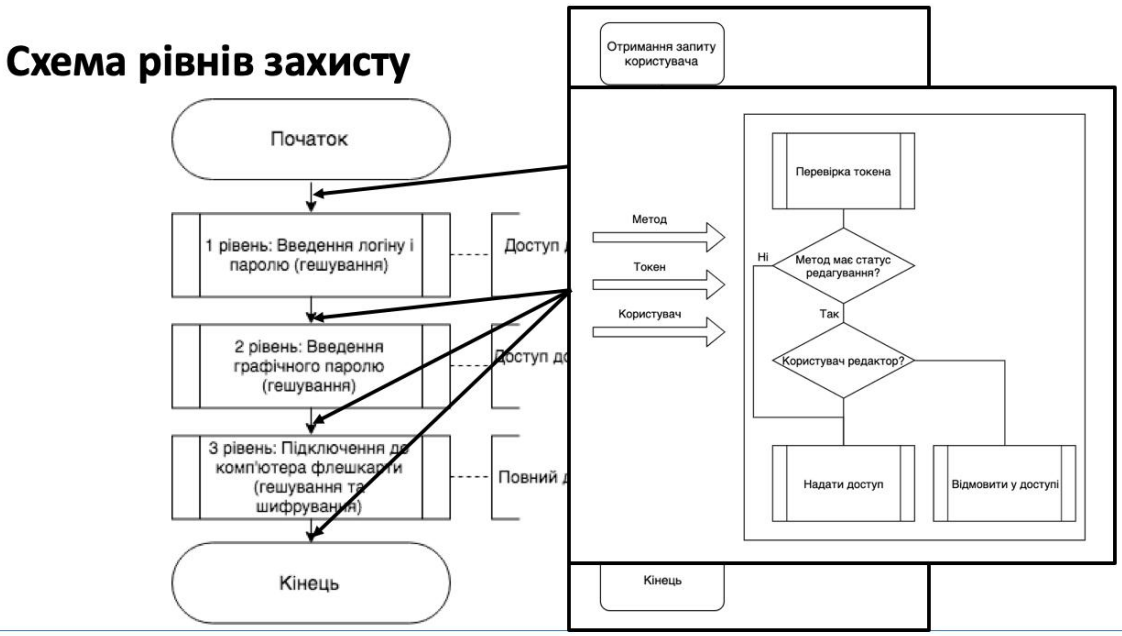
3

Рівні багатошарового захисту інформації



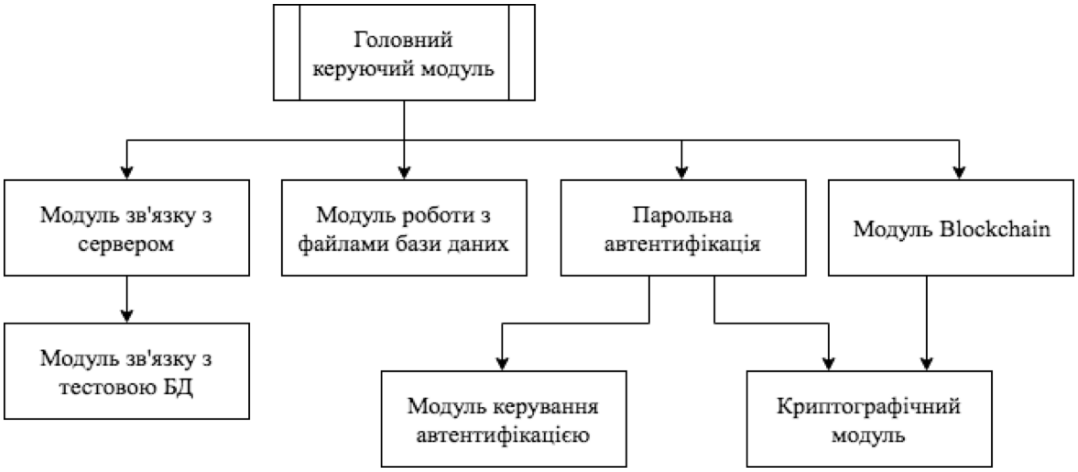
4

Схема рівнів захисту



5

Схема зв’язку модулів програмного засобу



6

Вигляд користувацького простору імен

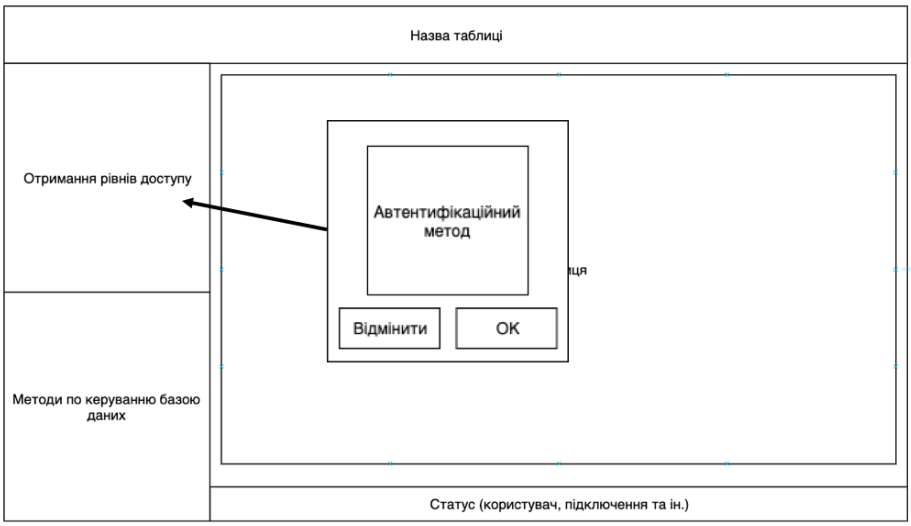
7

Відслідковування несанкціонованих змін

№ п.п.	Ключ 1	Ключ 2	Ключ 3	Ключ 4	Ключ 5
1	Значення 1	Значення 2	Текст 3	Значення 4	Значення 5
Геш-зн.	w7dlR	Piysb	*PUGfh*	FGxDO	xVb20
Попереднє геш-зн.	0	w7dlR	Piysb	SPpz0	FGxDO

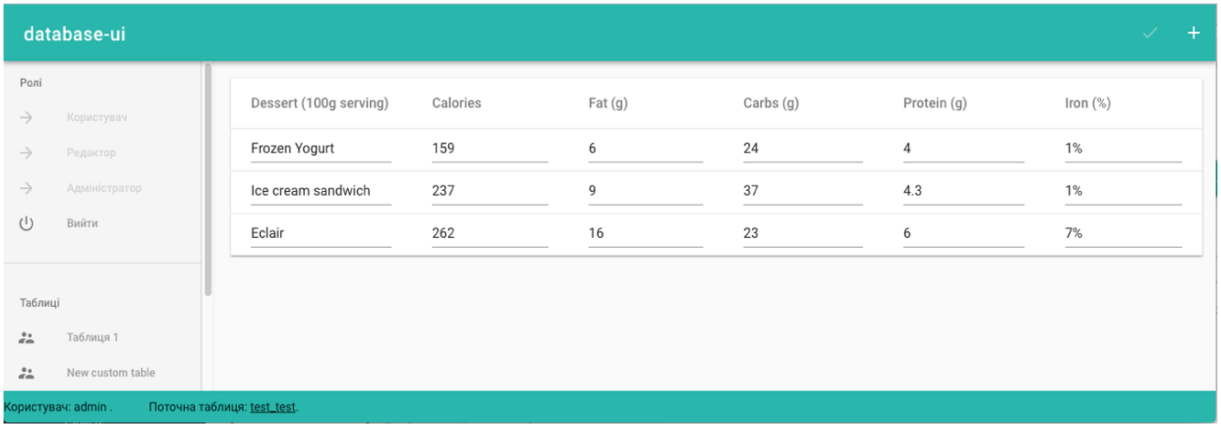
8

Графічна схема інтерфейсу



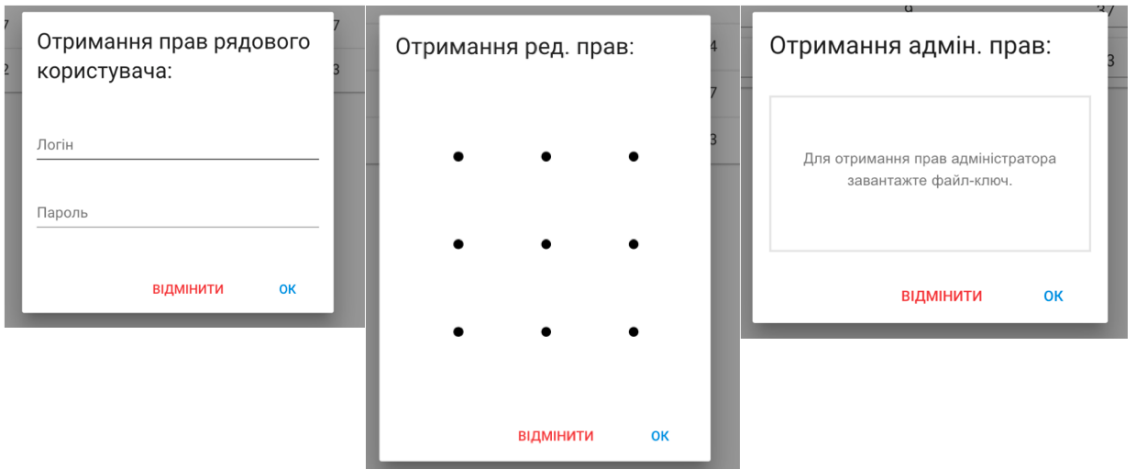
9

Вигляд основного вікна програми



10

Вигляд кроків автентифікації користувача



11

Наукова новизна:

- Вперше запропоновано модель розшарування доступу до інформації, що зберігається в сучасних інформаційних системах, особливістю якого є розмежування користувацького доступу до різних рівнів керування базою даних, що дозволило підвищити рівень захищеності даних та функцій сучасних СКБД.
- Подальшого розвитку отримав метод виявлення несанкціонованих дій при роботі з базою даних, який на відмінну від інших дозволяє виявити несанкціоновані дії зловмисників чи користувачів без належного рівня доступу.

Практичне призначення:

- було розроблено програмний засіб, що містить в собі модель багатошарового доступу до різних рівнів функціоналу в сучасних СКБД;
- розроблено програмну систему відслідковування несанкціонованого доступу до методів СКБД.

12

Апробація

Результати магістерської кваліфікаційної роботи були обговорені на двох конференціях:

- Науково-технічна конференція підрозділів Вінницького національного технічного університету. Факультет інформаційних технологій та комп'ютерної інженерії. Вінниця, 2018
- Міжнародна науково-практична конференція "Інформаційні технології та комп’ютерне моделювання", Івано-Франківськ, 2018
- Опубліковані тези та матеріали конференції.

13

Дякую за увагу!

Виконав: **Микитюк І.С.**
студент групи 2ПІ-18м
Керівник: **Романюк О.В.**
к.т.н., доц каф. ПІ

14