Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна
Навчально-науковий інститут комп’ютерних наук та штучного інтелекту
Кафедра комп’ютерних систем та робототехніки

«Затверджую»
в.о. завідуючого кафедри
комп’ютерних систем та робототехніки
_____ к. ф.-м. н., доцент Максим Хруслов
«___» червня 2025 р.

Пояснювальна записка

до кваліфікаційної роботи
бакалавра

на тему: «**СИСТЕМА АВТОМАТИЗОВАНОГО УПРАВЛІННЯ РЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ ДЛЯ ВИРОБНИЧИХ ПІДПРИЄМСТВ**»

Спеціальність 151 – Автоматизація та комп'ютерно-інтегровані технології
Галузь знань 15 – Автоматизація та приладобудування
Освітня програма «Автоматизація та комп’ютерно-інтегровані технології»

Захищено на засіданні
Екзаменаційної комісії № 46
протокол № __ від __.06.2025 р.
Оцінка _____ / _____
Голова Екзаменаційної комісії
_____ **ЧУГАЙ А.М.**

Виконав:
Студент групи КУ– 41
БОДНЯ Нікіта Вадимович_____

Керівник: доцент кафедри комп'ютерних систем та робототехніки, к.т.н.
БУЛАВІН Дмитро Олексійович_____

Рецензент: .т.н., доцент, професор кафедри теоретичної та прикладної інформатики
РУККАС Кирило Маркович_____

Харків – 2025

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи бакалавра складається з вступу, трьох розділів, висновків до них, списку використаних джерел і чотирьох додатків. Загальний обсяг роботи становить 72 сторінки, з яких 45 сторінок основної частини містять 46 рисунків, 13 найменувань списку використаних джерел та 4 додатки.

Об'єкт дослідження – процес автоматизованого управління реляційними базами даних у виробничих підприємствах.

Предмет дослідження – архітектурні підходи та технологічні рішення для побудови систем автоматизованого управління реляційними базами даних.

Мета дослідження – автоматизація процесів управління даними в реляційних базах даних на виробничих підприємствах.

Проблема, яку вирішує кваліфікаційна робота – відсутність інтегрованого програмного рішення для централізованого адміністрування реляційних баз даних у середовищі виробничих підприємств.

Область застосування – інформаційні системи управління виробничими процесами, ERP-модулі, рішення для адміністрування корпоративних баз даних на промислових підприємствах.

Ключові слова: реляційна база даних, автоматизоване управління, виробниче підприємство, архітектура сервісів, PostgreSQL, HTML-шаблони, JavaScript-модулі.

ABSTRACT

The explanatory note to the bachelor's qualification work consists of an introduction, three sections, conclusions to them, a list of sources used and four appendices. The total volume of the work is 72 pages, of which 45 pages of the main part contain 46 pictures, 13 names of the list of sources used and 4 appendices.

The object of the study is the process of automated management of relational databases in manufacturing enterprises.

The subject of the study is architectural approaches and technological solutions for building automated management systems of relational databases.

The purpose of the study is to automate data management processes in relational databases at manufacturing enterprises.

The problem that the qualification work solves is the lack of an integrated software solution for centralized administration of relational databases in the environment of manufacturing enterprises.

Scope of application is information systems for managing production processes, ERP modules, solutions for administering corporate databases at industrial enterprises.

Keywords: relational database, automated management, manufacturing enterprise , service architecture, PostgreSQL, HTML templates, JavaScript modules.

ЗМІСТ

ВСТУП

У сучасних умовах інтенсифікація виробничих процесів особливого значення набуває ефективне та надійне управління корпоративними даними. РБД(Реляційна база даних) залишаються основним сховищем інформації про технологічні процеси, ресурси та результати виробництва. Водночас обсяг інформації, що генерується на підприємствах, постійно збільшується, а вимоги до безперебійності доступу, швидкості обробки й можливості оперативного адміністрування стають все вимогливішими. Через те є актуальним створення інтегрованого програмного рішення, яке дозволяє автоматизувати повсякденні операції адміністрування РБД з мінімальним втручанням адміністратора, підвищити стійкість, продуктивність і конкурентоспроможність виробничих підприємств. Реалізація такої системи дозволяє:

1) Зменшити час реакції на відмови та пікові навантаження.
2) Скоротити операційні витрати завдяки мінімізації ручного втручання.
3) Підвищити якість даних шляхом централізованої валідації та контролю цілісності.
4) Прискорити аналітику та прийняття рішень завдяки єдиному інтерфейсу для виконання складних SQL-запитів, інтеграції з ERP.

Метою роботи є розробка системи автоматизованого управління реляційними базами даних для виробничих підприємств, що забезпечує централізоване адміністрування. Для досягнення цієї мети поставлено такі завдання:

1) Розробити концептуальну модель та архітектуру.
2) Реалізувати ERP-модулі на основі сучасних технологій.
3) Забезпечити графічний інтерфейс для віддаленого налаштування та керування через веб-додаток.

У роботі використано методи проєктування архітектури, автоматизації адміністративних задач з застосуванням скриптових рішень, методологічні підходи, викладені Л. С. Глобою в підручнику «Розробка інформаційних

ресурсів та систем» [10]. Наукова складова полягає у інтеграції ключових операцій адміністрування РБД в єдину платформу, що дає змогу зменшити час реагування на відмови та знизити людський фактор. Практичне значення системи полягає в її можливості швидкого впровадження на виробничих підприємствах.

РОЗДІЛ 1.
ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ІНСТРУМЕНТІВ

1.1. Актуальність автоматизації адміністрування реляційних баз даних у виробничих підприємствах.

У сучасному виробництві обсяг і швидкість генерації даних зчитування контролерів, сенсорів та систем управління поступово зростають. РБД залишаються основним методом зберігання таких даних, адже вони забезпечують цілісність записів та мають гнучкі запити SQL. Водночас класичні методи адміністрування стають не лише трудомісткими, а й затримують реакцію на критичні події.

Для виробничих підприємств вихід із ладу системи здатен призвести до значних фінансових втрат. Вимоги до безперервності роботи та швидкості обробки запитів поступово зростають: потрібно в цілодобовому режимі забезпечувати доступ до даних, синхронізувати інформацію між різними відділами виробничого процесу. Зростання навантаження породжує важливі виклики:

1) Ручні процедури адміністрування стають неефективними: традиційними задачами адміністратора БД(База даних) є налаштування резервних копій, моніторинг, застосування патчів, запуск скриптів, де будь-яка помилка в конфігурації може призвести до втрат даних.
2) Підвищені вимоги до збереження доступності: для виробничих підприємств недоступність інформаційної системи на деякий час тягне за собою зупинку виробництва, через що підприємство зазнає збитків. Забезпечення цілодобового доступу вимагає автоматизованих механізмів failover та перевірки стану кожного вузла, які складно виконувати вручну.
3) Гнучкість масштабування та адаптація до змін навантаження: виробничі цикли часто мають пікові навантаження, яких можна запобігти автоматичним розгортанням додаткових ресурсів, підключенням нових

реплік або масштабуванням параметрів. Інакше планувати і виконувати скрипти доводиться вручну, що займає деякий час та потребує специфічних знань.

4) Економічна ефективність: введення автоматизованих інструментів адміністрування приводить до зменшення витрат на підтримку інфраструктури, зменшення часу простоїв і оптимізації використання ресурсів.

Отже, автоматизація адміністрування РБД у промисловому середовищі є важливою: вона підвищує стійкість систем, знижує витрати та ризики, забезпечує оперативне реагування на зміни навантаження й відмови і сприяє інтеграції інфраструктури в більші цифрові платформи. Реалізація автоматизованих процесів є необхідністю для сучасних підприємств, що ставлять за мету безперебійність та ефективність виробництва.

1.2. Роль реляційних СУБД у корпоративних інформаційних системах: забезпечення зберігання та обробки виробничих даних

У сучасних виробничих підприємствах корпоративні ІС(Інформаційна система), що призначені для комплексної автоматизації всіх видів господарської діяльності підприємств [13], охоплюють декілька рівнів керування: від взаємодії з обладнанням до планування ресурсів і логістики. РСУБД(Реляційна система управління базами даних) є основою таких ІС через забезпечення надійного зберігання та ефективної обробки даних, що необхідні для виконання операційних і аналітичних завдань.

РСУБД реалізують модель ACID, що є дуже важливим для виробничих процесів. Атомарність гарантує, що операції повинні виконуватися у повному обсязі або не виконуватися взагалі. Узгодженість гарантує неможливість створення суперечливих записів. Ізоляція дозволяє багатьом користувачам та сервісам одночасно здійснювати операції без впливу один на одного. Довговічність забезпечується через журналювання, що зберігає всі зміни [4].

Реляційна модель даних є гарним вибором для відображення складних сутностей виробництва: товарних груп, замовлень, маршрутів виконання

операцій, персоналу. Використання таблиць зі зв’язками «багато-до-багатьох» дозволяє гарно відображати, які ресурси задіяні в кожному промисловому циклі, а зовнішні ключі – підтримувати ієрархічні залежності та запобігати неправильним операціям видалення чи оновлення даних [12].

РСУБД підтримують складні SQL-запити з використанням підзапитів та аналітичних вікон, що дозволяє отримувати оперативні звіти про промислові показники без залучення окремих бізнес-аналітичних систем. Індексація забезпечує прискорений доступ за ключовими полями.

Сучасні ERP-модулі взаємодіють із РСУБД через інтерфейси по типу REST API. Це дозволяє подавати заявки на резервування матеріалів децентралізовано, оновлювати статус замовлень, вести облік браку та витрат у режимі реального часу. Завдяки такій інтеграції відбувається обмін даними між рівнями оперативного управління та верхньою ланкою бізнес-аналізу, що забезпечує узгодженість інформації й мінімізує затримки в інформаційних потоках.

РСУБД підтримують горизонтальне та вертикальне масштабування. Горизонтальне масштабування дозволяє розподілити таблиці за ключовими ознаками між кількома вузлами. Вертикальне масштабування оптимізує продуктивність окремих інстансів.

РСУБД надають потужні механізми авторизації та аутентифікації. Можливості шифрування даних забезпечують захист виробничих параметрів і комерційної таємниці. Тригери та журнали доступу можна налаштувати так, щоб фіксувати будь-які спроби несанкціонованого доступу.

Таким чином, реляційні СУБД є незамінною основою ІС у виробничому середовищі, поєднуючи надійне зберігання, потужну обробку операцій та можливості складної аналітики.

1.3. Сучасні підходи та інструменти для автоматизації адміністрування РБД

У зв’язку зі швидким зростанням обсягів даних та складністю корпоративних систем, адміністрування РСУБД покладається не на ручні процедури, а на автоматизовані платформи й фреймворки. Основними підходами та інструментами, що використовуються у промисловому середовищі є:

1) Infrastructure as Code: його ідею полягає в декларативному описі конфігурації серверів та СУБД. Прикладом інструменту є Terraform, що дозволяє описатувати кластери БД, правила мережевого доступу, а далі використати цю інфраструктуру в хмарі чи локальному дата-центрі.
2) Висока доступність та аварійне відновлення: інструменти, такі як Patroni або Galera Cluster, забезпечують постійну роботу при збої одного з вузлів.
3) Автоматизовані рішення для бекапів і відновлення: сервіси, на зразок pgBackRest, Barman або MyDumper/MyLoader, реалізують диференційні та інкрементальні бекапи, швидке відновлення даних та перевірку контрольних сум.
4) Хмарні сервіси баз даних: вони пропонують автоматизовані патчі, бекапи, реплікацію та моніторинг, що допомагає адміністратору у виконанні більшості рутинних операцій.
5) Комп’ютеризовані системи керування підприємствами та виробництвами: вони надають функціональні можливості реєстрації та координації усіх виробничо-господарських процесів в реальному часі [9].

1.4. Безпека та захист даних у РСУБД

Розмежування прав доступу в РСУБД виконується на трьох основних рівнях – схема, таблиця та рядок. Конфігурація цих механізмів дозволяє забезпечити, що користувачі бачитимуть і модифікуватимуть тільки ті дані, на які вони мають дозвіл.

Для забезпечення інформаційної безпеки та можливості відстежувати критичні події в системі необхідно розробити механізми авторизації, управління доступом на основі посад, аудиту на рівні СУБД [11]. Аудит досягається обов'язковими полями у таблицях для зберігання часу створення та останнього оновлення інформації, а також використання тригерів, які автоматично фіксують важливі операції.

Однією з найпоширеніших загроз для РСУБД є SQL-ін'єкції – атаки, під час яких стороння людина вставляє шкідливі фрагменти SQL-коду у вхідні поля, що призводить до несанкціонованого доступу або видалення даних [6]. Замість того, щоб динамічно формувати SQL-рядок шляхом конкатенації змінних та рядків, потрібно використовувати фіксований шаблон із заповнювачами, а значення підставляти окремо. Це гарантує, що СУБД розрізняє статичну частину запиту й дані, ігноруючи шкідливі вставки. Підготовлені вирази дозволяють СУБД аналізувати й компілювати план виконання запиту один раз, а потім виконувати його багаторазово з різними параметрами. Це не лише підвищує безпеку, а й покращує продуктивність при повторних запитах.

Висновки за розділом 1

У першому розділі було розглянуто ключові теоретичні моменти та обґрунтовано необхідність автоматизації адміністрування РБД у виробничих підприємствах. Аналіз показав, що ручні методи опрацювання, моніторингу та оновлення СУБД не відповідають сучасним вимогам швидкого реагування та доступності в умовах постійного збільшення обсягів даних. Автоматизація цих операцій забезпечує:

1) Високу надійність та безпеку: централізоване резервне копіювання, контроль цілісності даних та їх відновлення.
2) Безперебійність роботи: автоматизовані механізми failover та перевірка станів вузлів, що забезпечує мінімальні простої.

3) Гнучке масштабування: стрімке розгортання нових реплік і налаштування параметрів без попереднього втручання забезпечує стабільну роботу в пікові навантаження.
4) Економію ресурсів: зниження операційних затрат за рахунок автоматизації рутинних процесів і концентрація персоналу на аналітичних та оптимізаційних завданнях.

Розглянута роль РСУБД у ІС підтвердила їхню необхідність для зберігання та обробки виробничих даних, завдяки реалізації ACID, підтримці складних зв’язків та потужній аналітиці SQL. Інтеграція з ERP через інтерфейси REST API забезпечує оперативну взаємодію між системами.

Таким чином, результати розділу 1 закладають міцну теоретичну основу для подальшого проєктування та реалізації системи автоматизованого управління реляційних баз даних у виробничому середовищі.

РОЗДІЛ 2.
ПРОЄКТУВАННЯ СИСТЕМИ АВТОМАТИЗОВАНОГО УПРАВЛІННЯ РБД

2.1. Функціональні та нефункціональні вимоги

У цьому підрозділі формалізуються загальні вимоги системи автоматизованого управління РБД, починаючи з базових процедур безпеки та закінчуючи сценаріями роботи з сутностями виробничого процесу. Серед вимог є наступні:

1) Вимоги до автентифікації та авторизації: система повинна забезпечувати надійну перевірку користувача та коректний розподіл прав доступу. Після переходу на сторінку входу перед користувачем з'являтиметься проста форма з полями «Email» та «Password», при введенні неправильних даних система виводитиме повідомлення про невірно введену пошту користувача або пароль.
2) Вимоги до CRUD-операцій користувачів і посад: Після успішного входу в систему з правами адміністратора користувач отримуватиме доступ до розділу «Recruit and hire employees», де будуть реалізовані чотири основні операції: створення, читання, оновлення та видалення. Інтерфейс починатиметься зі списку наявних користувачів: таблиця має колонки «First Name», «Last Name», «Role», «Email» та «Password». Для пошуку по обсягу даних буде передбачено фільтри за «First Name», «Last Name», «Role» та «Email». При натисканні на кнопку «Add Employee», перед чим потрібно ввести обов’язкові поля «First Name», «Last Name» та «Email», а також обрати відповідну йому посаду у списку «Role», адміністратор додаватиме нового користувача за допомогою POST-запиту на /api/users, і у разі успіху новий користувач з’являтиметься в табличному списку зі стандартним паролем, який можна буде змінити при першій спробі входу до системи. Щоб відредагувати профіль користувача, потрібно натиснути кнопку «Edit» в рядку. Система

завантажуватиме дані через GET /api/users/:id і заповнюватиме форму. Адміністратор зможе змінити всі поля, що були обов’язковими для додавання цього користувача, та пароль. При підтвердженні змін виконуватиметься PUT /api/users/:id з новими даними. Якщо адміністратору потрібно видалити користувача із системи, потрібно буде натиснути кнопку «Delete» в рядку, що відкриватиме модальне вікно із питанням підтвердження. Після підтвердження виконуватиметься DELETE /api/users/:id. Аналогічно буде реалізовано CRUD-операції для посад: у розділі «Roles Table» відображатиметься перелік існуючих посад та їх обов'язків. Додавання нової посади вимагатиме введення унікальної назви та списку обов'язків. Редагування посади через PUT /api/roles/:id дозволятиме змінити набір обов'язків, а видалення відбуватиметься через DELETE /api/roles/:id. Таким чином, вимоги до CRUD-операцій гарантують гнучке управління обліковими записами та посадами в системі, дотримання принципів безпеки та наявність повного журналу дій для аудиту.

3) Вимоги до обробки виробничих замовлень: Після автентифікації в системі користувач із відповідними правами переходитиме до розділу «Production Orders», де йому буде доступний весь список замовлень у вигляді таблиці з колонками «Product», «Quantity», «Status», «Responsible» та «Control Responsible». Для створення нового замовлення на створення продукту користувач обиратиме зі списків продукт для виготовлення, його кількість, відповідальну особу за виготовлення та відповідальну особу за контроль якості та натискатиме кнопку «Start Production». Система перевірятиме коректність введених даних, після чого виконуватиметься POST /api/production-orders, і замовлення з’являтиметься у списку, також воно з'являтиметься у відповідальної особи за виготовлення з розрахунком матеріалів, що потрібні для виготовлення всього замовлення, та полем для змінення статусу замовлення із опціями «Preparing», «In Progress», «Rejected» та

«Completed». При виставленні статусу «Completed», замовлення з'являтиметься у відповідальної особи за контроль якості з додатковими полями для встановлення якості виготовленої продукції, полем для нотаток та списком статусів з опціями «Rejected» та «Approved». Після встановлення статусу «Approved», замовлення заноситиметься в історію з усіма виставленими до неї параметрами. Редагування замовлень виконуватиметься самим замовником, натиснувши на кнопку «Edit» в рядку, де у модальному вікні можна буде змінювати всі параметри замовлення. Видалення замовлення виконуватиметься за допомогою кнопки кнопку «Delete» /api/production-orders/:id в рядку. Такий підхід до CRUD-операцій із виробничими замовленнями забезпечує коректність даних та прозорість процесу.

4) Вимоги до управління матеріалами та інвентарем: Обов'язок «Monitor warehouse inventory» має бути доступним для користувачів, що відповідальні за постачання та склад. Після присвоєння цього обов'язку система відображатиме дві вкладки: «Current Warehouse Inventory» та «Current Finished Goods Inventory». Матеріали та продукти з'являтимуться у відповідних вкладках при створенні їх у виробничому відділі.
5) Вимоги до закупівель та постачальників: У системі передбачатиметься обширне управління постачальниками та обробку замовлень на закупівлю, що забезпечуватиме своєчасне поповнення матеріалами. Після входу користувача із обов'язком «Procure raw materials» переводитиме у відповідний розділ, де будуть реалізовані CRUD-операції для постачальників та замовлень закупівель.
6) Нефункціональні вимоги: При розробці системи важливо не лише втілити необхідний функціонал, а й забезпечити такі властивості, які гарантуватимуть безперервну та ефективну роботу в реальному промисловому середовищі. Відповіді на CRUD-операції повинні швидко оброблятися, для цього в server.js використовуватиметься пул

підключень до PostgreSQL через pg.Pool, що дозволятиме повторно використовувати з’єднання й оминати затримки при створенні нових конекторів. Аутентифікація відбуватиметься через безпечні HTTP-only cookies із прапорцем Secure, що робить неможливим крадіжку сесійних токенів зі сторонніх скриптів. Також будуть реалізовані валідація довжини пароля та перевірка формату email в auth_server.js. У головному API під час завантаження перевірятиметься посада користувача, тому захищені маршрути будуть недоступні користувачам без відповідного обов'язку. Архітектуру можна буде масштабувати горизонтально, завдяки пулу та безстанному характеру HTTP-запитів можна розгортати кілька реплік сервісів. Таким чином, дані нефункціональні вимоги забезпечують, що система буде швидкою, безпечною та здатною зростати разом із ростом виробничого навантаження [8].

2.2. Загальна архітектура системи

1) Поділ на два сервіси: В архітектурі системи повинні бути чітко відокремлені два сервіси, кожен з яких відповідатиме за власну доменну область і виконуватиме чітко окреслені завдання. Перший сервіс буде сфокусований на обробці автентифікації та встановлені нового пароля. При запуску він створюватиме Express-додаток, що має порт 3001. В основному він відображатиме сторінку логіну через app.get('/') з формою для входу, оброблятиме POST-запит на /login, буде реалізувати функцію скидання пароля через GET+POST на /reset-password та переноситиме користувача на URL основного API-сервісу після успішного введення логіну та паролю. Завдяки ізольованому автентифікаційному сервісу мінімізуватимуться ризики витоку логіки авторизації та можна буде незалежно масштабувати або замінювати механізм автентифікації без впливу на основні бізнес-роути. Основний сервіс відповідатиме за всю бізнес-логіку роботи з виробничими даними, CRUD-операціями, управління продуктами, матеріалами, замовленнями, бюджетами і таке інше. Він ініціалізуватиме власний Express-додаток на порту 3002,

підключатиме пул pg.Pool для взаємодії з PostgreSQL, визначатиме численні маршрути для роботи з відділами, перевірятиме наявність прав доступу до конкретних обов'язків. Цей поділ гарантує, що основне API концентруватиметься на бізнес-процесах.

2) Схема взаємодії (див. рис. 2.1):
    - Користувач – Браузер: Користувач вводить у адресний рядок URL сервісу автентифікації та бачить форму для автентифікації.
    - Браузер – Сервіс автентифікації: Після введення пошти та паролю браузер відправляє POST-запит до /login. Сервер перевіряє введені дані та у разі успіху створює сесію та встановлює HTTP-only cookie з ідентифікатором сесії та Виконує res.redirect('/dashboard'), передаючи cookie для подальших запитів.
    - Браузер – Основний сервіс: На цьому етапі браузер звертається до основного сервісу, завантажуючи статичний HTML і JS.
    - Основний сервіс – PostgreSQL: Перед обробкою будь-якого запиту основний сервіс перевіряє наявність і валідність cookie, а також обов'язки, що присвоєні до посади користувача. Для операцій над даними основний сервіс звертається до бази даних через пул підключень pg.Pool.
    - Основний сервіс – Браузер: Браузер отримує відповіді і відображає дані в таблицях, модальних вікнах або графіках.

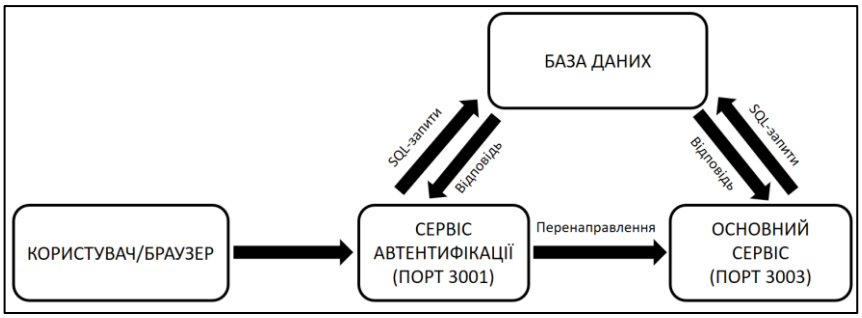


Рисунок 2.1 – Схема взаємодії.

2.3. Проєктування бази даних

У системі дані зберігатимуться в низці реляційних таблиць, кожна з яких відповідає за окремий бізнес-домен:

1) users і roles: users зберігатиме облікові записи користувачів, roles – міститиме назви та описи посад.
2) products, material_costs, warehouse_inventory: products буде довідником продуктів, які можна виготовляти, material_costs зберігатиме для кожного матеріалу його поточну вартість за одиницю, warehouse_inventory контролюватиме кількість матеріалів на складі.
3) production_orders, finished_goods_inventory: production_orders зберігатиме заявки на випуск продукції, finished_goods_inventory – облік готової продукції на складі.
4) purchase_requests, suppliers, purchase_history: suppliers буде довідником постачальників із контактною інформацією та відповідальною особою, purchase_requests – довідником заявок на закупівлю матеріалів, purchase_history – архівом завершених замовлень, що переміщуватимуться з purchase_requests після зміни статусу на Completed.
5) sales_orders, completed_sales: sales_orders міститиме заявки від клієнтів, completed_sales – історію виконаних продажів з точним часом та фінансовими деталями, що переноситиметься із sales_orders при їхньому завершенні.

6) clients, competitors, campaigns: clients матиме інформацію про клієнтів, а саме назву, контактні дані, competitors буде довідником конкурентів із їхніми характеристиками, campaigns зберігатиме маркетингові кампанії, у яких фігурують продукти, дати початку, дати завершення та бюджет.
7) budget_items, budgets: budgets матиме загальні бюджетні плани, budget_items – деталі бюджетів, які пов'язуються із конкретними записами в budgets.
8) forecast_scenarios, forecast_entries, workers_performance: forecast_scenarios і forecast_entries використовуватимуться для збереження сценаріїв прогнозування та отриманих значень показників, workers_performance – фіксуватиме продуктивність працівників.

Висновки за розділом 2

У другому розділі деталізовано весь процес проєктування системи автоматизованого управління РБД у промисловому середовищі від формулювання вимог до структури бази даних. Спочатку було окреслено функціональні вимоги, надійний вхід та скидання пароля, CRUD-операції над користувачами, ролями, виробничими замовленнями, матеріалами, інвентарем, закупівлями та постачальниками. Кожен сценарій має API-маршрут, механізм валідації та обробку помилок, що забезпечує коректність даних та прозорість бізнес-процесів.

Також було зазначено нефункціональні вимоги, які забезпечують стабільність та масштабованість системи: доступність у безперервному режимі, безпеку через HTTP-only cookies, а також горизонтальне масштабування.

У загальній архітектурі системи було обґрунтовано поділ на два незалежні сервіси: сервіс автентифікації, що буде сконцентрований на аутентифікації, встановленні сесій та скиданні пароля, та основний сервіс, який опрацьовуватиме всі бізнес-роути та звертається до PostgreSQL через pg.Pool. Між сервісами також буде реалізовано послідовне перенаправлення: після успішного входу сервіс автентифікації переноситиме користувача на

основний сервіс, де скрипти завантажуватимуть дані й динамічно оновлювати інтерфейс. Статичні HTML, CSS і JS-файли віддаватимуться через єдиний express.static із папки public, що спрощує розгортання й кешування.

На кінець, у розділі про проєктування БД наведено перелік основних таблиць: користувачі й посади, продукти й матеріали, виробничі та складські об’єкти, заявки на закупівлю та історію продажів, бюджетні плани й маркетингові кампанії, аналітичні сценарії та показники продуктивності працівників. Така модель забезпечує цілісність і гнучкість даних, зберігати історію змін за допомогою окремих архівних таблиць.

Усі прийняті рішення у сукупності формують модульну, стійку та масштабовану систему, вона дозволяє ізолювати механізми безпеки від бізнес-логіки, забезпечує прозорість і контроль усіх операцій, а архітектура двох сервісів спрощує подальше розширення та підтримку.

РОЗДІЛ 3.
РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ СИСТЕМИ

3.1. Реалізація сервісу автентифікації

Вихідний код знаходиться у файлі auth_server.js, на початку якого імпортуються необхідні пакети та визначаються потрібні константи (див. рис. 3.1).

```
const express : e | () => core.Express  = require('express');
const bodyParser : {...} | {...}  = require('body-parser');
const { Pool } = require('pg');
const app : any | Express  = express();
const PORT : number  = 3001;
const pool = new Pool({
    user: 'postgres',
    host: 'localhost',
    database: 'company_data',
    password: 'kolokol2',
    port: 5432
});
const STANDARD_PASSWORD : string  = '12345678';
```

Рисунок 3.1 – Лістинг імпорту пакетів та налаштування з'єднання з БД сервісу автентифікації.

1) Express – веб-фреймворк для створення HTTP-сервісів [3].
2) body-parser використовується для зручного отримання даних, які приходять у тілі HTTP-запитів [2].
3) pg.Pool – пул підключень до PostgreSQL [7].
4) константа PORT задає номер порту, на якому буде працювати сервіс.
5) константа STANDARD_PASSWORD задає початковий пароль для новостворених користувачів.

Далі використовуються посередники (див. рис. 3.2).

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static( root: 'public'));
app.use(express.json());
```

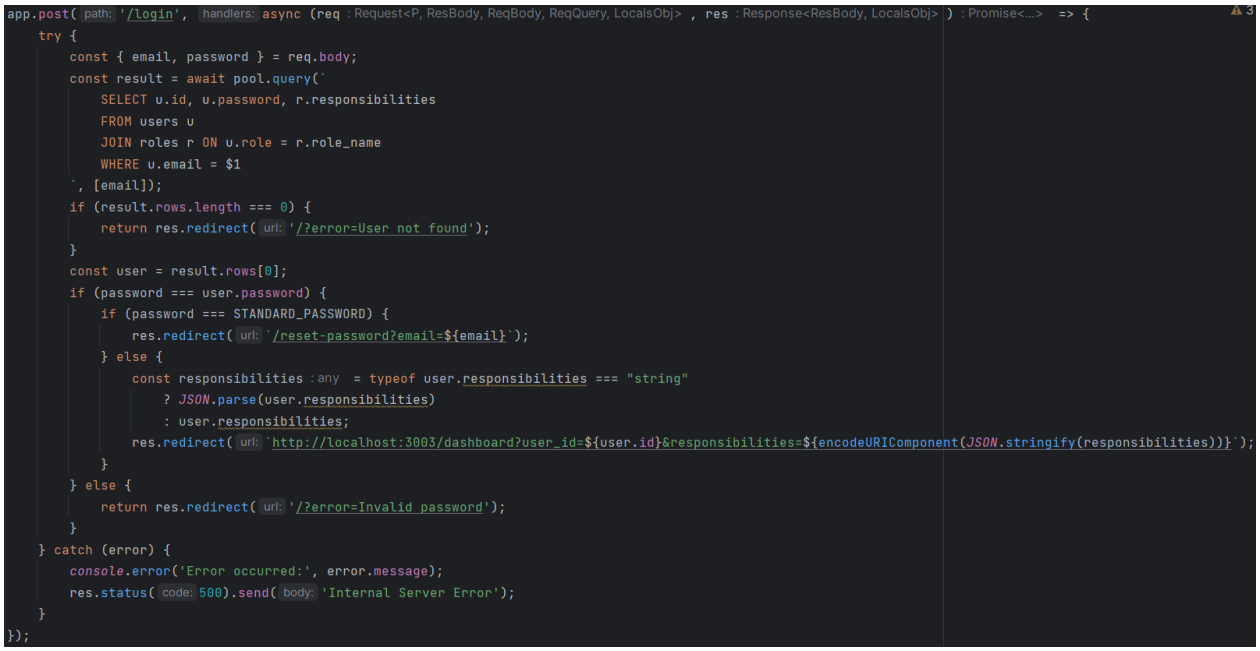
Рисунок 3.2 – Лістинг використання посередників.

1) bodyParser.urlencoded({ extended: true }) робить доступним req.body.email і req.body.password із форм.
2) express.static('public') віддає HTML/CSS/JS для сторінок логіну і ресету.
3) express.json() дозволяє приймати JSON у тілі запитів.

Основними елементами у коді є маршрути логіну, а саме GET / (див. рис. 3.3), у якому відображається статична сторінка автентифікації з полями email та password, та POST /login (див. рис. 3.4), який перевіряє внесені у ці поля облікові дані, шукає користувача в таблиці користувачів та посад через JOIN, а також, якщо пароль є стандартним, перенаправляє користувача на сторінку встановлення нового паролю через маршрут GET /reset-password (див. рис. 3.5), валідація та оновлення в БД якого відбувається за допомогою маршруту POST /update-password (див. рис. 3.6). При встановленні нового пароля присутня перевірка його довжини, що повинна бути від 16 до 35 символів.

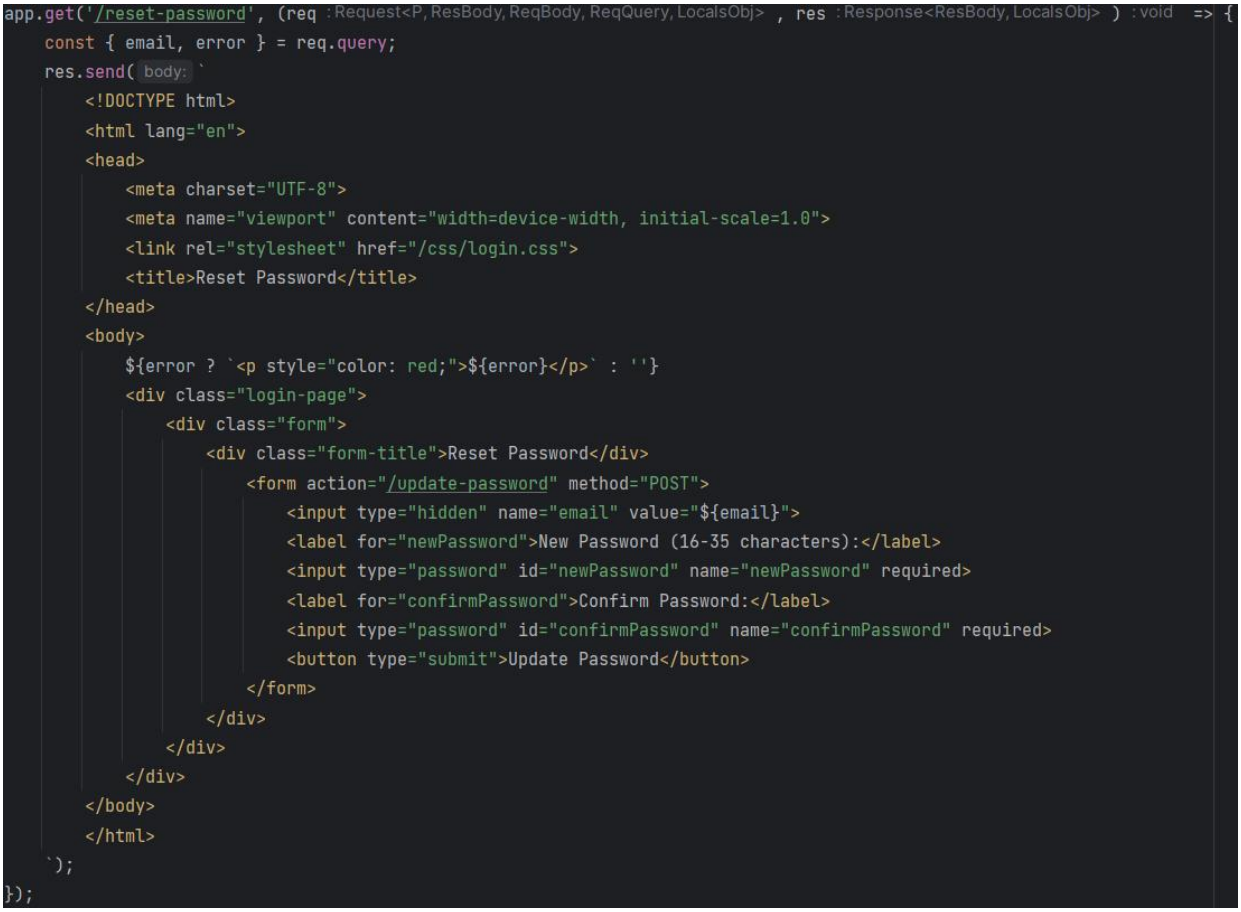
```
app.get('/', (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : void  => {
    const { error } = req.query;
    res.send( body: `
        <!DOCTYPE html>
        <html lang="en">
        <head>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width, initial-scale=1.0">
            <link rel="stylesheet" href="/css/login.css">
            <title>Login</title>
        </head>
        <body>
            ${error ? `<p style="color: red;">${error}</p>` : ''}
            <div class="login-page">
                <div class="form">
                    <div class="form-title">Login</div>
                    <form class="login-form" action="/login" method="POST">
                        <label for="email">Email:</label>
                        <input type="text" id="email" name="email" required>
                        <label for="password">Password:</label>
                        <input type="password" id="password" name="password" required>
                        <button type="submit">Login</button>
                    </form>
                </div>
            </div>
        </body>
        </html>
    `);
});
```

Рисунок 3.3 – Лістинг маршруту GET /.



```
app.post( path: '/login', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<...> => {
    try {
        const { email, password } = req.body;
        const result = await pool.query(`
            SELECT u.id, u.password, r.responsibilities
            FROM users u
            JOIN roles r ON u.role = r.role_name
            WHERE u.email = $1
        `, [email]);
        if (result.rows.length === 0) {
            return res.redirect( url: '/?error=User not found');
        }
        const user = result.rows[0];
        if (password === user.password) {
            if (password === STANDARD_PASSWORD) {
                res.redirect( url: `/reset-password?email=${email}`);
            } else {
                const responsibilities : any  = typeof user.responsibilities === "string"
                    ? JSON.parse(user.responsibilities)
                    : user.responsibilities;
                res.redirect( url: `http://localhost:3003/dashboard?user_id=${user.id}&responsibilities=${encodeURIComponent(JSON.stringify(responsibilities))}`);
            }
        } else {
            return res.redirect( url: '/?error=Invalid password');
        }
    } catch (error) {
        console.error('Error occurred:', error.message);
        res.status( code: 500).send( body: 'Internal Server Error');
    }
});
```

Рисунок 3.4 – Лістинг маршруту POST /login.



```
app.get('/reset-password', (req :Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res :Response<ResBody, LocalsObj> ) : void  => {
    const { email, error } = req.query;
    res.send( body: `
        <!DOCTYPE html>
        <html lang="en">
        <head>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width, initial-scale=1.0">
            <link rel="stylesheet" href="/css/login.css">
            <title>Reset Password</title>
        </head>
        <body>
            ${error ? `<p style="color: red;">${error}</p>` : ''}
            <div class="login-page">
                <div class="form">
                    <div class="form-title">Reset Password</div>
                        <form action="/update-password" method="POST">
                            <input type="hidden" name="email" value="${email}">
                            <label for="newPassword">New Password (16-35 characters):</label>
                            <input type="password" id="newPassword" name="newPassword" required>
                            <label for="confirmPassword">Confirm Password:</label>
                            <input type="password" id="confirmPassword" name="confirmPassword" required>
                            <button type="submit">Update Password</button>
                        </form>
                    </div>
                </div>
            </div>
        </body>
        </html>
    `);
});
```

Рисунок 3.5 – Лістинг маршруту GET /reset-password.

```
app.post( path: '/update-password', handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<...> => {
    try {
        const { email, newPassword, confirmPassword } = req.body;
        if (newPassword.length < 16 || newPassword.length > 35) {
            return res.redirect( url: `/reset-password?email=${email}&error=Password must be between 16 and 35 characters`);
        }
        if (newPassword !== confirmPassword) {
            return res.redirect( url: `/reset-password?email=${email}&error=Passwords do not match`);
        }
        await pool.query('UPDATE users SET password = $1 WHERE email = $2', [newPassword, email]);
        res.send( body: 'Password updated successfully! You can now login with your new password.');

    } catch (error) {
        console.error('Error occurred:', error.message);
        res.status( code: 500).send( body: 'Internal Server Error');
    }
});
```

Рисунок 3.6 – Лістинг маршруту POST /update-password.

Запуск сервера відбувається за допомогою константи PORT (див. рис. 3.7).

```
app.listen(PORT, hostname: () : void => {
    console.log(`Server(auth) is running on http://localhost:${PORT}`);
});
```

Рисунок 3.7 – Лістинг запуск серверу.

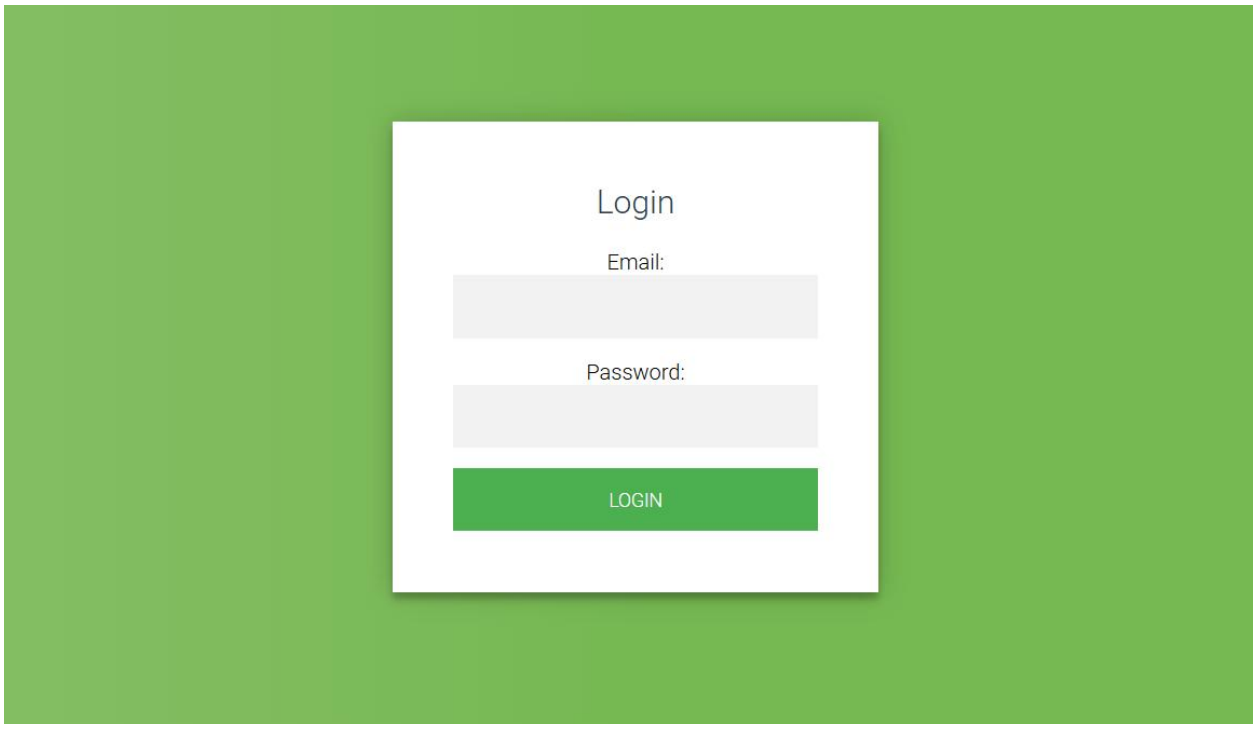


Рисунок 3.8 – Сторінка автентифікації.

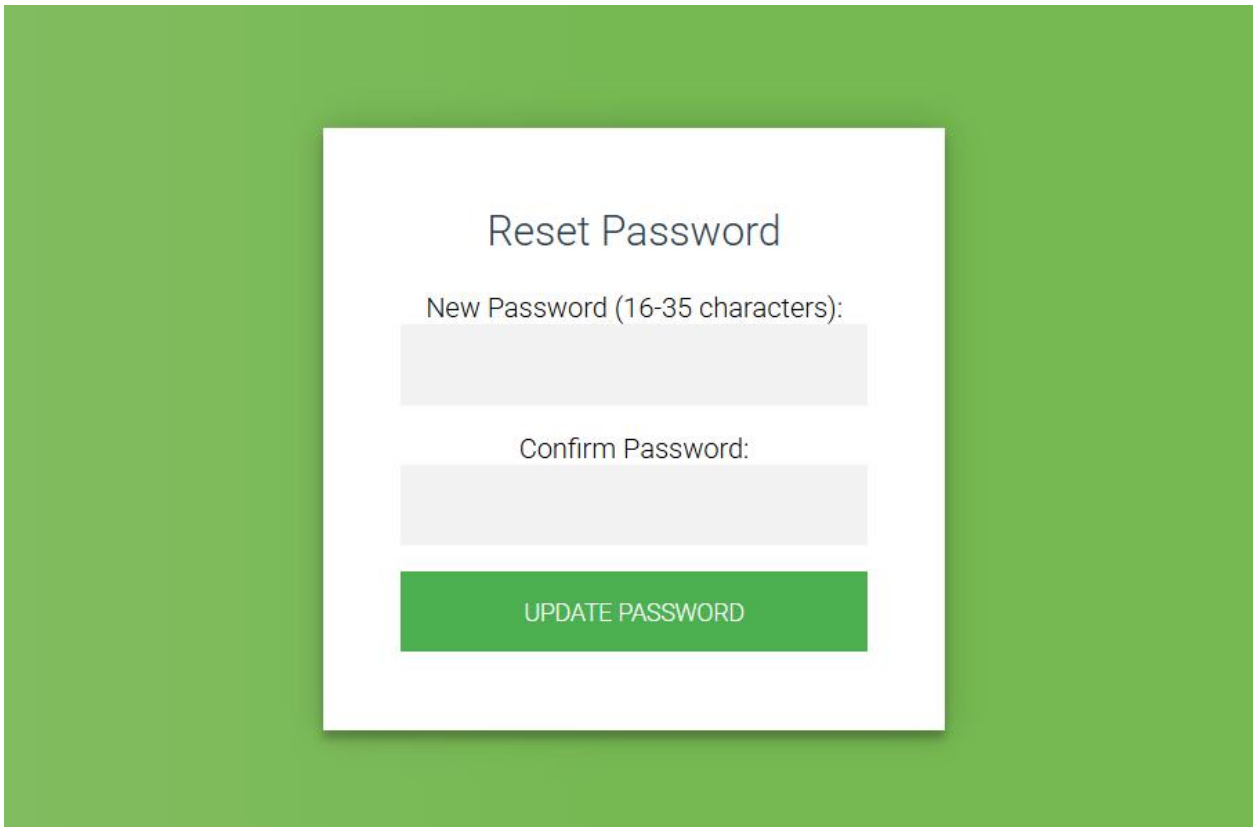


Рисунок 3.9 – Сторінка змінення пароля.

Як підсумок маємо окремо виділений сервіс автентифікації, що можна незалежно масштабувати та механізм якого оновлювати, забезпечує надійну та гнучку автентифікацію.

3.2. Реалізація основного сервісу

Вихідний код знаходиться у файлі server.js, що відповідає за всю бізнес-логіку: роботу з виробничими замовленнями, матеріалами, інвентарем, закупівлями, бюджетами, аналітикою та таким іншим. Код основного сервісу також використовує Express, body-parser та пул з’єднань pg.Pool, константу PORT та посередників (див. рис. 3.10).

```
const express : e | () => core.Express  = require('express
const bodyParser : {...} | {...}  = require('body-parser');
const { Pool } = require('pg');
const app : any | Express  = express();
const PORT : number  = 3003;
const pool = new Pool({
    user: 'postgres',
    host: 'localhost',
    database: 'company_data',
    password: 'kolokol2',
    port: 5432
});
app.use(bodyParser.json());
app.use(express.static( root: 'public'));
app.use(bodyParser.urlencoded({ extended: true }));
```

Рисунок 3.10 – Лістинг імпорту пакетів, налаштування з'єднання з БД та посередники основного сервісу.

Для зручності розділимо реалізацію сервісу на кілька блоків:

1) Контроль якості та продуктивність працівників:
    - PUT /api/update-quality-status – запис замовлення у workers_performance, автоматичне додавання виготовлених продуктів у finished_goods_inventory і видалення з production_orders [5].
    - GET/DELETE /api/worker-performance – отримання та видалення записів продуктивності відповідно [5].
    - GET /api/quality-control-orders/:userId – отримання замовлень із статусом Completed для контролю якості [5].
2) Виробничі замовлення:
    - POST /api/start-production – перевіряє та списує матеріали із warehouse_inventory, створює запис у production_orders, генерує заявки на закупівлю, якщо матеріалів для виготовлення продукції не достатньо [5].
    - GET /api/production-orders – повертає повний список замовлень.

- GET /api/production-orders/user/:userId – отримання матеріалів із розрахунком загальних кількостей для виробництва.
- PUT /api/update-order-status, PUT /api/update-order-notes, PUT /api/update-order – оновлення статусу, нотаток та параметрів замовлення відповідно.
- DELETE /api/production-orders/:id – видалення замовлення із поверненням матеріалів на склад.

3) Матеріали та інвентар:
    - GET /api/material-costs та PUT /api/material-costs/:material_name – отримання списку матеріалів з таблиці material_costs та змінення ціни матеріалу відповідно.
    - GET /api/warehouse-inventory – отримання списку матеріалів на складі.
4) Продукти:
    - CRUD-операції /api/products – довідник продуктів, оновлення масиву матеріалів, автоматичне синхронне очищення material_costs та warehouse_inventory.
5) Закупівлі та постачальники:
    - GET/POST/PUT/DELETE /api/purchase-requests, GET /api/active-purchase-requests, PUT /api/update-purchase-status – робота із заявками на поставки матеріалів у таблиці purchase_requests та переміщення виконаних поставок в purchase_history.
    - GET/POST/PUT/DELETE /api/suppliers – CRUD для довідника постачальників, з перевіркою на наявність пов’язаних заявок.
6) Бюджети та бюджетні статті:
    - GET/POST /api/budgets, GET /api/budgets/:id/items, PATCH /api/budget-items/:id/actual – CRUD-операції з бюджетом та оновлення фактичних витрат.
7) Прогнозні сценарії:

- GET/POST /api/forecasts, GET /api/forecasts/:id – створення та отримання сценаріїв для прогнозу та їх запис у таблиці forecast_scenarios та forecast_entries.

8) Користувачі та посади:
    - GET/POST/PUT/DELETE /api/employees та /api/roles – CRUD-операції для таблиць користувачів users та посад roles, із валідацією полів та перевіркою прав.
9) Конкуренти, кампанії, клієнти:
    - CRUD-операції /api/competitors, CRUD-операції /api/campaigns, CRUD-операції /api/clients – окремі маршрути для таблиць конкурентів, кампаній та клієнтів відповідно з бізнес-логікою валідації та обробки дат.
10) Замовлення продажів:
    - GET/POST/PUT/DELETE /api/sales-orders, GET /api/completed-sales – списання й повернення товару в складі, переміщення виконаних замовлень у completed_sales.
11) Розрахунок собівартості:
    - GET /api/product-cost/:productId – динамічний розрахунок вартості одиниці, побудова масиву breakdown і суми totalCost.

У основному сервісі кожен маршрут використовує конструкцію try/catch, що дає змогу централізовано ідентифікувати помилки доступу до бази та некоректні вхідні дані, що у разі непередбачуваної помилки повертає 500 Internal Server Error, а бізнес-помилки, некоректний статус чи недостатня кількість на складі, обробляються окремо з відповідним HTTP-кодом 400 та 404. Завдяки такій обробці помилок ми отримуємо:

- Чітку діагностику під час збоїв у консолі.
- Уніфіковані відповіді клієнту для легшої інтеграції й дебагу на фронтенді.

Запуск і логування в кінці файлу забезпечується коефіцієнтом порту 3003 (див. рис. 3.11).

```
app.listen(PORT,  hostname: () : void  => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Рисунок 3.11 – Лістинг запуску серверу.

Як підсумок маємо основний сервіс, що реалізовано як монолітний Express-додаток, розбитий на логічні блоки. Використання пулу pg.Pool забезпечує високу продуктивність БД, чітка обробка помилок та валідація в кожному маршруті гарантують стабільність роботи. Такий підхід повністю відповідає вимогам проєктування з розділу 2 та створює надійну основу для клієнтської частини. Частковий лістинг коду та посилання на повний лістинг знаходяться у додатку Г.

3.3. Клієнтська частина основного сервісу

У корені сервісу вся клієнтська частина зосереджена в папці public, яка містить:

- auth_server.js – сервіс автентифікації.
- server.js – основний сервіс.
- папку css з app.css для створення дизайну основного сервісу та login.css – для сервісу автентифікації.

Клієнтська частина основного сервісу базується на HTML-шаблонах, один з яких є основою сторінки, інші - опираються на обов'язках посад. Основа має заголовок, підключення сервісу до стилів дизайну до нього, простір для контенту, куди через JavaScript вставляється потрібна розмітка з урахуванням обов’язків поточної ролі та скрипти, що ініціалізують завантаження даних та керують відображенням розділів. Шаблони для секцій не зберігаються як окремі HTML-файли, а формуються під час завантаження сторінки за допомогою JS-модулів. Кожен модуль генерує розмітку таблиць і форм лише

для тих обов’язків, що у користувача є в обов'язках userResponsibilities. Наприклад, при наявності обов'язку «Monitor warehouse inventory» скрипт вставляє секцію зі списками матеріалів та продуктів на складі. Користувач із обов’язком «Calculate product cost» побачить власний блок з двома таблицями для розрахунків і редагування вартості матеріалів. Форма логіну та скидання пароля реалізована окремим сервісом. Вона підключає свій дизайн login.css, але далі не змінюється в залежності від посад, оскільки призначена лише для перевірки доступу. Таким чином, поєднання базового шаблону та динамічних шаблонів дозволяє гнучко адаптувати інтерфейс під різні набори обов’язків, забезпечуючи узгоджений дизайн і мінімальну кількість статичних HTML-файлів.

CSS-оформлення реалізоване в app.css і login.css, встановлює єдину стилістику, забезпечує зручний інтерфейс та підтримує всі потрібні компоненти: навігацію, таблиці, модальні вікна та сповіщення. Завдяки використанню CSS-змінних легко змінювати колірну гаму або відступи в усьому додатку, що відповідає вимогам масштабованості та підтримки в довгостроковій перспективі.

У якості прикладу для демонстрації візуальної частини роботи було обрано користувача з посадою «Admin» з усіма існуючими в БД обов'язками.

1) Обов’язок «Manage production processes»

Рисунок 3.12 – Форма створення продукту та таблиця їх відображення.

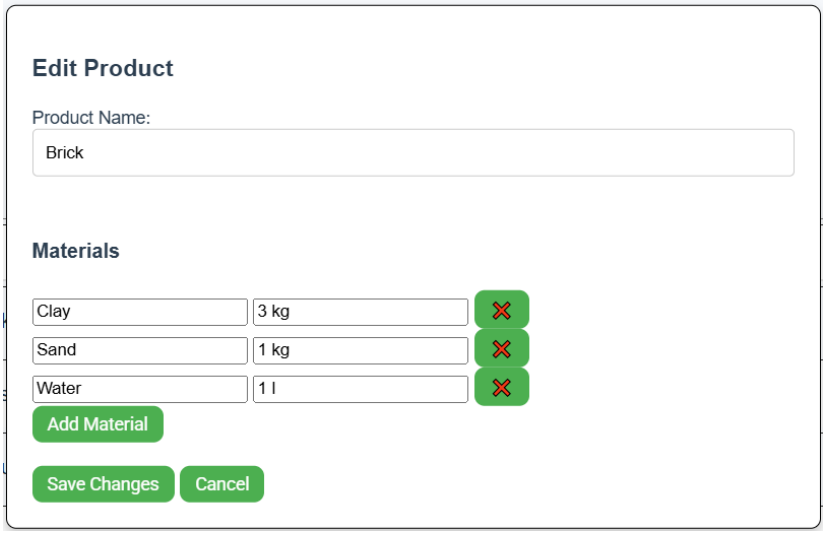


Рисунок 3.13 – Модальне вікно для редагування продукту.

Рисунок 3.14 – Форма створення заявки на виготовлення продукту та таблиця їх відображення.

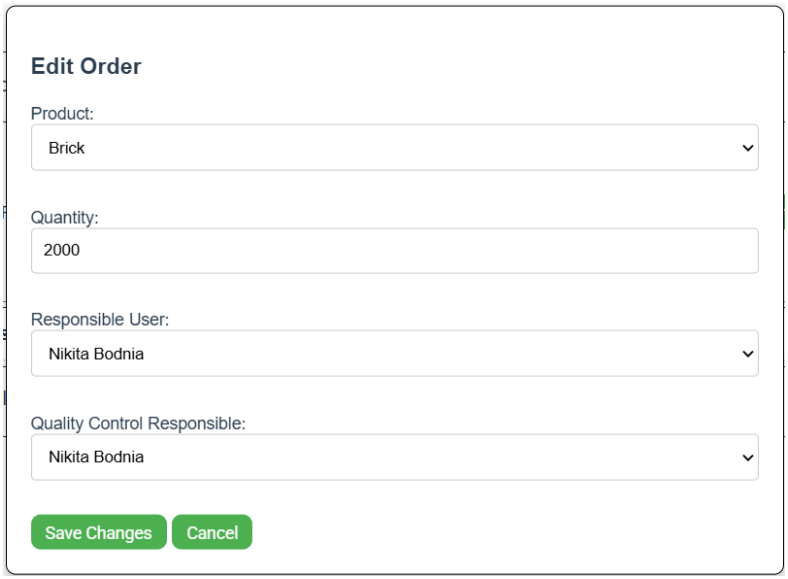


Рисунок 3.15 – Модальне вікно для редагування заявки на виготовлення продукту.

2) Обов’язок «Perform production»

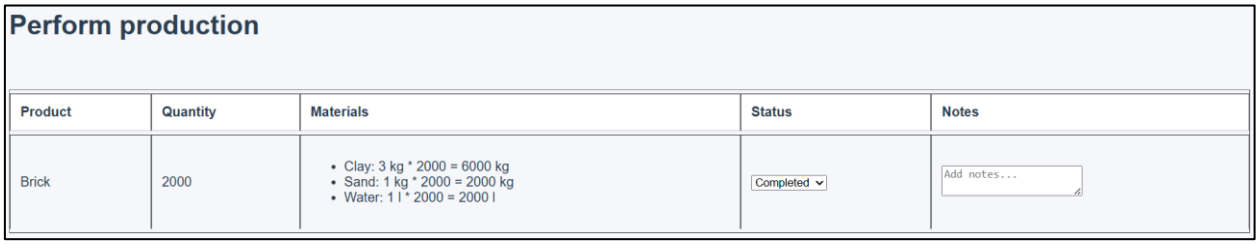


Рисунок 3.16 – Таблиця для відображення виробництва продукту для відповідальної особи.

3) Обов’язок «Control product quality»

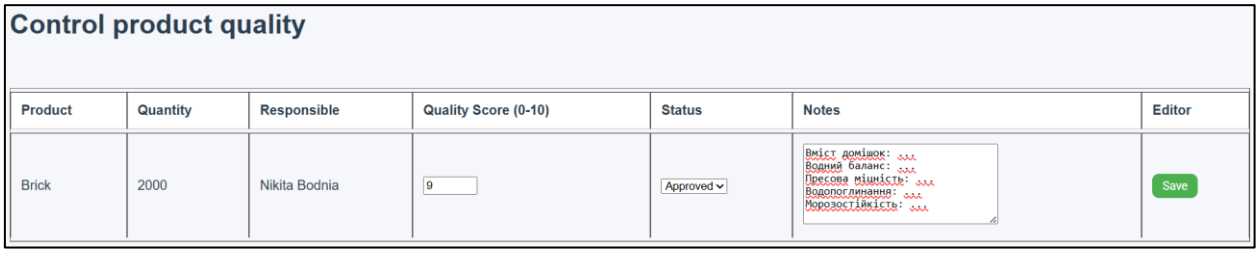


Рисунок 3.17 – Таблиця для контролю якості виробництва для відповідальної особи.

4) Обов’язок «Monitor workers performance»

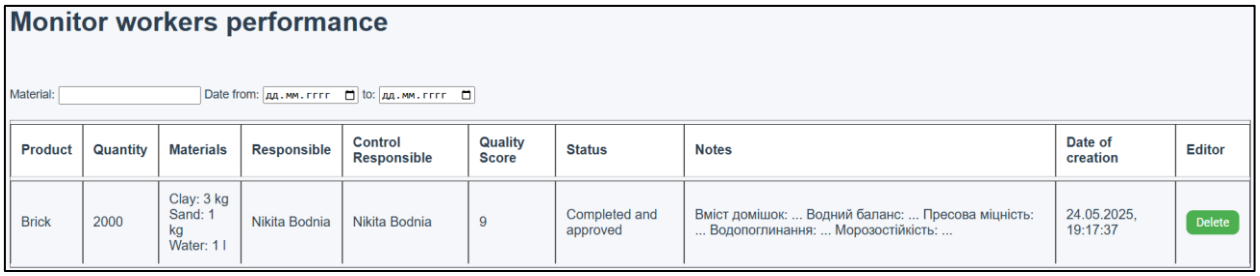


Рисунок 3.18 – Таблиця для відстеження якості роботи працівників та форма для пошуку.

5) Обов’язок «Procure raw materials»

Рисунок 3.19 – Форма додавання нового постачальника та таблиця для їх відображення.

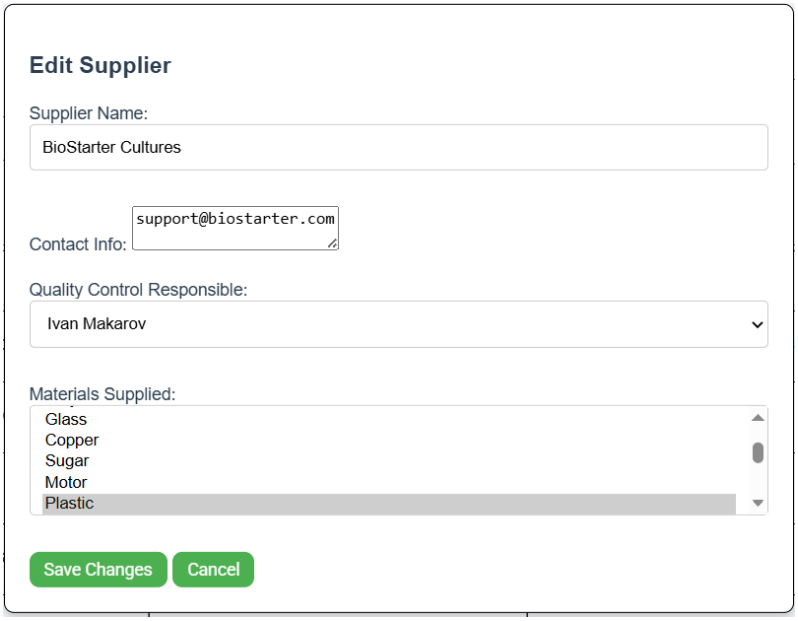


Рисунок 3.20 – Модальне вікно для редагування постачальника.

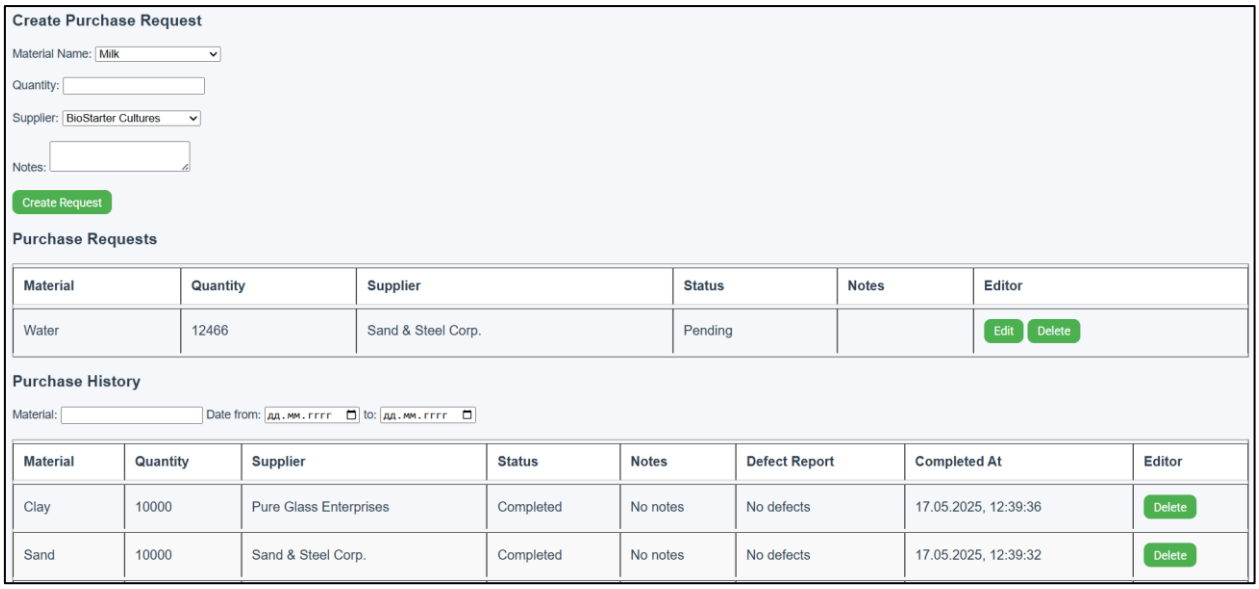


Рисунок 3.21 – Форма для створення заявки на постачання, таблиця для їх відображення та таблиця для відображення попередніх заявок.

Рисунок 3.22 – Модальне вікно для редагування заявки на постачання.

6) Обов’язок «Procure quality control»

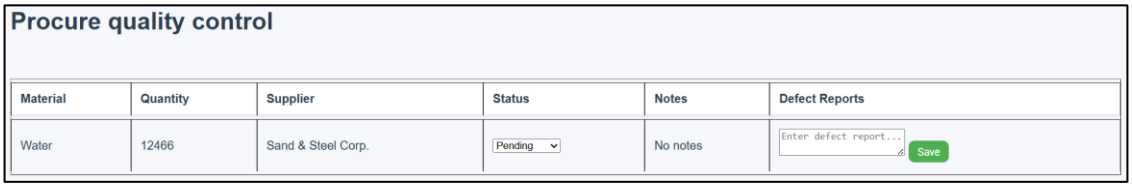


Рисунок 3.23 – Таблиця для контролю якості постачання для відповідальної особи.

7) Обов’язок «Monitor warehouse inventory»

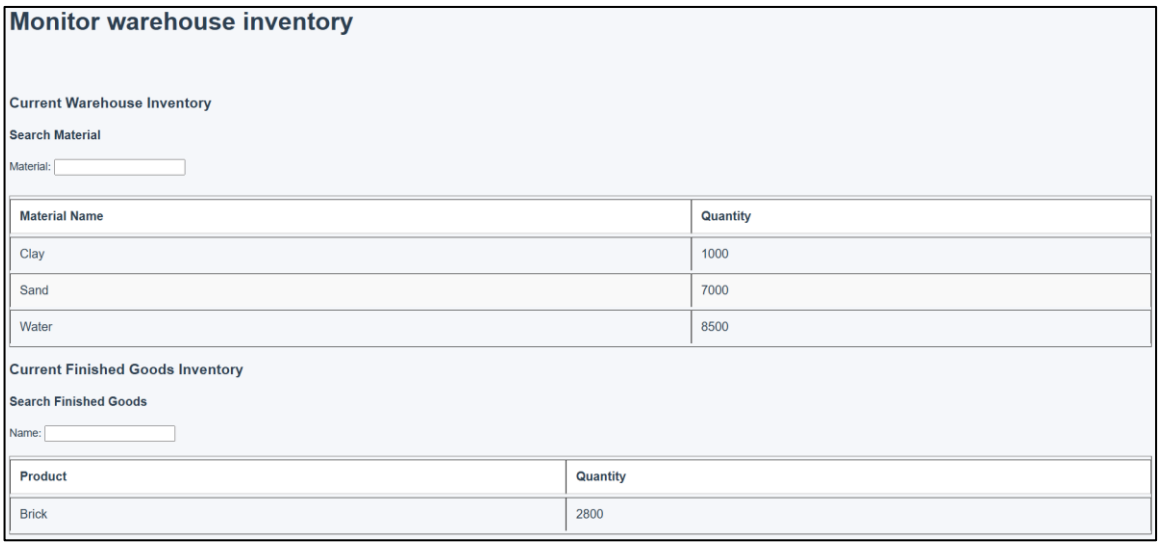


Рисунок 3.24 – Форма для пошуку матеріалу на складі, таблиця для їх відображення, форма для пошуку продукту на складі та таблиця для їх відображення.

8) Обов’язок «Recruit and hire employees»

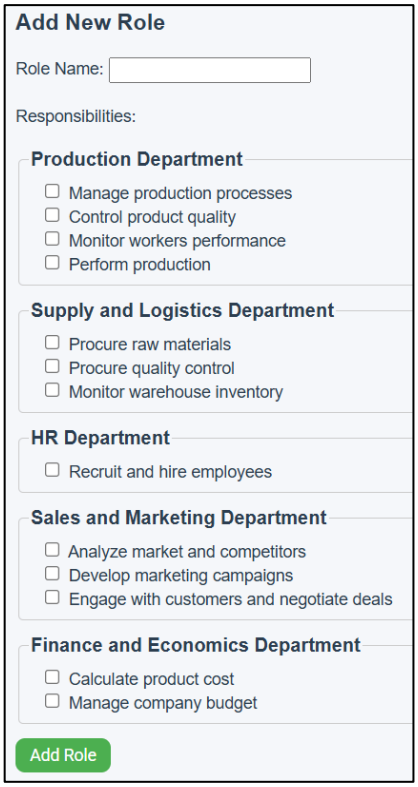
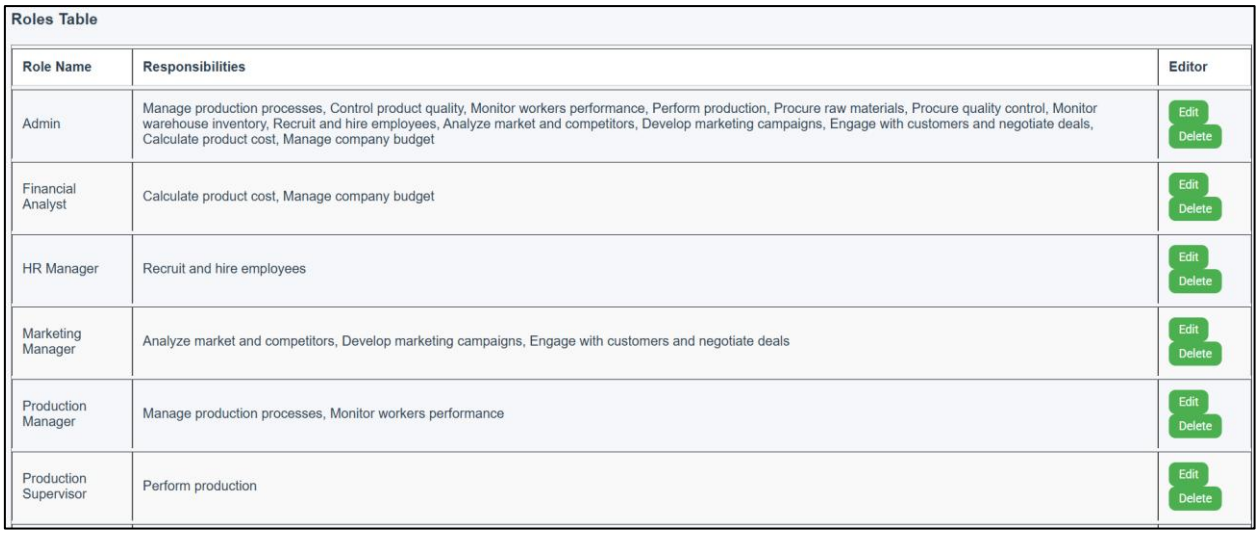


Рисунок 3.25 – Форма для створення посади.



Roles Table

Role Name	Responsibilities	Editor
Admin	Manage production processes, Control product quality, Monitor workers performance, Perform production, Procure raw materials, Procure quality control, Monitor warehouse inventory, Recruit and hire employees, Analyze market and competitors, Develop marketing campaigns, Engage with customers and negotiate deals, Calculate product cost, Manage company budget	Edit Delete
Financial Analyst	Calculate product cost, Manage company budget	Edit Delete
HR Manager	Recruit and hire employees	Edit Delete
Marketing Manager	Analyze market and competitors, Develop marketing campaigns, Engage with customers and negotiate deals	Edit Delete
Production Manager	Manage production processes, Monitor workers performance	Edit Delete
Production Supervisor	Perform production	Edit Delete

Рисунок 3.26 – Таблиця для відображення посад.

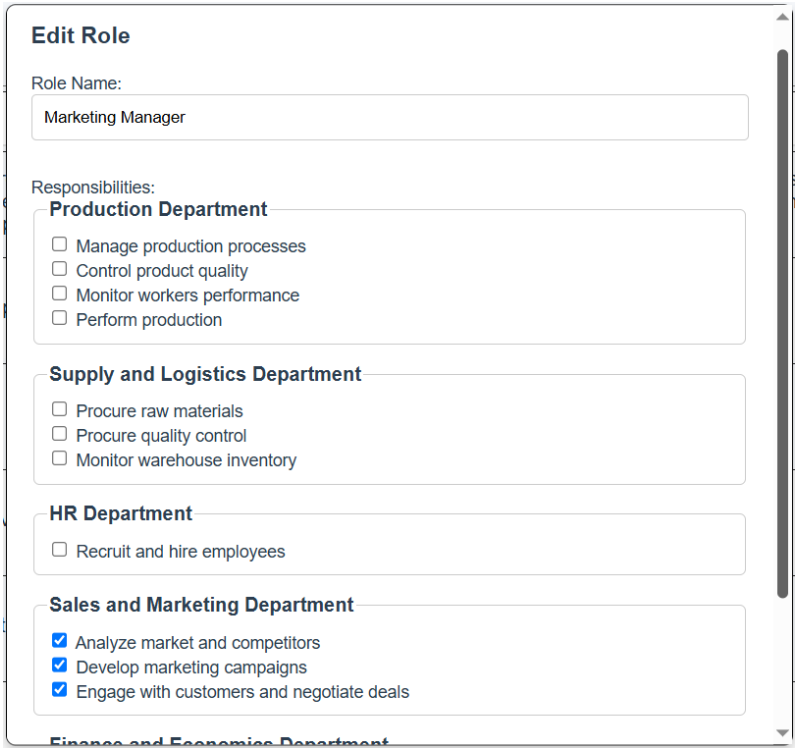


Рисунок 3.27 – Модальне вікно для редагування посади.

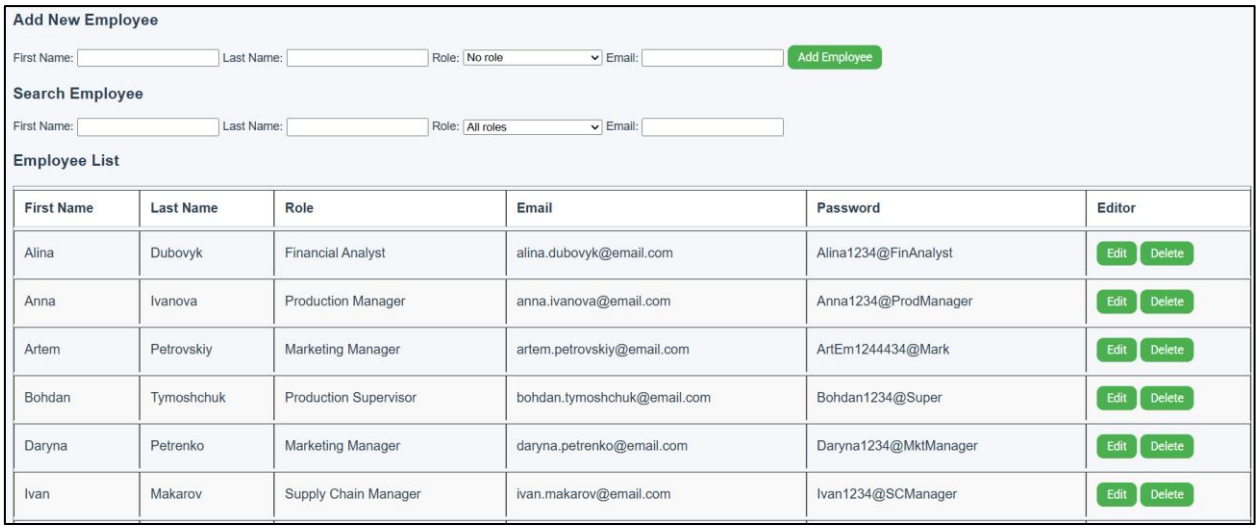


Рисунок 3.28 – Форми для створення нового користувача, пошуку користувача у таблиці та таблиця для їх відображення.

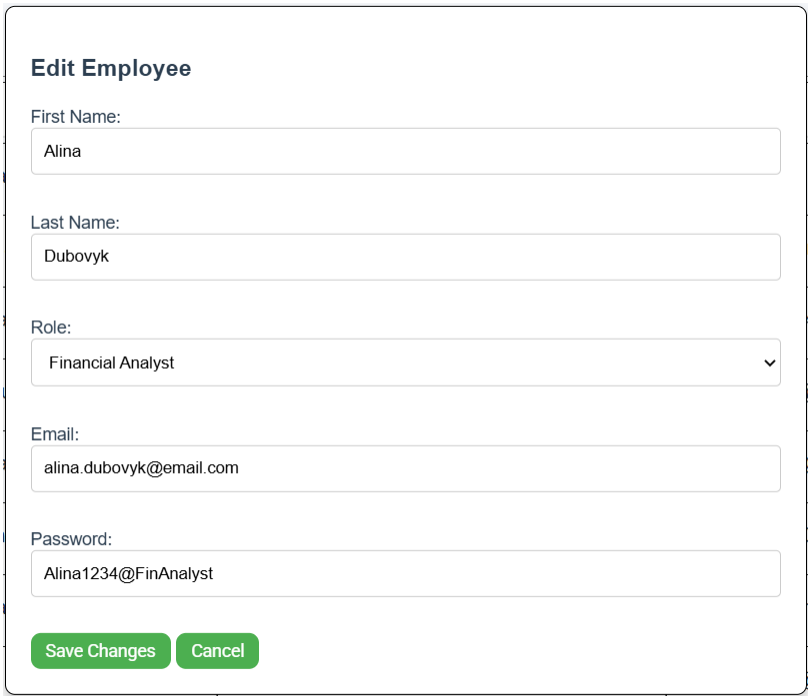


Рисунок 3.29 – Модальне вікно для редагування користувача.

9) Обов’язок «Analyze market and competitors»

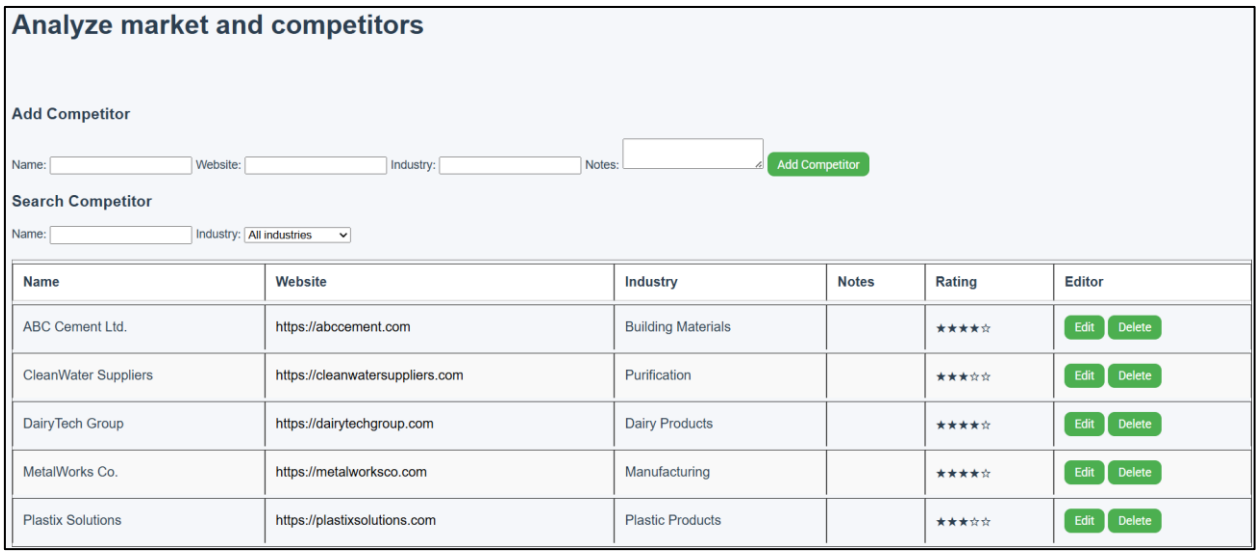


Рисунок 3.30 – Форми для створення конкурента, знаходження конкурента в таблиці та таблиця для їх відображення.

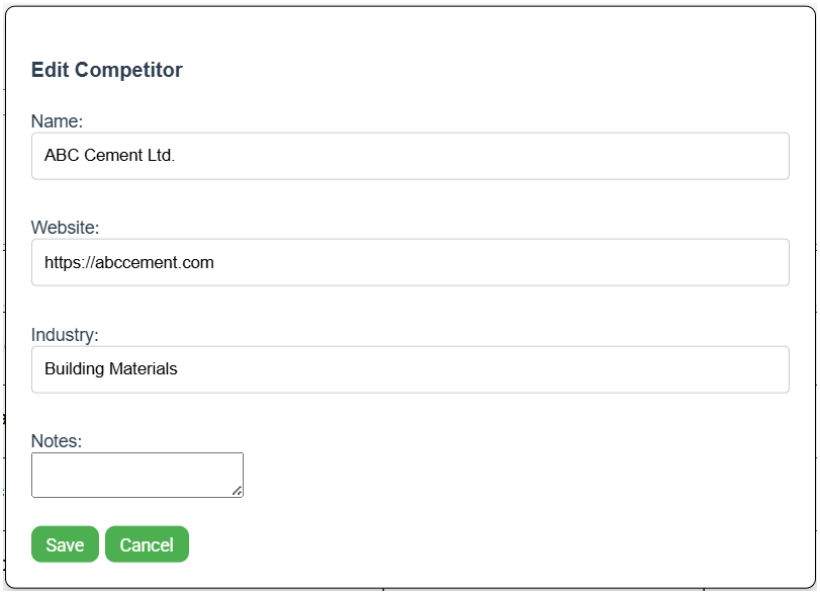


Рисунок 3.31 – Модальне вікно для редагування конкурента.

10) Обов’язок «Develop marketing campaigns»

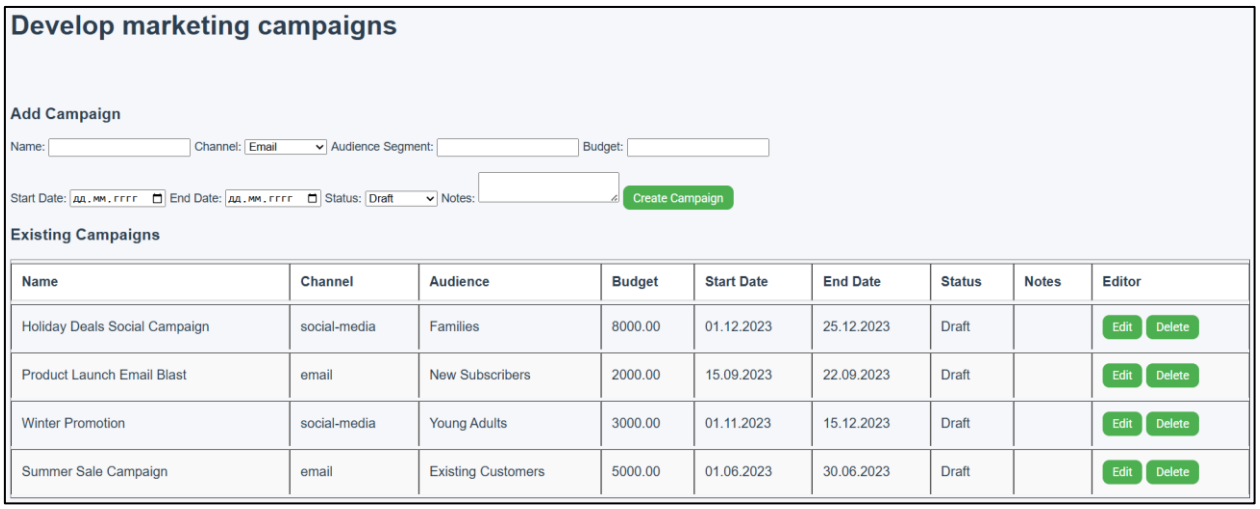


Рисунок 3.32 – Форма для створення кампанії та таблиця для їх відображення.

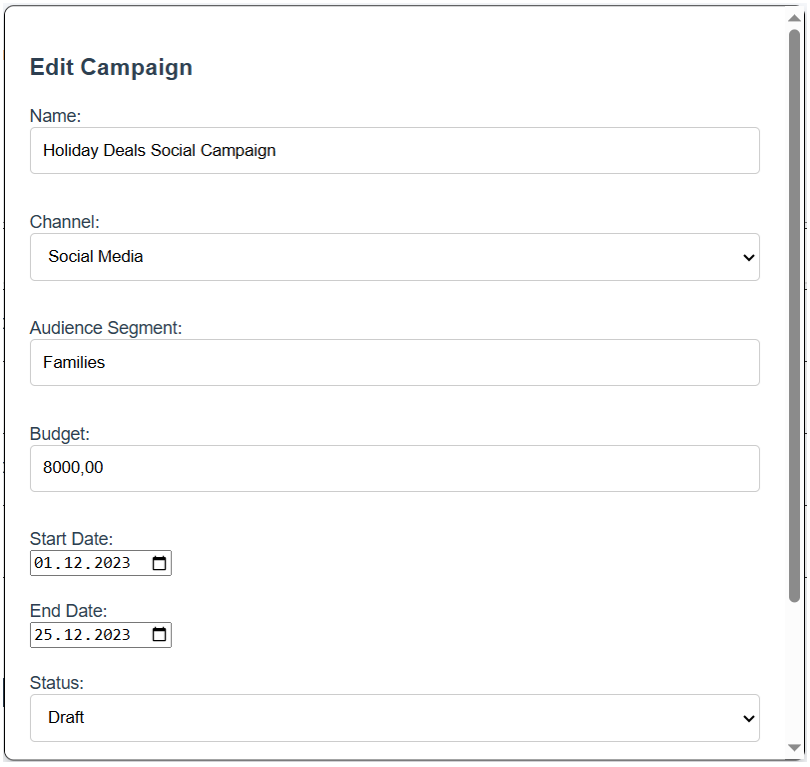


Рисунок 3.33 – Модальне вікно для редагування кампанії.

11) Обов’язок «Engage with customers and negotiate deals»

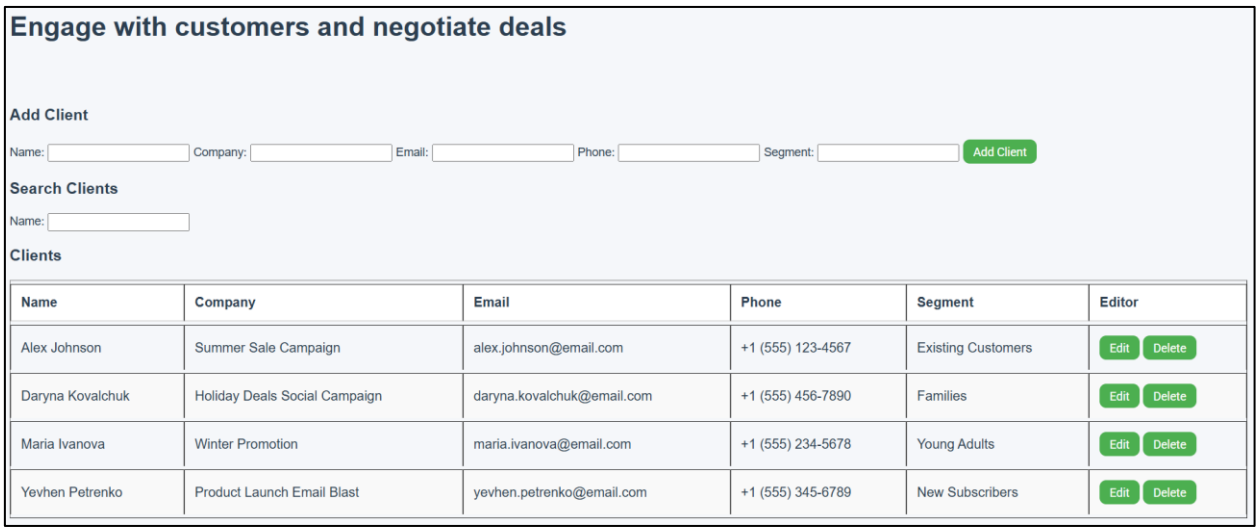


Рисунок 3.34 – Форми для створення клієнта, знаходження клієнта в таблиці та таблиця для їх відображення.

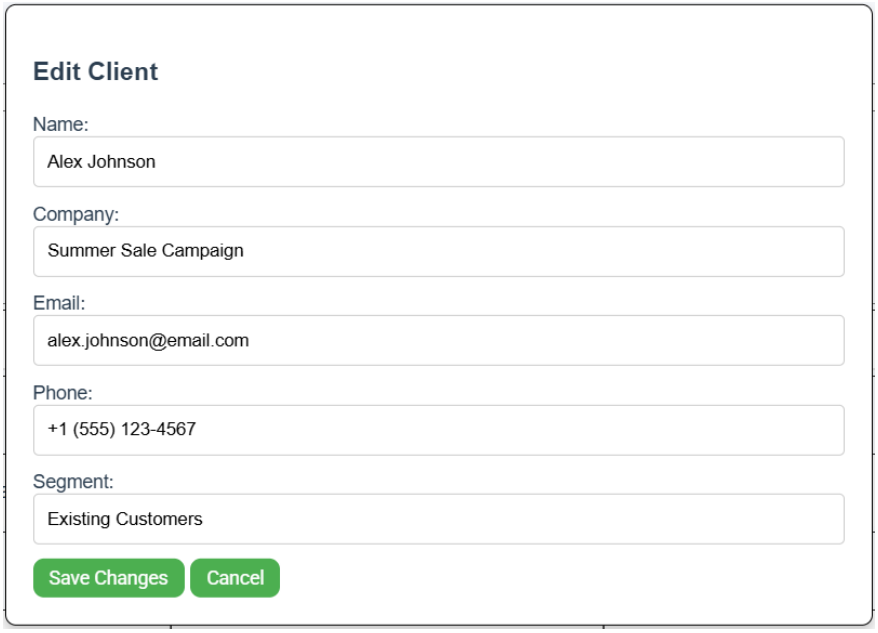


Рисунок 3.35 – Модальне вікно для редагування клієнта.

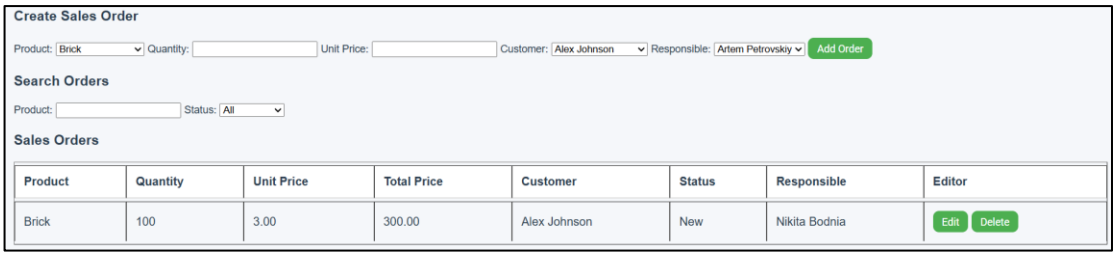


Рисунок 3.36 – Форми для створення заявки на продажу, пошуку заявки в таблиці та таблиця для їх відображення.

Рисунок 3.37 – Форма для пошуку заявки у таблиці історії продажів та таблиця для їх відображення.

12) Обов’язок «Calculate product cost»

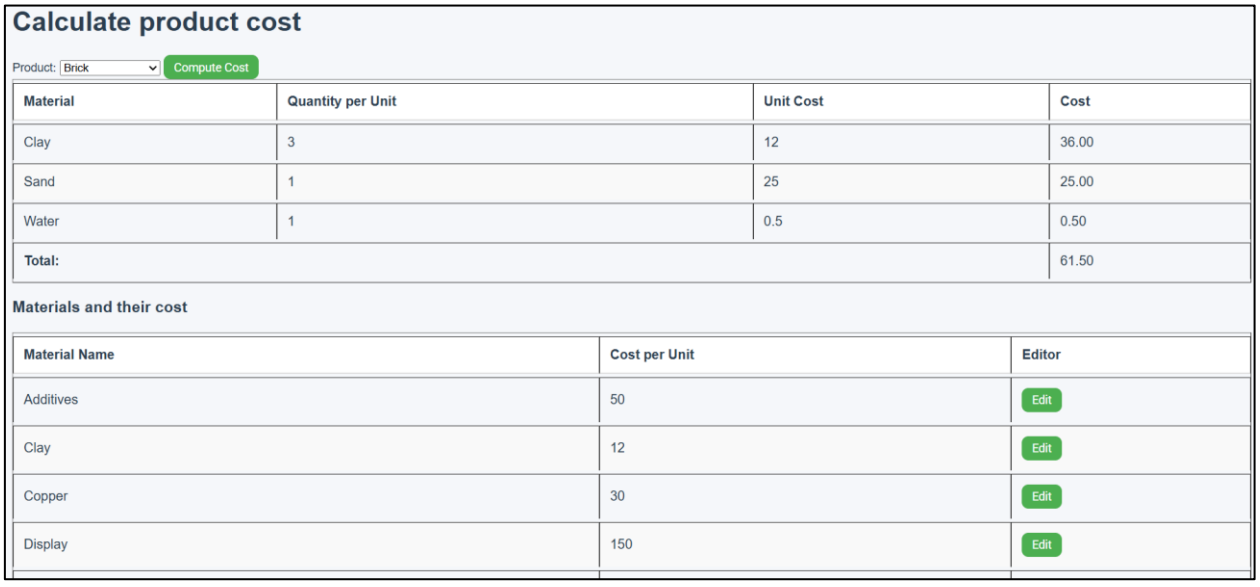


Рисунок 3.38 – Таблиця для розрахунку ціни виготовлення продукту за одну одиницю та таблиця відображення ціни матеріалу за одну одиницю.

Рисунок 3.39 – Модальне вікно для редагування ціни матеріалу за одну одиницю.

13) Обов’язок «Manage company budget»

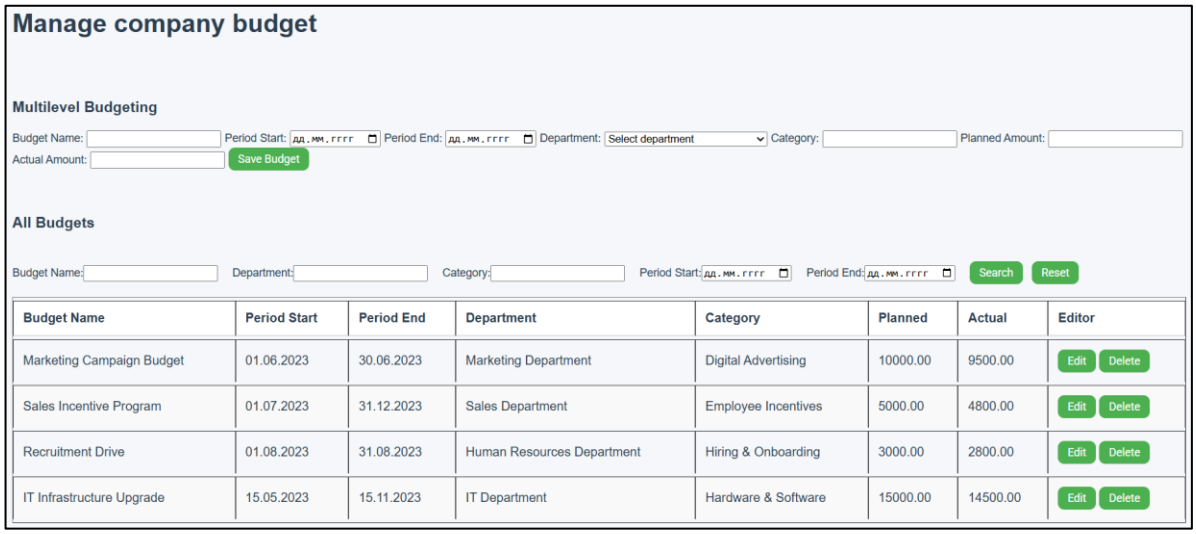


Рисунок 3.40 – Форми для створення бюджету, знаходження бюджету у таблиці та таблиця для їх відображення.

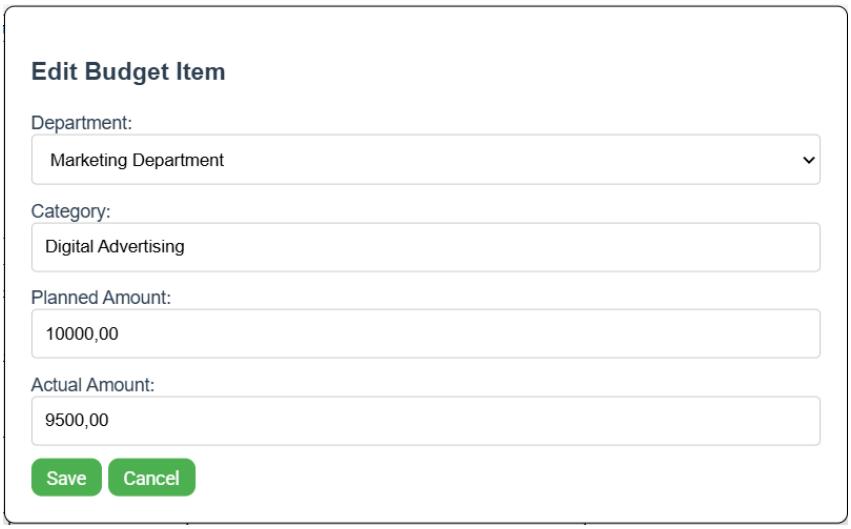


Рисунок 3.41 – Модальне вікно для редагування бюджету.

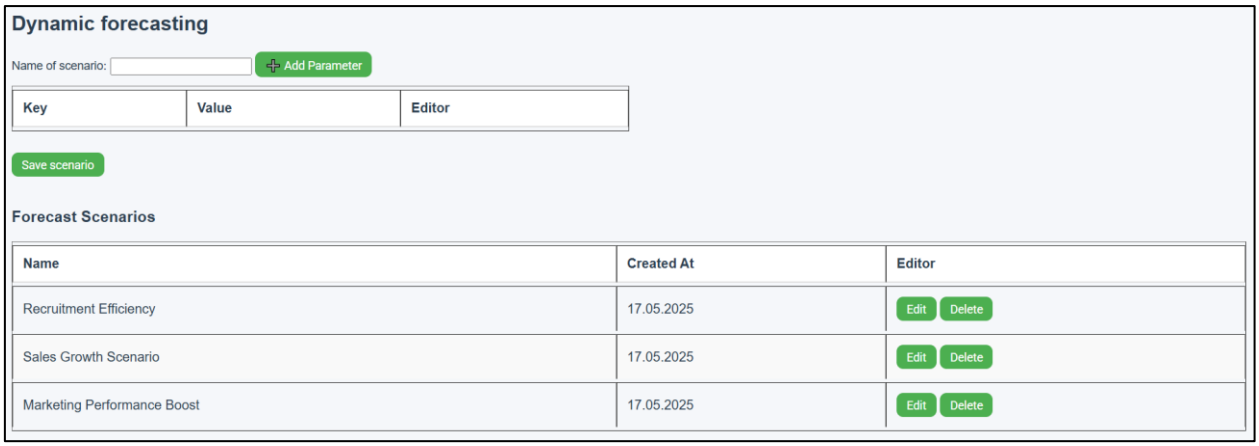


Рисунок 3.42 – Форма для створення сценарію та таблиця для їх відображення.

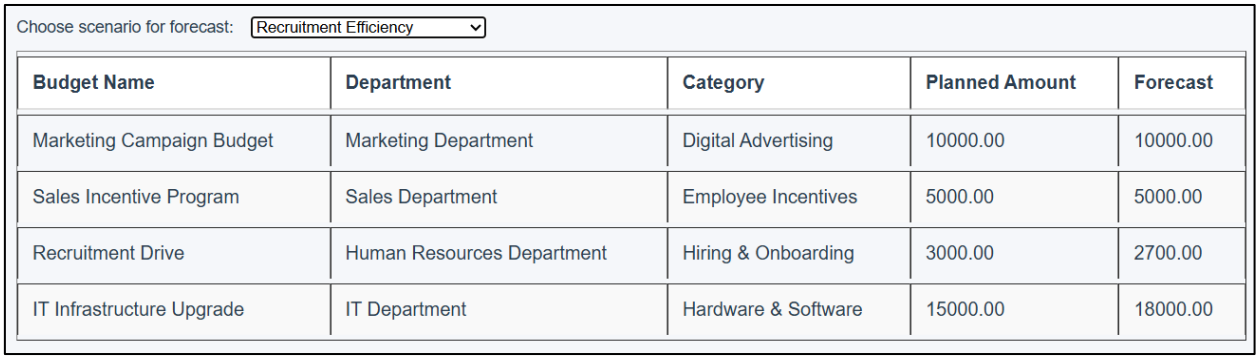


Рисунок 3.43 – Таблиця для розрахунку припущеної суми за обраним сценарієм.

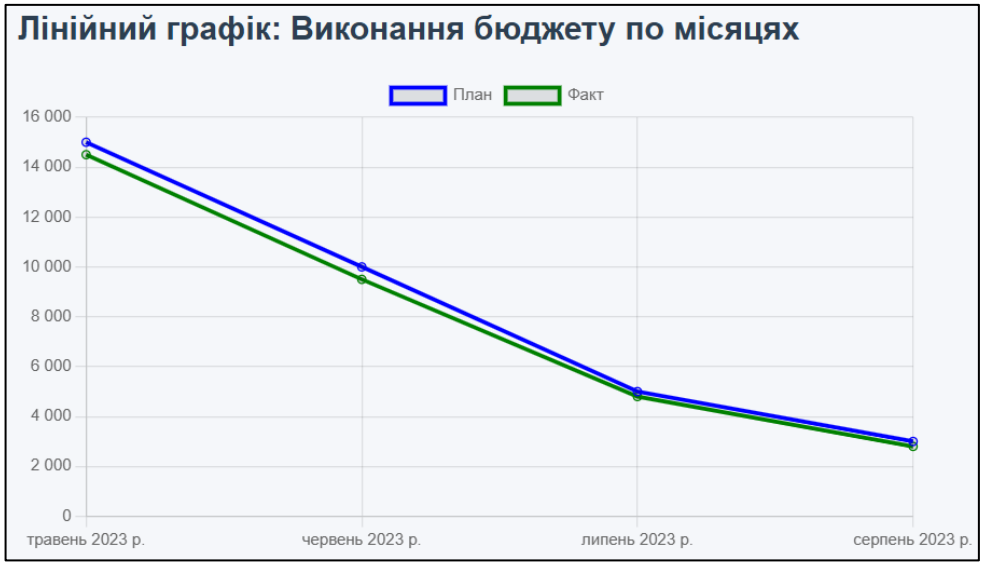


Рисунок 3.44 – Лінійний графік для відображення виконання бюджету по місяцях.

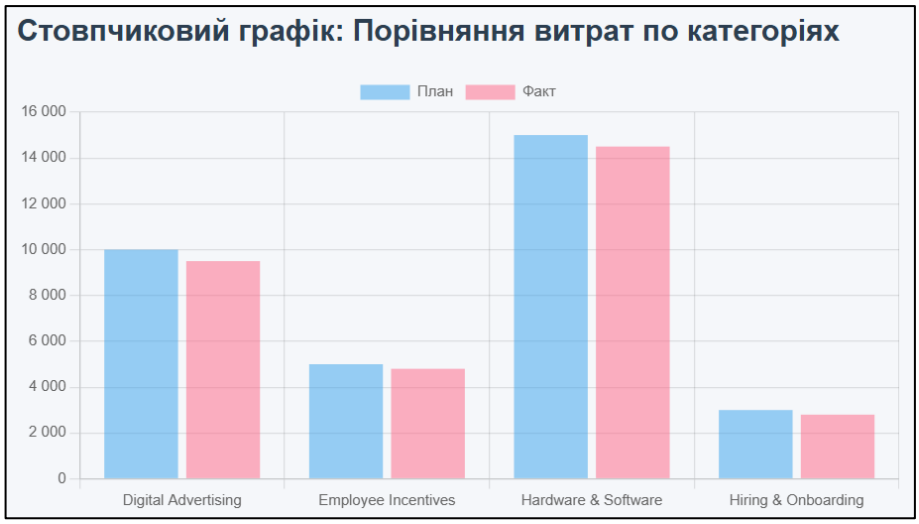


Рисунок 3.45 – Стовпчиковий графік для порівняння витрат за категоріями.

3.4. Рекомендації щодо вдосконалення системи

Враховуючи продемонстровану архітектуру та реалізацію, пропонується низка удосконалень, що підвищать стійкість, безпеку та гнучкість платформи:

1) Впровадження двофакторної автентифікації: додавання SMS-підтвердження при автентифікації значно зменшить ризик надання доступу сторонній особі до облікових записів. Для цього сервіс можна поєднати з популярними провайдерами Google Authenticator або Authy та зберігати секретні ключі у зашифрованому вигляді в БД.
2) Розробити асинхронну обробкУ ресурсоємних завдань: завдання на масовий розрахунок собівартості або імпорт та експорт даних можна винести в чергу і обробляти через worker-процеси. Це зменшить навантаження на основні HTTP-ендпоінти та забезпечить кращу відмовостійкість.
3) Додати кешування на рівні API: для часто запитуваних ресурсів, таких як довідник продуктів, постачальників, доцільно додати Redis-кеш із TTL, що скоротить кількість запитів до БД і покращить час відповіді під час пікових періодів.

4) Створити план відновлення після катастроф: розробити сценарії відновлення БД з архівів та регулярно відпрацьовувати їх у тестовому середовищі.

Реалізація цих рекомендацій дозволить суттєво підвищити рівень безпеки, продуктивності та масштабованості системи, зробить процес розробки та підтримки більш передбачуваним і керованим.

Висновки за розділом 3

У третьому розділі було реалізовано та впроваджено всі компоненти системи – окремий сервіс автентифікації, основний сервіс та клієнтську частину.

Сервіс автентифікації було виділено в окремий файл auth_server.js, що обробляє логін, скидання пароля та керує сесіями через HTTP-only cookies. Завдяки чіткому поділу на маршрути і використанню pg.Pool для роботи з таблицями користувачів та посад вдалося побудувати гнучкий та безпечний механізм перевірки доступу, який легко масштабувати.

Основний сервіс – це Express-додаток server.js, структурований за доменами: виробничі замовлення, матеріали та інвентар, контроль якості та продуктивності, закупівлі, бюджетні статті, аналітичні сценарії, управління користувачами та посадами маркетинг, продажі та розрахунок собівартості. Кожен маршрут реалізоване через try/catch, з централізованою обробкою помилок. Використання пулу з’єднань забезпечило високу продуктивність при одночасній обробці запитів.

Клієнтська частина організована на HTML-шаблонах та наборах JavaScript-модулів. Динамічні шаблони формуються під час завантаження сторінки залежно від обов’язків посади користувача, що забезпечує гнучкість та мінімізацію статичних HTML-файлів.

Тож розділ 3 підсумовує, що виконана реалізація є повноцінним, масштабованим і безпечним рішенням для автоматизованого управління РБД у промисловому середовищі.

ВИСНОВКИ

У цій кваліфікаційній роботі бакалавра було розроблено та обґрунтовано систему автоматизованого управління РБД для виробничих підприємств, що складається з трьох ключових розділів: теоретичні основи та аналіз інструментів, проєктування архітектури й БД та реалізація й впровадження рішення.

По-перше, у розділі 1 було детально обґрунтовано, що автоматизація адміністрування РБД є важливою для сучасних виробничих підприємств через те, що класичні методи налаштування резервних копій, моніторингу та оновлення СУБД не може гарантувати своєчасну реакцію на простої та відмови. Розглянута кандидатура РСУБД як основи корпоративних інформаційних систем забезпечує цілісність і узгодженість даних за великих навантажень та паралельної роботи підсистем. У ході огляду були виокремлені сучасні технологічні підходи, що формують екосистему автоматизації. Результати проведеного аналізу показали, що поєднання таких інструментів здатне забезпечити рівень безперервності, відмовостійкості та масштабованості, який вимагають сучасні виробничі процеси. Упровадження такої стратегії дозволяє не лише мінімізувати ризики тривалих простоїв і втрат даних, а й оптимізувати операційні витрати на утримання IT-інфраструктури.

По-друге, у розділі 2 ми детально окреслили всі аспекти проєктування системи – від чітко сформульованих функціональних і нефункціональних вимог до визначення архітектури та побудови реляційної моделі даних. Функціональні вимоги охопили усі бізнес-операції: безпечну автентифікацію та авторизацію, CRUD-операції для користувачів, посад, матеріалів, замовлень, закупівель, бюджетів та маркетингової складової, а також автоматизований розрахунок собівартості. Нефункціональні вимоги заклали основу для гарантії великої доступності, безпеки та масштабованості. Поділ на два сервіси – автентифікації для управління сесіями та основний для бізнес-логіки – спростив розробку, підвищив стійкість до змін та масштабованість.

Проєкт БД зі спеціалізованими таблицями гарантує цілісність та гнучкість, дозволяє легко формувати складні зв’язки та зберігати історичні дані в таблицях. Таким чином, розділ 2 створив міцну основу, яка забезпечує відповідність системи вимогам виробничого середовища.

По-третє, у розділі 3 було показано практичну реалізацію всієї спроєктованої системи та підтверджено її працездатність в реальних умовах. Автентифікаційний сервіс auth_server.js продемонстрував можливість встановлення нового пароля з довжиною 16–35 символів, управління сесіями через HTTP-only cookies та захист безпеки на рівні middleware. Основний сервіс server.js об’єднав у єдиному Express-додатку бізнес-роути для виробничих замовлень, матеріалів, закупівель, інвентаря, бюджетів, маркетингу та аналітики, кожен із яких реалізовано з чітким розподілом, централізованою обробкою помилок у блоках try/catch. Клієнтська частина базується на HTML-шаблонах, JavaScript-модулях, які динамічно генерують сторінку в залежності від обов'язків посади користувача, забезпечуючи адаптивність. Таким чином, процес впровадження системи підтвердив її відповідність запланованим вимогам, забезпечив необхідний рівень надійності, ефективності та готовність до масштабування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Date C. J. Introduction to Database Systems. Pearson Education, Limited, 2003. 1024 p.
2. body-parser [Електронний ресурс]. Модулі для Node.js. URL: https://ditsmod.github.io/native-modules/body-parser/ (дата звернення: 14.04.2025).
3. Express.js [Електронний ресурс]. Fast, Unopinionated, Minimalist Web Framework for Node.js. URL: https://expressjs.com (дата звернення: 14.04.2025).
4. Elmasri R., Navathe S. B. Fundamentals of Database Systems. Boston : Pearson, 2016. 1200 с.
5. Node.js Documentation [Електронний ресурс]. Node.js Foundation. URL: https://nodejs.org/en/docs (дата звернення: 14.04.2025).
6. OWASP Security Cheat Sheets [Електронний ресурс]. OWASP Foundation. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet (дата звернення: 14.04.2025).
7. PostgreSQL Documentation [Електронний ресурс]. PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs (дата звернення: 07.04.2025).
8. Визначення вимог до програмних систем [Електронний ресурс]. URL: https://surli.cc/ffdzxh (дата звернення: 29.04.2025).
9. Глонь О. В., Дубовой В. М., Мітюшкін Ю. І. Комп’ютеризовані системи керування. Вінниця : ВНТУ, 2005. 157с.
10. Глоба Л. С. Розробка інформаційних ресурсів та систем. Київ : Наук. думка, 2018. 312 с.
11. Корченко О. Г., Шелест М. Є., Казмірчук С. В., Ткач Ю. М., Іванченко Є. В. Менеджмент інформаційної безпеки. Ніжин : ФОП Лук’яненко В.В. ТПК «Орхідея», 2019. 408 с.

12. Куваєв Я. Г., Жукова О. А., Сєчкін І. А. Організація реляційних баз даних. Дніпро : НГУ, 2017. 157 с.
13. Сікірда Ю. В., Залевський А. В., Чорногор Н. О. Інформаційні системи і технології в управлінні зовнішньоекономічною діяльністю. Кропивницький : ЛА НАУ, 2021. 282 с.

ДОДАТКИ

Додаток А

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра комп'ютерних систем та робототехніки
Рівень вищої освіти (освітньо-кваліфікаційний рівень) **Бакалавр**
Галузь знань: 15 – Автоматизація та приладобудування
Спеціальність: 151 – Автоматизація та комп'ютерно-інтегровані технології
Освітня програма «Автоматизація та комп'ютерно-інтегровані технології»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри комп'ютерних
систем та робототехніки
к. ф.-м. н., доц. ХРУСЛОВ М. М.
«02» жовтня 2024року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бодні Нікіти Вадимовича
(прізвище, ім'я, по батькові студента)

1. Тема роботи **СИСТЕМА АВТОМАТИЗОВАНОГО УПРАВЛІННЯ РЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ ДЛЯ ВИРОБНИЧИХ ПІДПРИЄМСТВ**

керівник роботи **Булавін Дмитро Олексійович, кандидат технічних наук, доцент**,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від ***16 квітня 2025 року № 4101-5/962***

2. Строк подання студентом роботи ***30 травня 2025 року***

3. Перелік питань, які потрібно розробити:

1) Аналіз сучасних підходів до автоматизованого управління реляційними базами даних у виробничих підприємствах.
2) Розробка концептуальної моделі системи автоматизованого управління реляційними базами даних для виробничих підприємств.
3) Тестування розробленої системи на ефективність, продуктивність і масштабованість у контексті виробничого підприємства.
4) Розробка рекомендацій щодо вдосконалення системи та її адаптації для різних виробничих процесів.

4. План роботи

№ з/п	Назви етапів роботи	Термін виконання етапів роботи
1.	Затвердження теми роботи	02.10.2024 - 16.10.2024
2.	Аналіз методів управління базами даних, їх розподілу, оптимізації запитів та резервного копіювання	12.11.2024 - 10.12.2024
3.	Реалізація програмного прототипу системи, розробка схеми бази даних, інтеграція з веб-інтерфейсом	11.12.2024 - 28.02.2025
4.	Тестування системи: оцінка продуктивності, безпеки, масштабованості та відмовостійкості	01.03.2025 - 20.03.2025
5.	Аналіз результатів тестування, визначення ефективності роботи системи та її відповідності вимогам	21.03.2025 - 31.03.2025
6.	Підготовка рекомендацій щодо покращення системи та її адаптації для різних виробничих процесів	01.04.2025 - 15.04.2025
7.	Підготовка і оформлення звітних матеріалів	16.04.2025 - 30.04.2025
8.	Підготовка і оформлення звітних матеріалів та додатків кваліфікаційної роботи	01.04.2025 - 30.04.2025
9.	Оформлення пояснювальної записки кваліфікаційної роботи відповідно вимогам до звітів про НДР	01.05.2025 - 30.05.2025
10.	Оформлення звіту про переддипломну практику	01.05.2025 - 30.05.2025
11.	Представлення кваліфікаційної роботи керівнику та рецензенту	30.05.2025

5. Дата видачі завдання ***02 жовтня 2024 року.***

Студент Бодня Н. В. _____
ініціали, прізвище підпис

Керівник роботи Булавін Д.О. _____
ініціали, прізвище підпис

Додаток Б

Затверджую

_____

«_____»_____2024 р.

Технічне завдання на розробку програмного виробу

«Система автоматизованого управління реляційними базами даних для виробничих підприємств»

1.	Введення	1.1 Назва роботи – «Система автоматизованого управління реляційними базами даних для виробничих підприємств» 1.2. Галузь застосування: управління базами даних
2.	Підстава для розробки	2.1. Навчальний план за спеціальністю 151 – Автоматизація та комп'ютерно-інтегровані технології 2.2. Завдання на кваліфікаційну роботу бакалавра за Наказом №4101-5/962 від «16» квітня 2025 року (представити як Додаток А до пояснювальної записки до кваліфікаційної роботи).
3.	Призначення розробки	3.1. Мета розробки: автоматизація процесів управління даними в реляційних базах даних на виробничих підприємствах. 3.2. Призначення розробки: централізоване збереження даних про виробничі процеси, ресурси, фінанси, забезпечення взаємодії між підрозділами підприємства, обробка та оновлення інформації у реальному часі. 3.3. Вхідні дані розробки: витрати матеріалів, виробничі завдання, інформація про співробітників, постачальників. 3.4. Вихідні дані розробки: історія закупівель, виробничих завдань.
4.	Технічні вимоги до програмного виробу	4.1. Функціональні вимоги: реєстрація та автентифікація користувачів із розподілом прав доступу відповідно до посад, можливість пошуку та фільтрації даних за різними критеріями, оновлення даних у режимі реального часу, створення, редагування, видалення та перегляд записів у базі даних. 4.2. Нефункціональні вимоги: інтуїтивно зрозумілий інтерфейс користувача, забезпечення високої продуктивності системи при обробці великих обсягів даних, надійність і стійкість до відмов.

<table>
<tr><td></td><td></td><td colspan="2">4.3. Вимоги до безпеки: валідація вхідних даних для запобігання SQL-ін’єкціям, захист від несанкціонованого доступу до бази даних.</td></tr>
<tr><td>5.</td><td>Вимоги до програмної документації</td><td colspan="2">Документацією до «Система автоматизованого управління реляційними базами даних для виробничих підприємств» можна вважати:
1) Опис основних вимог та функціональності системи
2) Опис проєктного рішення (пояснювальна записка)
3) Технічне завдання
4) Текст програми</td></tr>
<tr><td>6.</td><td>Вимоги до техніко-економічних показників</td><td colspan="2">Вимогами до «Система автоматизованого управління реляційними базами даних для виробничих підприємств» можна вважати:
1) Скорочення часу обробки та аналізу виробничих даних у порівнянні з ручною обробкою.
2) Зниження витрат на адміністрування даних за рахунок автоматизації.</td></tr>
<tr><td>7.</td><td>Стадії і етапи розробки</td><td>Дата</td><td>Назва етапу</td></tr>
<tr><td></td><td></td><td>12.11.2024 - 10.12.2024</td><td>Аналіз методів управління базами даних, їх розподілу, оптимізації запитів.</td></tr>
<tr><td></td><td></td><td>11.12.2024 - 28.02.2025</td><td>Реалізація програмного прототипу системи, розробка схеми бази даних, інтеграція з веб-інтерфейсом.</td></tr>
<tr><td></td><td></td><td>01.03.2025 - 20.03.2025</td><td>Тестування системи: оцінка безпеки, масштабованості та відмовостійкості</td></tr>
<tr><td></td><td></td><td>21.03.2025 - 31.03.2025</td><td>Аналіз результатів тестування</td></tr>
<tr><td></td><td></td><td>01.04.2025 - 15.04.2025</td><td>Підготовка рекомендацій щодо покращення системи та її адаптації для різних виробничих процесів</td></tr>
<tr><td></td><td></td><td>16.04.2025 - 30.04.2025</td><td>Підготовка і оформлення звітних матеріалів</td></tr>
</table>

8.	Порядок контролю і приймання програмного продукту (моделі)	1. Перевірку ходу розробки комп'ютерної моделі виконувати раз в 3 тижні. 2. Випробування програмного продукту провести відповідно до програми та методики випробувань. 3. Захист розробленого продукту провести на засіданні Атестаційної комісії. 4. Пояснювальну записку подати в електронному вигляді в 1 примірнику.

Виконавець
студент групи КУ-41
БОДНЯ Н. В.
_____

Замовник
к.т.н., доцент
БУЛАВІН Д. О.
_____

Додаток В

Програма і методика випробувань програмного виробу

«СИСТЕМА АВТОМАТИЗОВАНОГО УПРАВЛІННЯ РЕЛЯЦІЙНИМИ БАЗАМИ ДАНИХ ДЛЯ ВИРОБНИЧИХ ПІДПРИЄМСТВ»

1. **Об’єкт випробувань**
   1) Назва програмного виробу: «Система автоматизованого управління реляційними базами даних для виробничих підприємств»
   2) Галузь застосування: управління базами даних
   3) Перераховані відомості запозичуються з відповідних розділів Технічного завдання.
2. **Мета випробувань**

Перевірка відповідності функціональності програмної реалізації системи заявленим функціональним можливостям в технічному завданні (Додаток Б до пояснювальної записки до кваліфікаційної роботи).

3. **Загальні положення**
   1) **Підстави для проведення випробувань**

      Підставою для проведення випробувань є наказ Замовника про призначення атестаційної комісії з оцінки якості кваліфікаційної роботи «Система автоматизованого управління реляційними базами даних для виробничих підприємств».

   2) **Місце і тривалість випробувань**

      Приймальні випробування проводяться у тестовому середовищі в період роботи атестаційної комісії.

   3) **Обсяг випробувань**

      Приймальні випробування програмного виробу охоплюють увесь перелік функціональних сценаріїв та нефункціональних тестів.

   4) **Організації, які беруть участь у випробуваннях**

      Приймальні випробування проводяться атестаційною комісією напередодні засідання за участю Замовника, Виконавця та інших осіб, присутніх на засіданні.

4. **Вимоги до програми або програмного виробу**
   1) Захист даних і прав користувачів: Забезпечувати розмежування доступу за обов'язками.
   2) Чітка структура компонентів: Відокремлена реалізація логіки автентифікації, бізнес-процесів і UI, взаємодія через визначені API-інтерфейси.
   3) Помірні вимоги до обладнання: Підходить для сучасних серверних і робочих станцій, не потребує надмірних ресурсів для коректної роботи в типових умовах.
   4) Захист від некоректних дій користувача: Перевірка всіх вхідних параметрів (формат, довжина, діапазон).
   5) Зручне використання: Інтуїтивна навігація з мінімумом кліків до ключових функцій, валідація форм із наочними повідомленнями про помилки.
   6) Продуктивність: Час відповіді API ≤ 1 с за типового навантаження при одночасної роботи приблизно 500 користувачів.
5. **Вимоги до програмної документації**

Програмною документацією до виробу «Система автоматизованого управління реляційними базами даних для виробничих підприємств» вважати:

1) Опис основних вимог та функціональності системи
2) Опис проєктного рішення (пояснювальна записка)
3) Технічне завдання
4) Текст програми

6. **Засоби і порядок випробувань**

6.1. Засоби випробувань

Для проведення випробувань використовувалися:

1) Інструмент для проведення навантажувального тестування Grafana k6
2) Інструмент для перевірки навантаження Resource Monitor

6.2. Порядок випробувань

Як правило, випробування проводяться в два етапи:

-ознайомчий (1-й етап);

-випробування програмного виробу (2-й етап).

Перелік перевірок, що проводяться на 1 етапі випробувань, включає в себе:

1) Перевірку комплектності програмної документації.
2) Перевірка комплектності складу програмної документації здійснюється за критерієм наявності зазначеної в ТЗ документації.
3) Перевірку комплектності складу технічних і програмних засобів.
4) Методику проведення перевірок на 1 етапі випробувань.
5) Якість програмної документації перевіряється на відповідність вимогам стандартів ДСТУ.

Перелік перевірок, що проводяться на 2 етапі випробувань, включає в себе:

1) перевірку відповідності технічних характеристик програми вимогам технічного завдання;
2) перевірку ступеня виконання функціональних вимог до програми;
3) методику проведення перевірок, що входять до переліку по 2 етапу випробувань.

1) Програма працює відповідно до умов експлуатації операційних систем MS Windows,
2) Для роботи необхідне середовище Node.js версії не нижче 14 та PostgreSQL версії не нижче 12.
3) Порядок проведення випробувань
   a) Запуск сервісів за допомогою команди "npm start".
   b) Після запуску програми необхідно у браузері комп’ютеру перейти за посиланням: http://localhost:3001/.

Для проведення випробувань пропонується тест 1, тест 2, тест 3 та тест 4.

Тест 1. Перевірка забезпечення розмежування доступу за обов’язками

1. Створення нового користувача та присвоєння йому деякої посади.
2. Отримання доступу до тільки тих обов’язків, які належать до цієї посади.

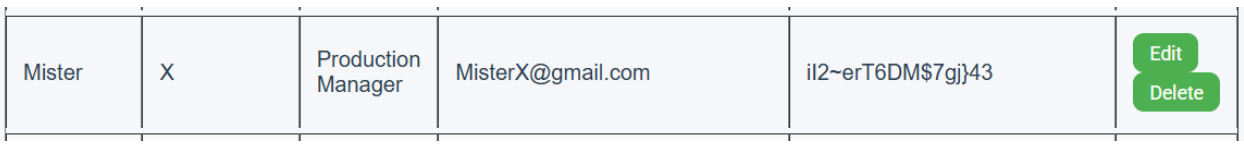


Рисунок В.1 – Тест 1.

Production Manager	Manage production processes, Monitor workers performance	Edit Delete

Рисунок В.2 – Тест 1.

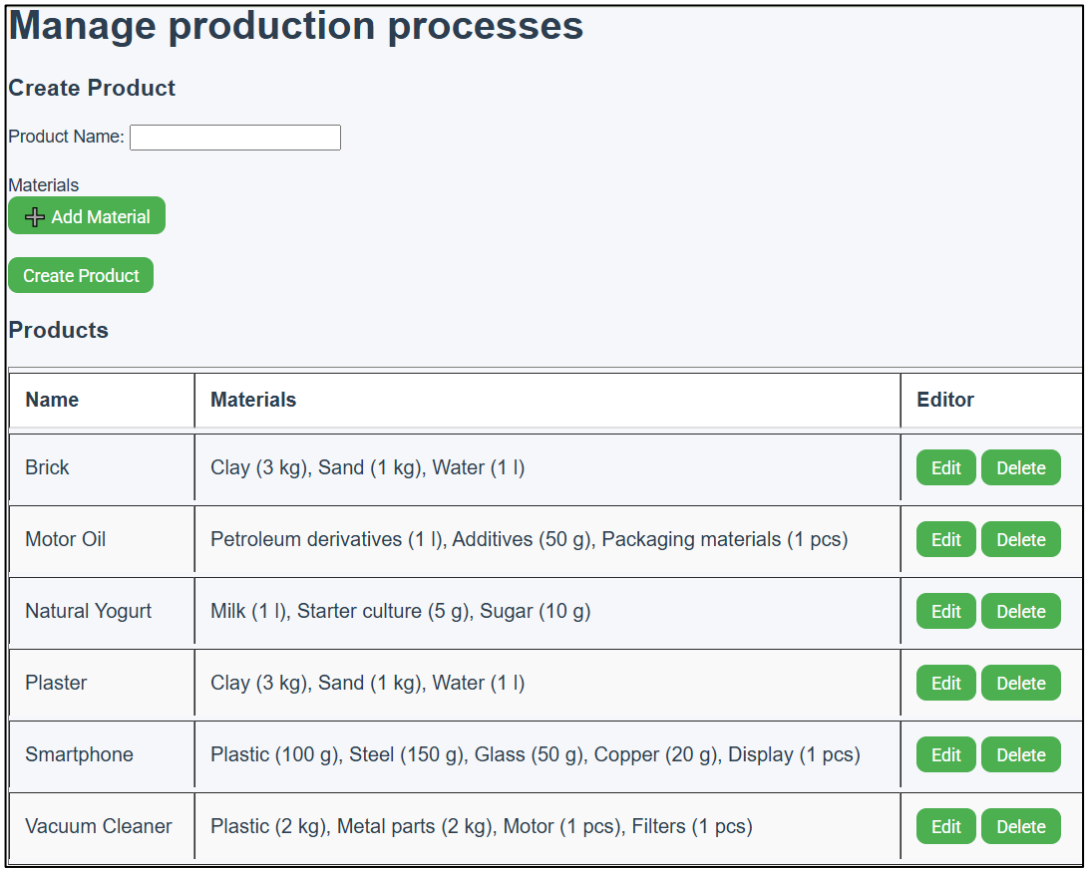


Рисунок В.3 – Тест 1

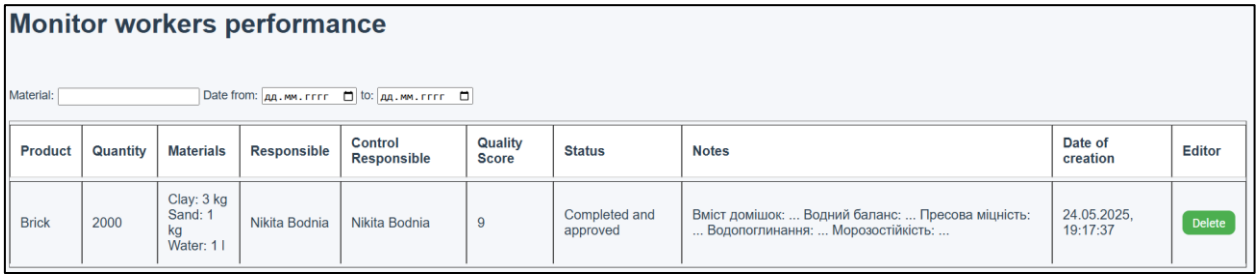


Рисунок В.4 – Тест 1

Тест 2. Помірні вимоги до обладнання

1. Створення трафіку на 500 віртуальних користувачів на 30 секунд.
2. Перевірка завантаженості центрального процесора(2 ядра, 3.0 ГГц) та оперативної пам’яті(8 ГБ) локального сервера(для проходження завантаженість CPU повинна ≤ 50%, Memory ≤ 1 ГБ).

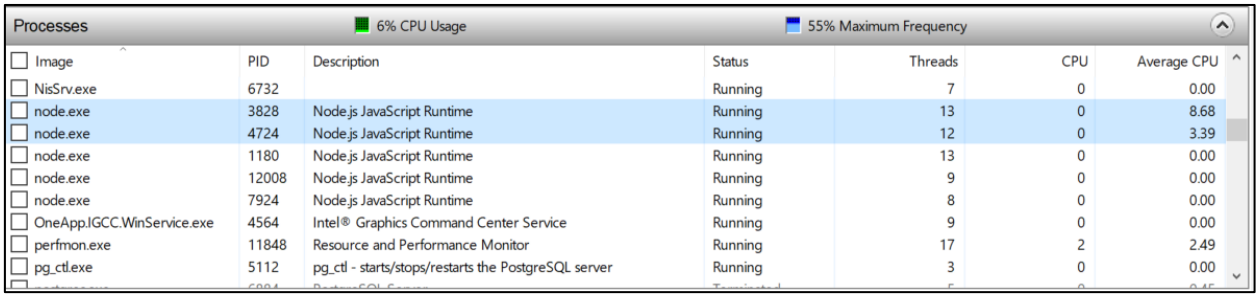


Рисунок В.5 – Тест 2

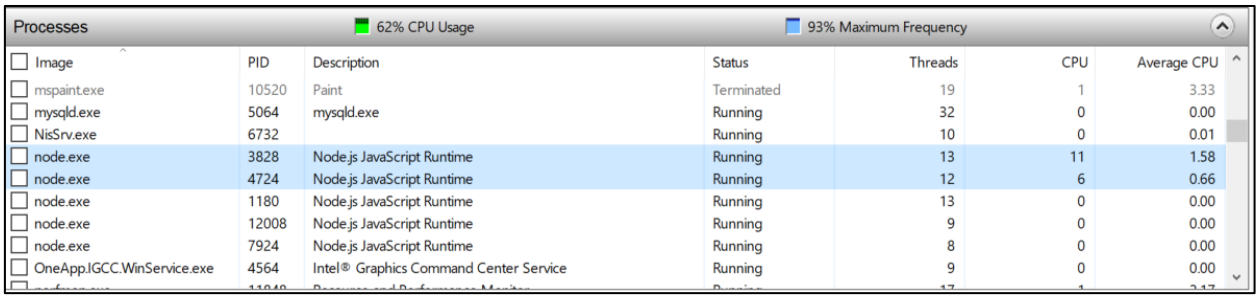


Рисунок В.6 – Тест 2

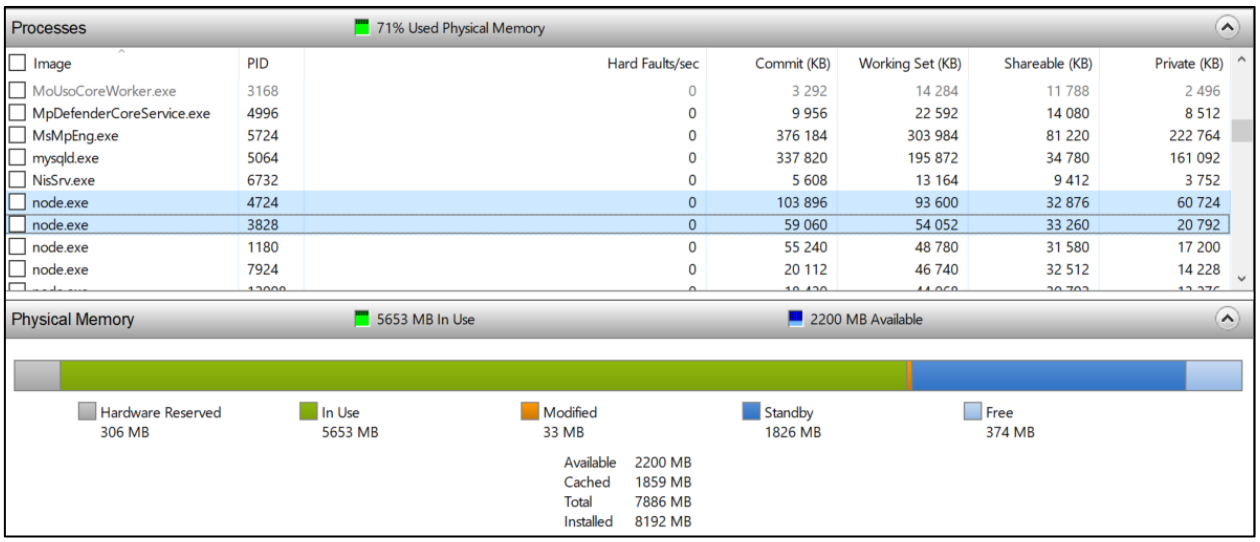


Рисунок В.7 – Тест 2

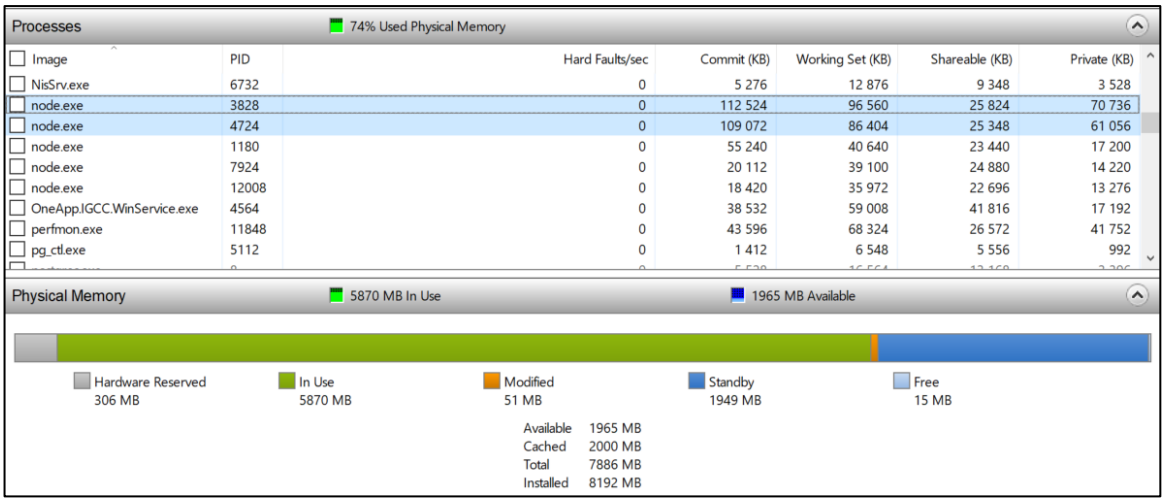


Рисунок В.8 – Тест 2

Тест 3. Захист від некоректних дій користувача

1. Спроба створення або редагування елементів у БД.
2. Видача помилки про незаповнене обов’язкове поле або його некоректне заповнення.

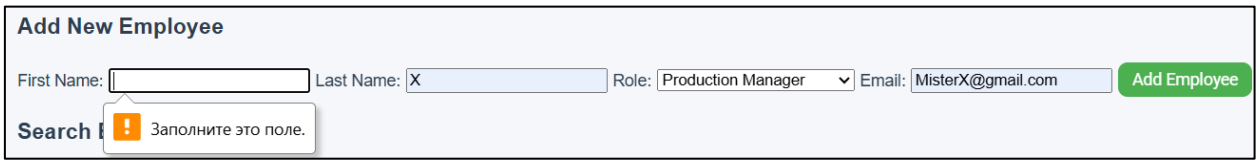


Рисунок В.9 – Тест 3

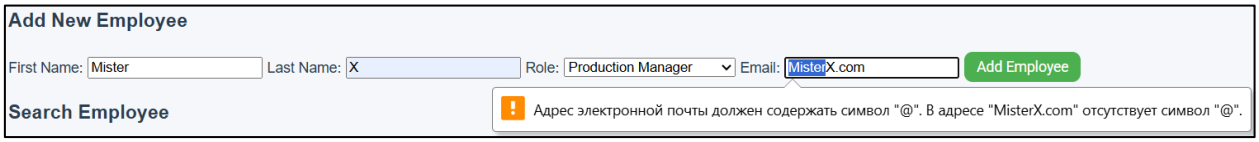


Рисунок В.10 – Тест 3

Рисунок В.11 – Тест 3

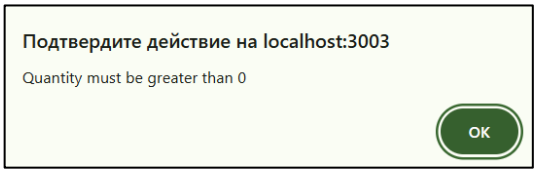


Рисунок В.12 – Тест 3

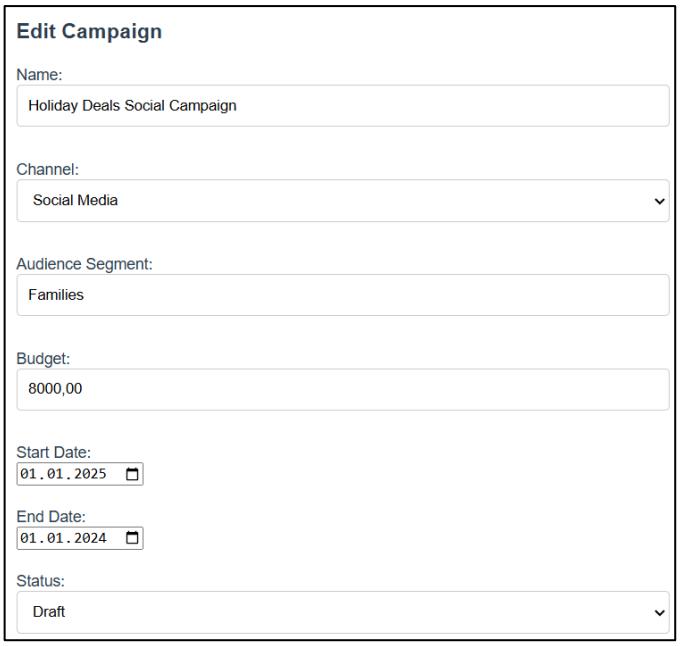


Рисунок В.13 – Тест 3.

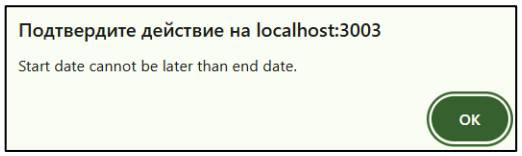


Рисунок В.14 – Тест 3.

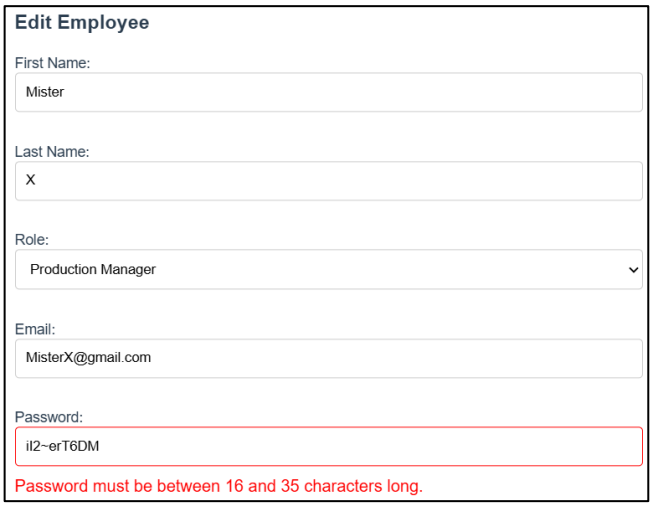


Рисунок В.15 – Тест 3.

Тест 4. Продуктивність

1. Створення трафіку на 500 користувачів.
2. Час відповіді API повинен ≤ 1 с.

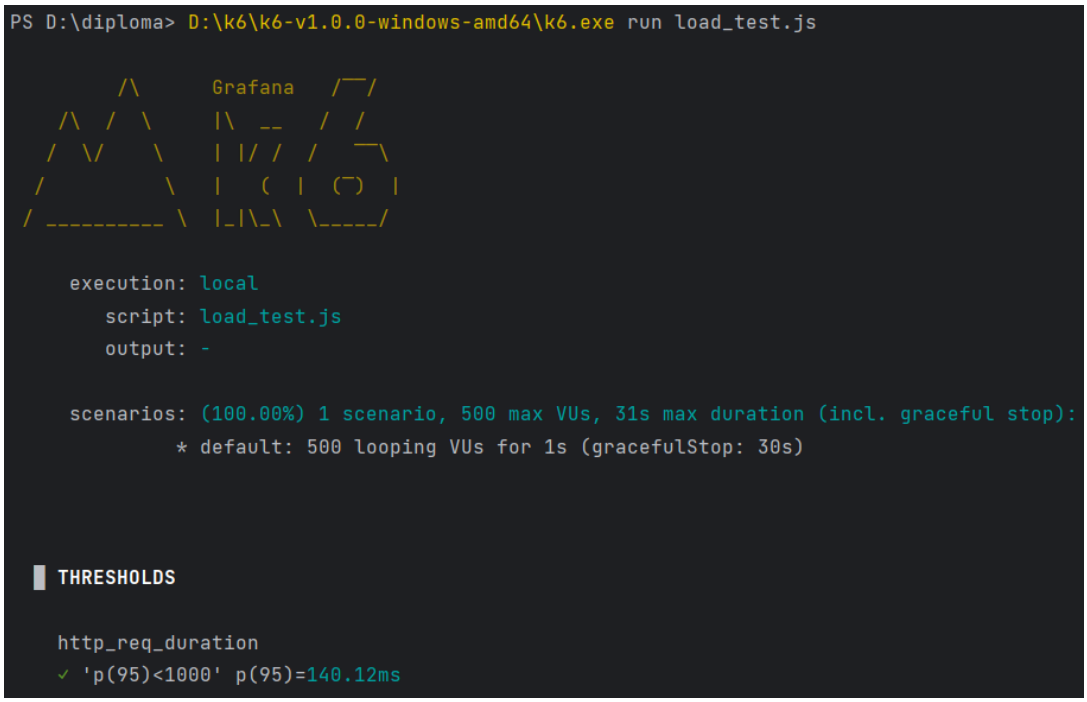


Рисунок В.16 – Тест 4.

Тест вважається пройденим, якщо відбуваються вказані операції або їх відображення у програмному продукті.

Висновки: тест 1 успішно пройшов випробування, тест 2 успішно пройшов випробування, тест 3 успішно пройшов випробування, тест 4 успішно пройшов випробування. Випробування пройшло успішно.

Виконавець: студент групи КУ-41, Бодня Н.В. _____

Додаток Г

Лістинг основного сервісу

```
const express = require('express');
const bodyParser = require('body-parser');
const { Pool } = require('pg');
const app = express();
const PORT = 3003;
const pool = new Pool({
   user: 'postgres',
   host: 'localhost',
   database: 'company_data',
   password: 'kolokol2',
   port: 5432
});
app.use(bodyParser.json());
app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: true }));
app.put('/api/update-quality-status', async (req, res) => {
   try {
      const { id, status, quality_score, notes } = req.body;
      if (!quality_score || isNaN(quality_score) || quality_score < 0 || quality_score > 10) {
         return res.status(400).json({ error: 'Quality score must be between 0 and 10.' });
      }
      if (!notes || notes.trim().length < 5) {
         return res.status(400).json({ error: 'Notes must be at least 5 characters long.' });
      }
      if (status === 'Completed') {
         return res.status(400).json({ error: 'You cannot save an order with status "Completed".' });
      }
      let finalStatus = status;
      if (status === 'Approved') {
         finalStatus = 'Completed and approved';
      }
      const orderData = await pool.query(`
         SELECT
            po.id,
            po.product_id,
            p.name AS product_name,
            po.quantity,
            p.materials,
            ru.first_name || ' ' || ru.last_name AS responsible_user,
            cu.first_name || ' ' || cu.last_name AS control_user
         FROM production_orders po
         JOIN products p ON po.product_id = p.id
         LEFT JOIN users ru ON po.responsible_user_id = ru.id
         LEFT JOIN users cu ON po.responsible_for_control_user_id = cu.id
         WHERE po.id = $1
      `, [id]);
      if (orderData.rows.length === 0) {
         return res.status(404).json({ error: 'Order not found.' });
      }
      const order = orderData.rows[0];
      await pool.query(
         `INSERT INTO workers_performance
         (product_name, quantity, materials, responsible_user, control_user, quality_score, status, notes, created_at)
         VALUES ($1, $2, $3, $4, $5, $6, $7, $8, NOW())`,
         [order.product_name, order.quantity, JSON.stringify(order.materials),
            order.responsible_user, order.control_user, quality_score, finalStatus, notes]
```

```
        );
        if (finalStatus === 'Completed and approved') {
            const { product_id, quantity } = order;
            await pool.query(`
    INSERT INTO finished_goods_inventory (product_id, quantity)
    VALUES ($1, $2)
    ON CONFLICT (product_id)
    DO UPDATE SET quantity = finished_goods_inventory.quantity + EXCLUDED.quantity
  `, [product_id, quantity]);
            await pool.query(
                'DELETE FROM production_orders WHERE id = $1',
                [id]
            );
        } else {
            await pool.query(
                'UPDATE production_orders SET status = $1, notes = $2 WHERE id = $3',
                [finalStatus, notes, id]
            );
        }
        res.status(200).json({ message: 'Quality control updated successfully and saved to performance log.' });
    } catch (error) {
        console.error('Error updating quality control status:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.get('/api/worker-performance', async (req, res) => {
    try {
        const { material, date_from, date_to } = req.query;
        let sql = `
          SELECT id, product_name, quantity,
                 materials::TEXT AS materials,
                 responsible_user, control_user,
                 quality_score, status, notes, created_at
            FROM workers_performance
           WHERE 1=1
        `;
        const params = [];
        if (material) {
            params.push(`%${material}%`);
            sql += ` AND materials::TEXT ILIKE $${params.length}`;
        }
        if (date_from) {
            params.push(date_from);
            sql += ` AND created_at >= $${params.length}`;
        }
        if (date_to) {
            params.push(date_to);
            sql += ` AND created_at <= $${params.length}`;
        }
        sql += ` ORDER BY created_at DESC`;
        const result = await pool.query(sql, params);
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching worker performance data:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.delete('/api/worker-performance/:id', async (req, res) => {
    try {
        const { id } = req.params;
        await pool.query('DELETE FROM workers_performance WHERE id = $1', [id]);
        res.status(200).json({ message: 'Worker performance record deleted successfully' });
```

```
    } catch (error) {
        console.error('Error deleting worker performance record:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.get('/api/quality-control-orders/:userId', async (req, res) => {
    try {
        const { userId } = req.params;
        const result = await pool.query(`
            SELECT
                po.id,
                p.name AS product_name,
                po.quantity,
                po.status,
                po.notes,
                ru.first_name AS responsible_first_name,
                ru.last_name AS responsible_last_name
            FROM production_orders po
            JOIN products p ON po.product_id = p.id
            LEFT JOIN users ru ON po.responsible_user_id = ru.id
            WHERE po.responsible_for_control_user_id = $1 AND po.status = 'Completed'
            ORDER BY po.created_at DESC;
        `, [userId]);
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching quality control orders:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.get('/api/production-orders/user/:userId', async (req, res) => {
    try {
        const { userId } = req.params;
        const result = await pool.query(`
            SELECT
                po.id,
                p.name AS product_name,
                po.quantity,
                jsonb_agg(jsonb_build_object(
                    'name', mat->>'name',
                    'quantity', (mat->>'quantity')::TEXT || ' * ' || po.quantity || ' = ' ||
                    ((regexp_replace(mat->>'quantity', '[^0-9]', '', 'g'))::INTEGER * po.quantity) || ' ' ||
                    regexp_replace(mat->>'quantity', '[0-9]', '', 'g')
                )) AS materials,
                po.status,
                po.notes
            FROM production_orders po
            JOIN products p ON po.product_id = p.id
            LEFT JOIN LATERAL jsonb_array_elements(p.materials) mat ON true
            WHERE po.responsible_user_id = $1
            GROUP BY po.id, p.name, po.quantity, po.status, po.notes
            ORDER BY po.created_at DESC;
        `, [userId]);
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching user production orders with materials:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.put('/api/update-order-notes', async (req, res) => {
    try {
        const { id, notes } = req.body;
        await pool.query(
```

```
            'UPDATE production_orders SET notes = $1 WHERE id = $2',
            [notes, id]
        );
        res.status(200).json({ message: 'Order notes updated successfully' });
    } catch (error) {
        console.error('Error updating order notes:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.put('/api/update-order-status', async (req, res) => {
    try {
        const { id, status } = req.body;
        await pool.query(
            'UPDATE production_orders SET status = $1 WHERE id = $2',
            [status, id]
        );
        res.status(200).json({ message: 'Order status updated successfully' });
    } catch (error) {
        console.error('Error updating order status:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.put('/api/update-order', async (req, res) => {
    try {
        const { id, product_id, quantity, responsible_user_id, responsible_for_control_user_id } = req.body;
        await pool.query(
            'UPDATE production_orders SET product_id = $1, quantity = $2, responsible_user_id = $3,
responsible_for_control_user_id = $4 WHERE id = $5',
            [product_id, quantity, responsible_user_id, responsible_for_control_user_id, id]
        );
        res.status(201).send('Production order updated successfully');
    } catch (error) {
        console.error('Error updating production order:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.get('/api/users', async (req, res) => {
    try {
        const result = await pool.query(`
            SELECT u.id, u.first_name, u.last_name
            FROM users u
            JOIN roles r ON u.role = r.role_name
            WHERE r.responsibilities @> $1
            ORDER BY u.first_name ASC
        `, ['["Perform production"]']);
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching users with Perform Production duty:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.put('/api/update-responsible', async (req, res) => {
    try {
        const { order_id, responsible_user_id } = req.body;
        await pool.query(
            'UPDATE production_orders SET responsible_user_id = $1 WHERE id = $2',
            [responsible_user_id, order_id]
        );
        res.status(201).send('Responsible person updated successfully');
    } catch (error) {
        console.error('Error updating responsible person:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
```

```
    }
});
app.get('/api/products', async (req, res) => {
    try {
        const result = await pool.query('SELECT * FROM products ORDER BY name ASC');
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching products:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.put('/api/products/:id', async (req, res) => {
    try {
        const { id } = req.params;
        const { name, materials } = req.body;
        await pool.query('UPDATE products SET name = $1, materials = $2 WHERE id = $3', [name,
JSON.stringify(materials), id]);
        for (const mat of materials) {
            await pool.query(`INSERT INTO material_costs (material_name, cost_per_unit) VALUES ($1, 0) ON
CONFLICT (material_name) DO NOTHING`, [mat.name]);
        }
        await pool.query(`DELETE FROM material_costs WHERE material_name NOT IN (SELECT DISTINCT
elem->>'name' FROM products, jsonb_array_elements(materials) AS elem)`);
        await pool.query(`
          DELETE FROM warehouse_inventory
           WHERE material_name NOT IN (
             SELECT DISTINCT elem->>'name'
               FROM products, jsonb_array_elements(materials) AS elem
           )
        `);
        res.status(200).json({ message: 'Product updated successfully' });
    } catch (error) {
        console.error('Error updating product:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.post('/api/products', async (req, res) => {
    try {
        const { name, materials } = req.body;
        const exists = await pool.query('SELECT 1 FROM products WHERE name = $1', [name]);
        if (exists.rows.length) {
            return res.status(400).json({ error: 'Product with this name already exists' });
        }
        await pool.query('INSERT INTO products (name, materials) VALUES ($1, $2)', [name,
JSON.stringify(materials)]);
        for (const mat of materials) {
            await pool.query(`INSERT INTO material_costs (material_name, cost_per_unit) VALUES ($1, 0) ON
CONFLICT (material_name) DO NOTHING`, [mat.name]);
        }
        await pool.query(`DELETE FROM material_costs WHERE material_name NOT IN (SELECT DISTINCT
elem->>'name' FROM products, jsonb_array_elements(materials) AS elem)`);
        await pool.query(`
          DELETE FROM warehouse_inventory
           WHERE material_name NOT IN (
             SELECT DISTINCT elem->>'name'
               FROM products, jsonb_array_elements(materials) AS elem
           )
        `);
        res.status(201).json({ message: 'Product created successfully' });
    } catch (error) {
        console.error('Error creating product:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
```

```
    }
});
app.delete('/api/products/:id', async (req, res) => {
    try {
        const { id } = req.params;
        const prodCheck = await pool.query('SELECT 1 FROM products WHERE id = $1', [id]);
        if (!prodCheck.rows.length) {
            return res.status(404).json({ error: 'Product not found' });
        }
        await pool.query('DELETE FROM products WHERE id = $1', [id]);
        await pool.query(`DELETE FROM material_costs WHERE material_name NOT IN (SELECT DISTINCT
elem->>'name' FROM products, jsonb_array_elements(materials) AS elem)`);
        await pool.query(`
          DELETE FROM warehouse_inventory
           WHERE material_name NOT IN (
             SELECT DISTINCT elem->>'name'
               FROM products, jsonb_array_elements(materials) AS elem
           )
        `);
        res.status(201).json('Product deleted and orphan material costs cleaned up');
    } catch (error) {
        console.error('Error deleting product and cleaning material_costs:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.post('/api/start-production', async (req, res) => {
    try {
        const { product_id, quantity, responsible_user_id, responsible_for_control_user_id } = req.body;
        const productQuery = await pool.query('SELECT * FROM products WHERE id = $1', [product_id]);
        if (productQuery.rows.length === 0) {
            return res.status(404).json({ error: 'Product not found' });
        }
        const product = productQuery.rows[0];
        let materials = Array.isArray(product.materials) ? product.materials : JSON.parse(product.materials);
        for (const material of materials) {
            const inventoryQuery = await pool.query('SELECT * FROM warehouse_inventory WHERE material_name
= $1', [material.name]);
            if (inventoryQuery.rows.length === 0) {
                return res.status(400).json({ error: `Material ${material.name} is not available in warehouse` });
            }
            const currentQuantity = inventoryQuery.rows[0].quantity;
            const requiredQuantity = parseInt(material.quantity) * quantity;
            if (currentQuantity < requiredQuantity) {
                const existingRequest = await pool.query(
                    `SELECT * FROM purchase_requests WHERE material_name = $1 AND status = 'Pending'`,
                    [material.name]
                );
                if (existingRequest.rows.length > 0) {
                    const currentRequestQuantity = existingRequest.rows[0].quantity;
                    const newQuantity = currentRequestQuantity + requiredQuantity;
                    await pool.query(
                        `UPDATE purchase_requests SET quantity = $1 WHERE id = $2`,
                        [newQuantity, existingRequest.rows[0].id]
                    );
                } else {
                    const supplierQuery = await pool.query('SELECT id FROM suppliers LIMIT 1');
                    const supplier_id = supplierQuery.rows[0].id;
                    await pool.query(
                        `INSERT INTO purchase_requests (material_name, quantity, supplier_id, status)
                         VALUES ($1, $2, $3, 'Pending')`,
                        [material.name, requiredQuantity, supplier_id]
                    );
```

```
            }
        }
    }
    await pool.query(
        `INSERT INTO production_orders
        (product_id, quantity, responsible_user_id, responsible_for_control_user_id, status)
        VALUES ($1, $2, $3, $4, $5)`,
        [product_id, quantity, responsible_user_id, responsible_for_control_user_id, 'Preparing']
    );
    for (const material of materials) {
        const inventoryQuery = await pool.query('SELECT * FROM warehouse_inventory WHERE material_name
= $1', [material.name]);
        const currentQuantity = inventoryQuery.rows[0].quantity;
        const requiredQuantity = parseInt(material.quantity) * quantity;
        const newQuantity = currentQuantity - requiredQuantity;
        await pool.query(
            `UPDATE warehouse_inventory SET quantity = $1 WHERE material_name = $2`,
            [newQuantity, material.name]
        );
    }
    res.status(201).json({ message: 'Production started successfully' });
  } catch (error) {
    console.error('Error starting production:', error);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
app.get('/api/production-orders', async (req, res) => {
  try {
    const result = await pool.query(`
        SELECT
            po.id,
            p.name AS product_name,
            po.quantity,
            po.status,
            po.notes,
            po.created_at,
            po.responsible_user_id,
            po.responsible_for_control_user_id,
            ru.first_name AS responsible_first_name,
            ru.last_name AS responsible_last_name,
            cu.first_name AS control_first_name,
            cu.last_name AS control_last_name
        FROM production_orders po
        JOIN products p ON po.product_id = p.id
        LEFT JOIN users ru ON po.responsible_user_id = ru.id
        LEFT JOIN users cu ON po.responsible_for_control_user_id = cu.id
        ORDER BY po.created_at DESC;
    `);
    res.json(result.rows);
  } catch (error) {
    console.error('Error fetching production orders:', error.message);
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
app.get('/api/control-users', async (req, res) => {
  try {
    const result = await pool.query(`
        SELECT u.id, u.first_name, u.last_name
        FROM users u
        JOIN roles r ON u.role = r.role_name
        WHERE r.responsibilities @> $1
        ORDER BY u.first_name ASC
```

```
      `, [["Control product quality"]]);
      res.json(result.rows);
    } catch (error) {
      console.error('Error fetching control users:', error.message);
      res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.delete('/api/production-orders/:id', async (req, res) => {
  try {
    const { id } = req.params;
    const orderData = await pool.query(`SELECT p.materials, po.quantity FROM production_orders po JOIN
products p ON po.product_id = p.id WHERE po.id = $1
    `, [id]);
    if (orderData.rows.length === 0) {
      return res.status(404).json({ error: 'Production order not found' });
    }
    const { materials, quantity: orderQty } = orderData.rows[0];
    const mats = Array.isArray(materials) ? materials : JSON.parse(materials);
    for (const mat of mats) {
      const perUnit = parseFloat(mat.quantity);
      if (isNaN(perUnit)) {
        console.warn(`Cannot parse quantity "${mat.quantity}" for material ${mat.name}`);
        continue;
      }
      const returnQty = perUnit * orderQty;
      const invRes = await pool.query(
        'SELECT quantity FROM warehouse_inventory WHERE material_name = $1',
        [mat.name]
      );
      if (invRes.rows.length === 0) {
        await pool.query(
          `INSERT INTO warehouse_inventory (material_name, quantity)
         VALUES ($1, $2)`,
          [mat.name, returnQty]
        );
      } else {
        const currentQty = invRes.rows[0].quantity;
        const newQty = currentQty + returnQty;
        await pool.query(
          `UPDATE warehouse_inventory SET quantity = $1 WHERE material_name = $2`,
          [newQty, mat.name]
        );
      }
    }
    await pool.query('DELETE FROM production_orders WHERE id = $1', [id]);
    res.status(200).json({
      message: 'Production order deleted and materials returned to warehouse successfully'
    });
  } catch (error) {
    console.error(
      'Error deleting production order and updating warehouse inventory:',
      error.message
    );
    res.status(500).json({ error: 'Internal Server Error' });
  }
});
app.delete('/api/purchase-history/:id', async (req, res) => {
  try {
    const { id } = req.params;
    const historyCheck = await pool.query('SELECT * FROM purchase_history WHERE id = $1', [id]);
    if (historyCheck.rows.length === 0) {
      return res.status(404).json({ error: 'Purchase history record not found' });
```

```
        }
        await pool.query('DELETE FROM purchase_history WHERE id = $1', [id]);
        res.status(200).json({ message: 'Purchase history record deleted successfully' });
    } catch (error) {
        console.error('Error deleting purchase history record:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.get('/api/control-users-supplies', async (req, res) => {
    try {
        const result = await pool.query(`
            SELECT u.id, u.first_name, u.last_name
            FROM users u
            JOIN roles r ON u.role = r.role_name
            WHERE r.responsibilities @> $1
            ORDER BY u.first_name ASC
        `, ['["Procure quality control"]']);
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching control users:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.get('/api/purchase-requests/:userId', async (req, res) => {
    try {
        const { userId } = req.params;
        const result = await pool.query(`
            SELECT pr.id, pr.material_name, pr.quantity, pr.supplier, pr.status, pr.notes, pr.defect_report, pr.created_at
            FROM purchase_requests pr
            JOIN suppliers s ON pr.supplier_id = s.id
            WHERE s.control_user_id = $1
            ORDER BY pr.created_at DESC;
        `, [userId]);
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching purchase requests for control user:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.put('/api/update-defect-report', async (req, res) => {
    try {
        const { id, defect_report } = req.body;
        if (!defect_report || defect_report.trim().length < 5) {
            return res.status(400).json({ error: 'Defect report must be at least 5 characters long.' });
        }
        await pool.query(
            'UPDATE purchase_requests SET defect_report = $1 WHERE id = $2',
            [defect_report, id]
        );
        res.status(200).json({ message: 'Defect report updated successfully' });
    } catch (error) {
        console.error('Error updating defect report:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.get('/api/purchase-requests', async (req, res) => {
    try {
        const result = await pool.query(`
            SELECT id, material_name, quantity, supplier, status, notes, defect_report, created_at
            FROM purchase_requests
            ORDER BY created_at DESC;
        `);
```

```
        res.json(result.rows);
    } catch (error) {
        console.error('Error fetching purchase requests:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.post('/api/purchase-requests', async (req, res) => {
    try {
        const { material_name, quantity, supplier_id, notes } = req.body;
        const status = 'Pending';
        const supplierQuery = await pool.query('SELECT name, control_user_id FROM suppliers WHERE id = $1',
[supplier_id]);
        if (supplierQuery.rows.length === 0) {
            return res.status(400).json({ error: 'Supplier not found' });
        }
        const supplier_name = supplierQuery.rows[0].name;
        const control_user_id = supplierQuery.rows[0].control_user_id;
        await pool.query(
            `INSERT INTO purchase_requests (material_name, quantity, supplier, supplier_id, status, notes)
             VALUES ($1, $2, $3, $4, $5, $6)`,
            [material_name, quantity, supplier_name, supplier_id, status, notes]
        );
        res.status(201).json({ message: 'Purchase request created successfully' });
    } catch (error) {
        console.error('Error creating purchase request:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.put('/api/purchase-requests/:id', async (req, res) => {
    try {
        const { id } = req.params;
        const { material_name, quantity, supplier_id, notes } = req.body;
        const supplierQuery = await pool.query('SELECT name FROM suppliers WHERE id = $1', [supplier_id]);
        if (supplierQuery.rows.length === 0) {
            return res.status(400).json({ error: 'Supplier not found' });
        }
        const supplier_name = supplierQuery.rows[0].name;
        await pool.query(
            `UPDATE purchase_requests
             SET material_name = $1, quantity = $2, supplier = $3, supplier_id = $4, notes = $5
             WHERE id = $6`,
            [material_name, quantity, supplier_name, supplier_id, notes, id]
        );
        res.status(200).json({ message: 'Purchase request updated successfully' });
    } catch (error) {
        console.error('Error updating purchase request:', error.message);
        res.status(500).json({ error: 'Internal Server Error' });
    }
});
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

Посилання на повний лістинг коду:

https://github.com/NELEON256/Diploma/tree/c9ac8bc484d20ffda6fb1d171897426e6d302675