МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ПАВЛОВ ДМИТРО ЛЕОНІДОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 20__ р.

РОЗРОБКА CRM СИСТЕМИ ДЛЯ КОЛОРИСТІВ З ВИКОРИСТАННЯМ NoSQLDateBase

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
І. В. Богач, доцент кафедри
інформаційних технологій,
к.т.н., доцент
_____

Оцінка:____/ ____/ _____

(бали за шкалою СКТС/за національною шкалою)

Голова ЕК: _____

(підпис)

Вінниця 2024

АНОТАЦІЯ

Павлов Д.Л. Розробка CRM Системи для колористів з використанням NoSqlDateBase. Спеціальність 122 «Комп’ютерні науки», освітня програма «Комп’ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2024.

У кваліфікаційній (бакалаврській) роботі досліджено та проаналізовано поняття CRM системи, NoSQL баз Даних,їх роль в сучасних веб застосунках та переваги, а також надано практичний приклад застосування Firebase NoSQL бази даних.

Ключові слова: CRM система, бази даних, NoSQL, Firebase,

75 ст, 29 рис., 3 дод., 36 джерел.

ABSTRACT

Pavlov D.L. Development of CRM System for Colorists Using NoSQL Database. Specialty 122 "Computer Science", educational program "Computer Science". Vasyl Stus Donetsk National University, Vinnytsia 2024.

The bachelor's thesis explores and analyzes the concept of CRM systems, NoSQL databases, their role in modern web applications, and advantages, as well as provides a practical example of using Firebase NoSQL database.

Keywords: CRM system, databases, NoSQL, Firebase.

ЗМІСТ

ВСТУП

Актуальність теми дослідження. У сучасному світі, де конкуренція на ринку послуг надзвичайно висока, індивідуальні підходи до клієнтів стають визначальними для успіху будь-якого бізнесу. Спеціалізовані професії, такі як колористи у сфері дизайну та мистецтва, не виключення. Для їхнього успіху критично важливо не лише навички та талант, але й ефективне керування клієнтськими відносинами. [1-5]

CRM-системи (Customer Relationship Management) допомагають у впорядкуванні та автоматизації процесів взаємодії з клієнтами, що дозволяє забезпечити індивідуальний підхід та високу якість обслуговування. Для колористів, які працюють у галузі підбору фарб для автомобілів, важливо мати доступ до точної інформації про клієнтів, історію їх замовлень, специфічні вимоги та побажання. Це дозволяє швидко і якісно надавати послуги, що значно підвищує конкурентоспроможність на ринку.

З використанням NoSQL баз даних у CRM-системах відкриваються нові можливості для зберігання та обробки великих обсягів неструктурованих даних, що є особливо актуальним для колористів. Такі бази даних, як MongoDB або Firebase, дозволяють ефективно працювати з різноманітними даними, що надходять від клієнтів, а також з інформацією про різні фарби та їх комбінації. Це забезпечує гнучкість і масштабованість системи, що є важливим для обслуговування великої кількості клієнтів та обробки великих обсягів інформації. [1-9]

З впровадженням сучасних технологій у CRM-системи, таких як штучний інтелект та машинне навчання, можливе автоматичне аналізування даних для прогнозування потреб клієнтів і надання рекомендацій щодо вибору фарб. Це сприяє підвищенню ефективності роботи колористів, зменшенню часу на підбір оптимальних рішень та покращенню задоволеності клієнтів.

Враховуючи вищезазначене, актуальною є розробка CRM-системи для колористів з використанням NoSQL баз даних. Це дозволить підвищити рівень обслуговування клієнтів, оптимізувати процеси управління замовленнями та забезпечити індивідуальний підхід до кожного клієнта, що є ключовим фактором успіху у сучасному конкурентному середовищі. [1, 2, 7, 8]

Метою дипломної роботи є розробка та впровадження CRM системи, спеціально адаптованої для потреб колористів, що відповідає потребам сучасного бізнесу та вимагає швидкого, масштабованого та надійного рішення для керування клієнтами.

Для виконання поставленої мети потрібно виконати наступні **завдання дослідження:**

1. Визначити актуальність даної системи та дослідити сучасні технології для вирішення поставленої задачі.
2. Спроектувати архітектуру бази даних та архітектуру CRM системи.
3. Розробити CRM систему на основі обраних технологій
4. Протестувати працездатність виконаної роботи та довести виконання поставленої мети.

Об’єктом дослідження є процеси проектування та реалізації програмних продуктів, а саме CRM систем.

Предмет дослідження: програмні засоби для проектування та реалізації програмних продуктів, а саме розробки CRM системи для оптимізації робочих процесів колористів.

Теоретичне та практичне значення одержаних результатів полягає в розробці та впровадженні CRM системи, адаптованої для потреб колористів, що базується на технології NoSqlDatabase, конкретно Firebase та відповідає потребам сучасного бізнесу та вимагає швидкого, масштабованого та надійного рішення для керування клієнтами.

Використання технології NoSqlDatabase, зокрема Firebase, дозволяє створювати високоефективні та гнучкі CRM системи, які можуть легко

адаптуватися до змінних потреб бізнесу. Firebase забезпечує швидку інтеграцію з іншими сервісами Google та забезпечує безпеку та масштабованість даних.

Розробка імплементованої CRM системи допоможе колористам ефективно керувати клієнтськими відносинами, використовуючи передові технології баз даних. Така система дозволить оптимізувати процеси взаємодії з клієнтами, збільшити їх задоволеність та забезпечить більшу ефективність у роботі. А також дослідження ролі сучасної технології NoSQL Firebase.

Апробація результатів дослідження

Результати роботи було представлено на Всеукраїнській науково-практичній Інтернет-конференції студентів, аспірантів та молодих науковців «Молодь в науці: Дослідження, проблеми, перспективи (МН-2024)» [27].

Структура кваліфікаційної (бакалаврської) роботи складається зі вступу, трьох розділів, висновку, списку використаних посилань, 29 рисунків. Загальний обсяг основної частини роботи – 55 сторінок.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Поняття CRM системи

Визначення та призначення CRM системи

CRM (Customer Relationship Management) система є стратегічним та технологічним підходом до керування взаємовідносинами з клієнтами. Вона орієнтована на забезпечення ефективної комунікації та взаємодії з клієнтами з метою збільшення їх задоволеності, лояльності та збільшення обсягів продажів.

Основними Функціями CRM системи є:

1. Збір та збереження інформації про клієнтів: контактні дані, історія взаємодії, попередні покупки тощо.

2. Аналіз даних про клієнтів для виявлення їхніх потреб, попередніх та потенційних угод.

3. Управління контактами та комунікацією з клієнтами через різноманітні канали зв'язку.

4. Автоматизація процесів продажів, маркетингу та обслуговування клієнтів.

5. Побудова стратегій розвитку взаємовідносин з клієнтами та збереження лояльності.

Поняття CRM системи з'явилося в кінці 20-го століття, коли бізнеси розпочали усвідомлювати важливість підтримки довгострокових та взаємовигідних відносин з клієнтами. Перші CRM системи були базовими базами даних, що містили інформацію про клієнтів та їхні контакти.

Протягом наступних десятиліть CRM системи еволюціонували від простих баз даних до комплексних платформ з великою кількістю функціональних можливостей. З поширенням Інтернету та цифрових технологій в останні десятиліття CRM системи отримали новий імпульс розвитку. Завдяки Інтернету з'явилися нові канали комунікації з клієнтами, такі як електронна пошта,

соціальні медіа та онлайн-чати, що вимагають інтеграції з CRM системами для ефективного управління взаємодією з клієнтами через ці канали.

Також важливим етапом став розвиток мобільних технологій, що дозволяють клієнтам взаємодіяти з компаніями в будь-який час та з будь-якого місця. Відповідно, сучасні CRM системи повинні бути мобільними та адаптованими до різних типів пристроїв, що використовуються клієнтами.

Сьогоднішні CRM системи використовують передові технології штучного інтелекту, аналітики даних, хмарних сервісів та мобільних додатків для забезпечення найвищого рівня ефективності та зручності для користувачів.

Основна ціль та мотивація для створення

Основною ціллю створення CRM систем є забезпечення компаніям можливості ефективно управляти своїми клієнтськими відносинами, збільшувати їхню лояльність та сприяти зростанню обсягів продажів. Це досягається шляхом покращення комунікації з клієнтами, персоналізації підходу до них та вчасної реакції на їхні потреби та запити.

Мотивація для створення CRM систем виникла з усвідомлення того, що клієнти, які відчуваються цінними та підтриманими, більш схильні до повторних покупок та рекомендацій інших потенційних клієнтів. Тому бізнесам вигідно інвестувати у системи, які допомагають підтримувати позитивні взаємовідносини з клієнтами.

Крім того, CRM системи дозволяють збирати величезний обсяг даних про клієнтів, що може бути використаний для аналізу тенденцій ринку, прогнозування попиту та розробки стратегій розвитку бізнесу.

Методології та підходи до CRM

Існує кілька методологій та підходів до впровадження CRM систем. Один з найпоширеніших - це підхід з фокусом на клієнта (customer-centric approach). Цей підхід передбачає, що всі дії бізнесу, включаючи маркетинг, продажі та обслуговування клієнтів, повинні бути орієнтовані на задоволення потреб та очікувань клієнтів.

Інші підходи включають технологічний підхід, який акцентується на використанні передових інформаційних технологій для автоматизації процесів та підвищення ефективності взаємодії з клієнтами; стратегічний підхід, який покладає акцент на інтеграцію CRM зі стратегічними цілями та бізнес-процесами компанії; та аналітичний підхід, який використовує аналіз даних для виявлення паттернів та трендів у поведінці клієнтів.

1.2 Огляд існуючих CRM систем

Задля розуміння, які основні можливості слід використовувати під час написання CRM системи, слід звернути увагу на “гігантів” в цій сфері, підкреслити їх основні фішки, можливості використання їх систем, способи взаємодії з клієнтом та варіативність функціоналу.

Salesforce - одна з найпопулярніших CRM систем у світі. Вона визначається своєю гнучкістю, широким функціоналом та масштабованістю. Salesforce вперше була випущена в 1999 році й стала першою CRM системою, яка повністю базується на хмарних технологіях. Основними її фішками є автоматизація продажів, маркетингу та обслуговування клієнтів, а також велика кількість додаткових модулів та інтеграція зі сторонніми сервісами.

Microsoft Dynamics 365 є іншою популярною CRM системою, що відрізняється відмінною інтеграцією з іншими продуктами Microsoft та високою ступенем настроюваності. Вона випущена в 2016 році та пропонує рішення для продажів, маркетингу, обслуговування клієнтів, фінансів та операцій.

HubSpot - це CRM система, спрямована на невеликі та середні бізнеси. Вона відзначається інтуїтивним інтерфейсом, високим рівнем автоматизації маркетингу та продажів, а також безкоштовним базовим планом для користувачів.

Кожна ця CRM система має певні свої недоліки та переваги. Багато плюсів ми можемо використати під час розробки CRM системи задля кращого досвіду користувачів. Й саме унікальність роботи полягає у тому, що загальновідомих та

популярних систем для колористів не існує. Усі представлені системи вище є більш абстрактними.

1.3 Поняття NoSQL баз даних

NoSQL бази даних - це клас баз даних, які відмінні від традиційних реляційних баз даних (SQL) за своєю структурою та моделлю даних. Основна сутність NoSQL баз даних полягає в тому, що вони призначені для зберігання та обробки великих обсягів даних, що змінюються динамічно та не мають чіткої структури.[1]

Основним мотивом створення NoSQL баз даних було потреба в розробці систем, які можуть ефективно опрацьовувати та зберігати великі обсяги неструктурованих даних, таких як дані з соціальних мереж, логи серверів, документи HTML, графові дані тощо. Традиційні реляційні бази даних мали обмеження у роботі з такими типами даних. Кожен тип NoSQL баз даних має свої особливості, переваги та недоліки. Вибір конкретного типу залежить від вимог проекту та особливостей даних, що будуть зберігатися. [2]

Основні поняття

1. Гнучкість: NoSQL бази даних дозволяють зберігати дані без чіткої структури та змінювати їх схему без перерви у роботі системи.

2. Горизонтальне масштабування: NoSQL бази даних легко масштабуються горизонтально, тобто додаються нові сервери для збільшення продуктивності та об'єму даних.

3. Висока доступність: Багато NoSQL систем підтримують реплікацію даних, що забезпечує високу доступність та надійність системи. [3]

Історія

Термін "NoSQL" з'явився у 1998 році, коли Карлогу Геманн використовав його для опису бази даних, яка зберігала XML документи без використання SQL. Проте справжній популярності NoSQL базам даних надав вибух Big Data в 2000-х роках та виросла потреба в зберіганні та обробці великих обсягів даних.

Варіативність

NoSQL бази даних поділяються на кілька основних типів:

1. Ключ-Значення (Key-Value): Вони зберігають дані у вигляді пар ключ-значення ідеально підходять для кешування та сесійних даних.

2. Документні (Document-Oriented): Дозволяють зберігати дані у вигляді документів, таких як JSON або XML, ідеально підходять для сховищ документів та контенту.

3. Стовпчасті (Column-Family): Зберігають дані у вигляді колонок замість рядків, що робить їх ефективними для аналітичних запитів та широкої кількості колонок.

4. Графові (Graph): Призначені для роботи з графовими структурами даних, такими як соціальні мережі або мережі передачі даних. [4]

1.4 Основи роботи з NoSQL базами даних

NoSQL бази даних відрізняються від традиційних реляційних баз даних своєю структурою та моделлю даних. Для ефективної роботи з NoSQL базами даних необхідно розуміти їх особливості, методи запитів та загальний підхід до роботи з ними.

Для чого використовують певні методи роботи з NoSQL:

1. Методи Роботи з Документами:

Документ-орієнтовані бази даних використовують методи роботи з документами, такі як додавання, оновлення та видалення документів. Ці методи дозволяють зберігати та організовувати неструктуровані дані у форматі документів (наприклад, JSON або XML). [5]

2. Методи Роботи з Ключами та Значеннями:

Ключ-значення бази даних використовують методи роботи з ключами та значеннями, такі як додавання, отримання та видалення значень за певним ключем. Ці методи дозволяють ефективно зберігати та отримувати доступ до даних за допомогою унікального ключа. [6]

3. Методи Роботи з Колонками:

Стовпчасті бази даних використовують методи роботи з колонками, такі як додавання, оновлення та видалення колонок. Ці методи дозволяють ефективно зберігати та обробляти великі обсяги даних за допомогою колонкової структури. [7]

4. Методи Роботи з Графами:

Графові бази даних використовують методи роботи з графами, такі як додавання, зміна та видалення вершин та ребер графа. Ці методи дозволяють зберігати та оптимізовувати дані, що мають складну графову структуру.

Основні Методи Запитів до NoSQL Баз Даних

Додавання Документів/Записів: Метод додає новий документ або запис до бази даних.

Оновлення Документів/Записів: Метод оновлює існуючий документ або запис у базі даних.

Видалення Документів/Записів: Метод видаляє документ або запис з бази даних.

Отримання Документів/Записів за Ключем або Умовою: Метод отримує документи або записи за певним ключем або за умовою. [8-10]

Запити з Об'єднанням та Агрегацією: Деякі NoSQL бази даних підтримують запити з об'єднанням та агрегацією, аналогічні до SQL запитів.

Загальний підхід роботи з NoSQL базами даних

Аналіз Вимог: Спочатку необхідно чітко зрозуміти вимоги проекту та типи даних, які будуть зберігатися в базі даних.

Вибір Відповідного Типу Бази Даних: Вибір конкретного типу NoSQL бази даних залежить від потреб проекту та типів даних.

Розробка Схеми Даних: Після вибору бази даних необхідно розробити схему даних, яка відповідає вимогам проекту.

Імплементація та Оптимізація Запитів: Згодом слід реалізувати запити до бази даних та оптимізувати їх для максимальної продуктивності та ефективності.

Тестування та Моніторинг: Після реалізації бази даних важливо провести тестування та моніторинг її продуктивності та надійності.

Розуміння основних методів роботи з NoSQL базами даних допоможе розробникам ефективно використовувати їх для зберігання та обробки різноманітних типів даних.

1.5 Особливості використання NoSQL бази даних у розробці CRM системи

NoSQL бази даних можуть бути корисним інструментом у розробці CRM системи, оскільки вони дозволяють ефективно зберігати та обробляти великі обсяги даних, що змінюються динамічно та не мають чіткої структури. Ось детальні особливості використання NoSQL баз даних у розробці CRM системи:

1.Гнучкість в схемі даних:

Однією з основних переваг NoSQL баз даних є їх гнучкість в схемі даних. У розробці CRM системи, де структура даних може змінюватися від клієнта до клієнта, ця гнучкість дозволяє легко адаптувати базу даних до різних потреб.

2.Масштабованість:

Багато NoSQL баз даних мають вбудовану підтримку горизонтального масштабування, що дозволяє легко розширювати систему з ростом обсягів даних та навантаження.

3.Швидкодія:

Деякі типи NoSQL баз даних, такі як ключ-значення та стовпчасті, забезпечують швидкий доступ до даних. Це особливо важливо для CRM систем, де часто вимагається миттєва відповідь на запити користувачів.

4. Підтримка великих обсягів даних:

NoSQL бази даних оптимізовані для роботи з великими обсягами даних, що робить їх ідеальними для CRM систем, де може бути велика кількість клієнтів та транзакцій.

5. Гнучкість в моделі даних:

Різні типи NoSQL баз даних підтримують різні моделі даних, такі як документ-орієнтована, ключ-значення, стовпчаста та графова. Це дозволяє розробникам вибирати найбільш підходящий тип для конкретного випадку використання.

6. Можливість реплікації та резервного копіювання:

Багато NoSQL баз даних підтримують реплікацію та резервне копіювання, що забезпечує високу доступність та надійність даних у випадку виникнення проблем.

7.Відкритість для Розширення:

NoSQL бази даних зазвичай мають відкрите API та підтримують різноманітні інструменти для розширення та налаштування, що дозволяє розробникам створювати спеціалізовані рішення для своїх потреб.

8. Аналітика та машинне навчання:

Деякі NoSQL бази даних мають вбудовану підтримку для аналітики та машинного навчання, що дозволяє використовувати дані CRM системи для отримання цінної інформації та прогнозування трендів.

Розуміння цих особливостей дозволяє ефективно використовувати NoSQL бази даних у розробці CRM системи, що сприяє покращенню їхньої продуктивності та функціональності.

Однак важливо пам'ятати, що вибір конкретної NoSQL бази даних повинен бути зроблений з урахуванням конкретних потреб проекту та характеристик даних, що будуть зберігатися. Деякі бази даних можуть бути більш підходящими для певних сценаріїв використання, ніж інші, тому важливо провести детальний аналіз та вибрати оптимальний варіант для своєї CRM системи.

РОЗДІЛ 2 ПРОЕКТУВАННЯ СИСТЕМИ ТА ВИБІР КОМПОНЕНТІВ

2.1 Актуальність та огляд NoSQL Firebase

Під час підготовки до розробки застосунку постало питання у вибору бази даних. В мене було декілька варіантів проте я зупинився саме на NoSQL базі даних Firebase. [11]

Firebase - це платформа для розробки мобільних та веб-додатків, яка включає в себе різноманітні сервіси, включаючи NoSQL базу даних, хостинг, аутентифікацію, зберігання файлів та інші. Firebase був створений в 2011 році як незалежний стартап, а потім був придбаний Google у 2014 році. Однією з головних мотивацій створення Firebase була потреба в зручних та ефективних інструментах для розробки застосунків, які б забезпечували швидкий розвиток та простоту використання.[12]

Сьогодні Firebase є однією з найпопулярніших платформ для розробки мобільних та веб-додатків, оскільки вона пропонує широкий спектр функцій та послуг, які забезпечують ефективний розвиток та управління додатками.

Firebase набув значної популярності серед розробників через свою простоту використання, гнучкість та широкий спектр функцій. За останні роки Firebase став ключовою платформою для розробки мобільних та веб-додатків для мільйонів розробників по всьому світу.

Одним із головних мотивів вибору Firebase є його NoSQL база даних, яка пропонує зручний та ефективний спосіб зберігання даних для мобільних та веб-додатків. Firebase Realtime Database - це NoSQL база даних, яка забезпечує можливість синхронізації даних в реальному часі між різними клієнтами та пристроями.[13]

Firebase Realtime Database пропонує наступні особливості:

1.Реальний час:

Дані в Firebase Realtime Database синхронізуються в реальному часі між усіма підключеними пристроями та клієнтами.

2.Гнучкість схеми:

База даних Firebase не вимагає строго визначеної схеми даних, що дозволяє зберігати та оновлювати дані динамічно.

3.Швидкодія:

Firebase Realtime Database пропонує швидкий доступ до даних, оскільки вона побудована на основі технології WebSockets та використовує відомі алгоритми синхронізації даних.

4.Масштабованість:

Firebase Realtime Database має вбудовану підтримку горизонтального масштабування, що дозволяє легко розширювати обсяги даних та навантаження.

5.Автентифікація та Авторизація:

Firebase надає зручні інструменти для автентифікації та авторизації користувачів, що спрощує роботу зі створенням безпечних додатків.

Firebase Realtime Database є потужним інструментом для розробки мобільних та веб-додатків, який пропонує зручний та ефективний спосіб зберігання та синхронізації даних в реальному часі. Його гнучкість, швидкодія та простота використання роблять його популярним вибором серед розробників по всьому світу. [14]

2.2 Переваги використання NoSQL Firebase

Переваги використання Firebase у порівнянні із іншими NoSQL базами даних

1. Простота використання:

Firebase надає інтуїтивно зрозуміле та просте API, яке дозволяє легко працювати з базою даних навіть для початківців. Порівняно з іншими NoSQL базами даних, які можуть мати складніші API та налаштування, Firebase виграє в простоті використання.

2.Реальний час:

Однією з ключових переваг Firebase є підтримка роботи в реальному часі. Дані в Firebase Realtime Database синхронізуються між клієнтами та сервером миттєво, що дозволяє створювати додатки зі спільним доступом до даних без затримок.

3.Швидкодія:

Firebase використовує оптимізований протокол обміну даними, що забезпечує високу швидкість передачі та обробки даних. Це особливо важливо для мобільних додатків, які потребують швидкої реакції на дії користувачів.

4.Масштабованість:

Firebase надає вбудовану підтримку горизонтального масштабування, що дозволяє легко розширювати обсяги даних та навантаження в майбутньому. Це дозволяє додаткам на основі Firebase зростати разом з їхніми потребами.

5.Інтеграція з іншими сервісами Firebase:

Firebase пропонує не лише базу даних, але й цілий набір інших сервісів, таких як аутентифікація, хостинг, зберігання файлів та інші. Інтеграція цих сервісів дозволяє розробникам створювати повноцінні додатки з одного джерела.

6.Безкоштовний тарифний план:

Firebase надає безкоштовний тарифний план, який включає в себе обмежені, але достатні ресурси для розвитку та тестування додатків. Це робить його доступним для широкого кола розробників та стартапів.

7.Підтримка різних платформ:

Firebase підтримує різні платформи, включаючи Android, iOS, веб та Unity. Це дозволяє розробникам створювати універсальні додатки, які працюють на різних пристроях та платформах без необхідності писати різні версії коду для кожної платформи. [15]

8.Надійність та безпека:

Firebase забезпечує високий рівень надійності та безпеки для даних. Google, який володіє Firebase, забезпечує постійну підтримку та моніторинг, що забезпечує безперебійну роботу сервісу. [16]

У порівнянні з іншими NoSQL базами даних, Firebase вирізняється своєю простотою використання, швидкодією, підтримкою реального часу та інтеграцією з іншими сервісами Firebase. Це робить його привабливим вибором для розробників, які шукають швидкий та ефективний спосіб розробки додатків зі спільним доступом до даних у реальному часі. [17]

2.3 Особливості використання та методи взаємодії з NoSQL Firebase

Firebase Realtime Database працює з даними у форматі JSON, що дозволяє легко організувати та зберігати дані в ієрархічній структурі. Дані зберігаються у вигляді дерева, де кожен вузол може містити інші вузли або значення.

Для читання даних з Firebase Realtime Database можна використовувати методи, які дозволяють отримати дані з бази даних у реальному часі або в одноразовому режимі. Наприклад, метод on() дозволяє слухати зміни у вказаному розділі бази даних, в той час як метод once() дозволяє отримати дані один раз без слухання подальших змін. Для запису даних до Firebase Realtime Database можна використовувати методи, такі як set(), update() та push(). Метод set() дозволяє встановити значення для певного шляху у дереві даних, update() дозволяє оновити частину даних, а push() додає новий елемент до списку даних, створюючи унікальний ключ. Для видалення даних з Firebase Realtime Database можна використовувати метод remove(), який видаляє вказаний вузол та всі його підвузли. Firebase дозволяє встановлювати обробники подій для слухання змін у базі даних. Це дозволяє реагувати на зміни даних в реальному часі та виконувати певні дії при зміні стану даних [18]

Firebase надає можливості для автентифікації та авторизації користувачів. Це дозволяє обмежувати доступ до даних залежно від ідентифікації користувача та його ролі в системі. Для оптимізації запитів до бази даних можна використовувати різні підходи, такі як індекси та фільтри. Firebase надає можливості для створення індексів та використання їх для швидкого пошуку даних. Firebase підтримує транзакції, які дозволяють забезпечити

консистентність даних у випадку паралельних змін. Транзакції дозволяють атомарно виконувати групу операцій з базою даних. [19]

Однією з головних особливосей Firebase-у є швидкість розробки, Firebase забезпечує швидку розробку додатків завдяки своєму інтуїтивно зрозумілому API та готовим до використання сервісам, що дозволяє розробникам швидко створювати та впроваджувати новий функціонал. Також Firebase пропонує розширений інструментарій для збору та аналізу даних про користувачів та використання додатків. Це дозволяє розробникам зрозуміти, як користувачі взаємодіють з їхнім додатком та як його можна покращити.

```
static async updateClient(data: ICreateNewClientProps, userID: string) {
  const docID = await this.getClientDoc(userID);
  const newClient = { id: userID, ...data };

  await updateDoc(doc(db, 'clients', docID), newClient);
  await InvoceService.updateInvocesClient(newClient as IClientDto);

}
```

Рисунок 2.1 – Приклад Запиту На Оновлення Користувача

2.4 Потенційні області застосування NoSQL Firebase

NoSQL Firebase - це багатофункціональний інструмент, який надає розробникам гнучку та потужну інфраструктуру для створення сучасних веб-додатків. Завдяки своїм унікальним можливостям та широкому спектру інтеграцій, NoSQL Firebase знаходить застосування у різних областях індустрії і технологічних сферах.

Потенційні області застосування NoSQL Firebase досліджують різноманітність веб-додатків, які можна створити з використанням цієї технології. Від соціальних мереж та мобільних додатків до інтернет-магазинів та інтерактивних ігор, NoSQL Firebase надає розробникам всі необхідні інструменти для успішного втілення їхніх ідей та проектів у реальність.

Розглянемо деякі з найбільш захоплюючих та потенційно вигідних областей застосування NoSQL Firebase

1. Соціальні мережі та спільноти

NoSQL Firebase надає механізми реального часу, які ідеально підходять для реалізації функцій соціальних мереж, таких як спільні чати, стрімінгові оновлення, сповіщення та інші. Він дозволяє створювати динамічні та інтерактивні спільноти зі швидкими та ефективними можливостями обміну даними.

2. Мобільні додатки

Завдяки своїм SDK та підтримці мобільних платформ, NoSQL Firebase є ідеальним вибором для розробки мобільних додатків. Він забезпечує легку інтеграцію з Android та iOS додатками, реалізуючи функціональність зберігання даних, аутентифікації користувачів, аналітики та багато іншого.

3. Інтерактивні веб-сайти та ігри

NoSQL Firebase дозволяє створювати інтерактивні веб-сайти та ігри, які реагують на дії користувачів миттєво. Це може бути використано для реалізації глобальних лідербордів, онлайн-ігор з реальними гравцями та інших форм інтерактивного взаємодії.

4. Інтернет-магазини та електронна комерція

NoSQL Firebase забезпечує швидку та масштабовану базу даних, що може бути використана для зберігання каталогів товарів, історії замовлень, корзин покупок та інших даних, пов'язаних з електронною комерцією. Він також надає інструменти для реалізації реактивних інтерфейсів та персоналізації пропозицій для користувачів.

5. Інтерактивні мультимедійні додатки

NoSQL Firebase дозволяє зберігати та обмінюватися мультимедійним контентом, таким як зображення, відео та аудіо, у реальному часі. Це може бути використано для створення інтерактивних додатків для спільного перегляду зображень, стрімінгового відео та інших мультимедійних взаємодій.

NoSQL Firebase відкриває широкі можливості для розробки різноманітних веб-додатків у різних областях. Від соціальних мереж і мобільних додатків до інтернет-магазинів та мультимедійних додатків, він надає потужні інструменти для створення сучасних та інноваційних рішень. [20]

2.5 Аналіз витрат на розробку та утримання проекту з використанням NoSQL Firebase порівняно з традиційними SQL базами даних

Розробка та утримання проекту бази даних є однією з найважливіших складових у створенні сучасних веб-додатків. Використання різних типів баз даних, таких як NoSQL Firebase та традиційні SQL бази даних, може мати значний вплив на витрати, пов'язані з розробкою та підтримкою проекту. У даному аналізі ми розглянемо різницю витрат між цими двома підходами.

1. Витрати на розробку

NoSQL Firebase: Розробка з використанням NoSQL Firebase може бути відносно швидкою та ефективною, оскільки вона не вимагає створення схеми бази даних та може бути легко інтегрована з фронтендом. Однак, інтеграція деяких додаткових функціональностей, таких як аутентифікація або аналітика, може потребувати додаткового часу та ресурсів.

SQL бази даних: Розробка з використанням традиційних SQL баз даних може бути більш часомісткою, оскільки вона вимагає створення детальної схеми бази даних та використання мови запитів SQL для взаємодії з даними. Однак, здійснення складних запитів та забезпечення консистентності може бути спрощене завдяки стандартним функціям SQL.

Якщо це класифікувати, отримаємо наступні пункти

Час розробки: Розробка з використанням NoSQL Firebase може бути швидшою, оскільки вона не потребує створення складних схем баз даних або написання запитів SQL. Розробники можуть швидко реалізувати функціональність, використовуючи зручні API та інструменти, що надає Firebase.

Оплата послуг: Витрати на розробку з NoSQL Firebase включають в себе оплату планування хмарних послуг Firebase. Ці витрати можуть зростати з ростом обсягу даних та використання інших функцій Firebase, таких як аутентифікація, аналітика та інші.

Навчання та підготовка персоналу: Хоча NoSQL Firebase може бути легким у використанні, розробники можуть витратити час на навчання та ознайомлення з його функціональністю. Це може включати участь у тренінгах, читання документації та експериментування з різними можливостями Firebase.

Порівняно з NoSQL Firebase, розробка з використанням традиційних SQL баз даних може мати свої особливості та витрати:

Схема бази даних: Розробка з використанням SQL баз даних вимагає створення детальної схеми бази даних перед початком роботи. Це може зайняти час, особливо для проектів з складною структурою даних.

Запити SQL та оптимізація: Написання складних запитів SQL та їх оптимізація може вимагати досвіду та знань в області баз даних. Це може призвести до додаткових витрат на час розробки та тестування.

Установка та конфігурація: Установка та налаштування SQL бази даних може бути складним завданням, особливо для великих або розподілених систем. Це може вимагати залучення спеціалізованого персоналу або використання зовнішніх послуг.

Витрати на розробку проекту з використанням NoSQL Firebase порівняно з традиційними SQL базами даних можуть варіюватися в залежності від різних факторів, таких як обсяг та складність проекту, рівень експертизи команди розробників та використання додаткових функцій та сервісів. Прийняття рішення щодо вибору бази даних важливо ретельно зважити всі плюси та мінуси кожного підходу та врахувати специфіку проекту.

А тепер розглянемо витрати на утримання таких проектів відповідно для NoSQL Firebase та звичайні SQL бази даних

NoSQL Firebase: Утримання NoSQL Firebase може бути більш простим та ефективним, оскільки підтримка та моніторинг інфраструктури здійснюється з

боку провайдера хмарних послуг. Однак, зростання обсягу даних та потреба в додаткових функціях може призвести до збільшення витрат.

SQL бази даних: Утримання традиційних SQL баз даних може бути більш складним, оскільки воно вимагає досвідченого адміністратора баз даних для забезпечення оптимальної продуктивності, резервного копіювання та забезпечення безпеки даних. Однак, це також надає більшу гнучкість у керуванні та оптимізації бази даних.

Якщо це також класифікувати, то отримаємо наступні пункти:

Утримання проекту з використанням NoSQL Firebase може включати такі витрати:

Плата за хмарні послуги: NoSQL Firebase пропонує платні плани хмарних послуг, які можуть включати обсяг даних, кількість запитів та інші метрики. Зростання обсягу даних та використання додаткових функцій може призвести до збільшення витрат на хмарні послуги.

Резервне копіювання та відновлення: Для забезпечення надійності даних може знадобитися здійснювати регулярне резервне копіювання бази даних та можливість її відновлення в разі втрати даних або аварійної ситуації. Це може займати додатковий час та ресурси.

Оновлення та підтримка: NoSQL Firebase регулярно оновлюється з метою виправлення помилок, вдосконалення функціоналу та забезпечення безпеки. Поновлення до нових версій може вимагати тестування та впровадження змін, що також може призвести до додаткових витрат на утримання.

Утримання проекту з використанням традиційних SQL баз даних може включати наступні витрати:

Хостинг та інфраструктура: Для традиційних SQL баз даних може знадобитися власний сервер або хостинговий план для забезпечення інфраструктури. Це може включати витрати на оренду серверного обладнання, оплату хостингових послуг та інших витрат на інфраструктуру.

Ліцензії та підтримка: Деякі комерційні SQL бази даних можуть вимагати придбання ліцензій на користування та підтримку програмного забезпечення. Це може становити значну частину витрат на утримання проекту.

Адміністрування та підтримка: Утримання SQL баз даних може вимагати наявності кваліфікованого адміністратора баз даних для забезпечення оптимальної продуктивності, резервного копіювання та забезпечення безпеки даних.

Витрати на утримання проекту залежать від різних факторів, включаючи тип бази даних, розмір проекту, рівень експертизи команди та потреби у додаткових функціях та послугах. Прийняття рішення про вибір бази даних важливо ретельно розглянути всі аспекти витрат та ефективності, щоб забезпечити оптимальне управління проектом.

РОЗДІЛ 3 РОЗРОБКА CRM СИСТЕМИ

3.1 Приклад та опис структури бази даних CRM системи

Враховуючи увесь функціонал CRM системи було побудовано наступну схему взаємодій в NoSQL Firebase Базі Даних (рисунок 3.1)

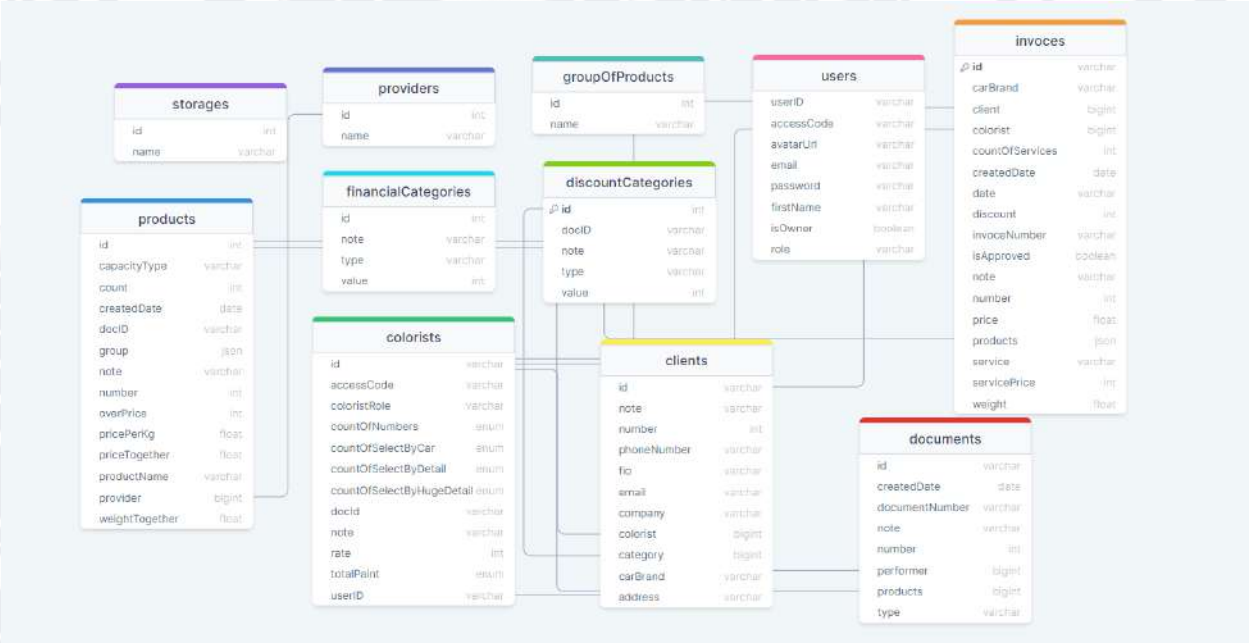


Рисунок 3.1 – Схема зв’язків CRM системи колористів

Опишемо схему. Колонка Users містить в собі унікальний ключ userID, поле accessCode яке відповідає за такий собі індикатор колориста, який у кожного колориста унікальний, за допомогою нього колорист підписує документи, працює з накладними, взаємодіє зі складом та товаром на складі та ще декілька важливих рішень. Цей ключ допомагає відрізняти колористів та захищає будь які дії в застосунку від несанкціонований входів. Поле avatarURL відповідає за зображення користувача в застосунку, яке ми маємо можливість змінювати за потреби. Email, password, унікальні поля створені задля можливості авторизації користувача в застосунку. Поле firstName – це відповідно ім’я колориста, role – позначає роль користувача, усього в застосунку маємо декілька ролей Admin,User, що позначають Власника та Колориста. Різниця між цими

типами доступність до функціоналу застосунку, наприклад Колорист має доступ лише до сторінок зі списком Накладних, Користувачів, та зі списком товарів на складі. Взаємодіяти з певним функціоналом на цих сторінках цей користувач також не може, в той час як користувач із роллю Admin має доступ абсолютно до усього функціоналу застосунку. Поле isOwner позначає чи є власником саме цей користувач, це означає що в нього як й у Admin є доступ до усього функціоналу.

Колонка Storages містить в собі лише 2 поля це унікальний id та name, що позначає назву складу. По замовченню ми матимемо лише декілька складів й можливість переміщувати товари між ними. Цей функціонал ми переглянемо пізніше.

Колонка Providers містить в собі також усього 2 поля з id та name що позначають унікальну ключ та ім’я постачальника певних продуктів.

Колонка GroupOfProducts містить id, name та означають унікальний ключ, ім’я групи продукта.

Колонка Financial Categories містить в собі id,docID за допомогою якого в мене є можливість керувати саме документом в Firebase, note що позначає замітки, які залишає колорист, type та value що позначають тип знижки та її значення

Тепер розглянемо більші по місткості колонки, які в собі містять декілька полів, які є частинами інших колонок. Наприклад колонка products, яка відповідно позначає продукт (товар) застосунку. Він містить в собі унікальний ключ, тип місткості продукту (в літрах, кілограмах, штуках), дату створення,групу продукту що є об’єктом від таблиці Group of Products.

Примітка до товару, кількість, прайс за штуку або за кілограм, назву продукута, постачальника що є об’єктом від таблиці Providers, загальна вага (якщо тип товару належить до тих, що треба зважити), націнка на товар та загальна його ціна

Колонка Colorists містить в собі інформацію щодо колористів. Варто не плутати колонку Users із цією, адже користувач містить в собі певну інформацію

накшталт ім’я, пошта, пароль та код, колорист в собі містить роль Колориста, яка може бути просто Колорист та Супер-Колорист, від вибору ролі буде залежити який функціонал доступний для колориста. Його кількість підборів по деталям, по габаритним деталям, по машинам та по номерам, по фарбам. Нотатки та id користувача який дозволяє нам отримати інформацію з таблиці Users.

Останньою таблицею є накладні, найбільша таблиця, оскільки дуже тісно пов’язані з таблицями Products,Clients,Colorist.Містить в собі такі поля як, марка машини, дату створення, знижку, номер накладної, чи затверджена вона, ціну, послугу, ціна за послугу та вага.

3.2 Огляд ключових аспектів при побудові схем для NoSQL бази даних

При побудові схем для NoSQL баз даних важливо враховувати кілька ключових аспектів, що допоможуть забезпечити ефективність, масштабованість та надійність вашої бази даних. Ось детальний перелік порад:

Аналіз вимог застосунку: Слід ретельно проаналізувати вимоги додатку та потреби в зберіганні та доступі до даних. Визначити типи даних, їх структуру, частоту доступу та потребу у масштабованості.

Вибір відповідного типу NoSQL бази даних: можна розглянути тип NoSQL бази даних, що найбільше підходить для ваших потреб (ключ-значення, документ-орієнтований, стовпчастий, графовий). Також треба врахувати особливості кожного типу бази даних та їх можливості у відповідності до ваших вимог.

Розробка оптимальної структури даних: Варто використовувати денормалізацію для зменшення кількості запитів та підвищення швидкодії. Розділити дані на логічні групи, щоб зменшити конфлікти при одночасному доступі та підвищити масштабованість. Використовувати вбудовані або зовнішні зв'язки між даними відповідно до потреб вашого додатку.

Управління доступом до даних:

Встановлення належних прав доступу до даних для забезпечення безпеки та конфіденційності. Слід також розглянути можливість реалізації механізму автентифікації та авторизації користувачів.

Реалізація механізму резервного копіювання та відновлення даних: Варто передбачити механізми регулярного резервного копіювання бази даних для забезпечення безпеки даних та можливості відновлення у випадку аварій.

Тестувати механізми відновлення для перевірки їх ефективності та швидкодії.

Моніторинг та оптимізація продуктивності: Необхідно встановити моніторинг продуктивності бази даних для виявлення проблем та узагальнення місць оптимізації. Оптимізувати запити та індексацію даних для підвищення швидкодії та продуктивності.

Управління консистентністю даних: Слід розробити стратегію управління консистентністю даних, особливо у розподілених середовищах.

Використовувати механізми транзакцій або інші засоби забезпечення консистентності даних відповідно до ваших потреб.

Скалабельність та висока доступність: Розгляньте можливості масштабування бази даних та високої доступності для забезпечення неперервності роботи вашого додатку. Використовуйте реплікацію та шарування для підвищення масштабованості та надійності.

Тестування та валідація: Варто також проводити ретельне тестування розробленої схеми даних для перевірки її правильності та ефективності.

Виконувати валідацію даних для перевірки їх відповідності встановленим правилам та обмеженням.

Побудова схеми для NoSQL баз даних вимагає уважного аналізу вимог додатку, вибору відповідного типу бази даних, розробки оптимальної структури даних, управління доступом до даних, реалізації механізмів резервного копіювання та відновлення даних, моніторингу та оптимізації продуктивності,

управління консистентністю даних, забезпечення скалабельності та високої доступності, а також тестування та валідації.

При побудові схеми даних для NoSQL бази даних важливо враховувати особливості конкретного типу бази даних (ключ-значення, документ-орієнтований, стовпчастий, графовий), потреби додатку у швидкодії, масштабованості та надійності, а також специфіку обробки даних та зв'язків між ними.

Забезпечення ефективності, надійності та масштабованості бази даних вимагає ретельного проектування, тестування та оптимізації схеми даних, а також постійного моніторингу та удосконалення на етапі роботи з базою даних. Відповідне врахування всіх цих аспектів дозволить забезпечити оптимальну роботу NoSQL бази даних у вашому додатку.

3.3 Опис структури запитів до БД за допомогою Firebase API

3.3.1 Приклад ініціалізації Firebase змінних та його опис

Відповідно до задачі CRM системи, я використав Firebase API для запитів до NoSQL бази даних застосунку. Перш за все необхідно ініціалізувати проект та дати зрозуміти Firebase до якого проекту ми звертаємося, з якими колонками будемо працювати та які методи Firebase застосовуватимемо. Відповідний приклад ініціалізації Firebase застосунку видно на рисунку 3.2

Проаналізуємо наступний код.

Імпорт необхідних функцій з Firebase SDK:

В даному пункті імпортуються необхідні функції з Firebase SDK. Конкретно, initializeApp для ініціалізації додатку Firebase, getAuth для отримання сервісу аутентифікації, getFirestore для роботи з Firestore базою даних та getStorage для роботи з Firebase Cloud Storage.

```
import { initializeApp } from 'firebase/app';
import { getAuth } from 'firebase/auth';
import { getFirestore } from 'firebase/firestore';
import { getStorage } from 'firebase/storage';

const { VITE_API_KEY, VITE_AUTH_DOMAIN, VITE_PROJECT_ID, VITE_STORAGE_BUCKET, VITE_MESSAGING_ID, VITE_APP_ID } = import.meta.env;

const firebaseConfig = {
  apiKey: VITE_API_KEY,
  authDomain: VITE_AUTH_DOMAIN,
  projectId: VITE_PROJECT_ID,
  storageBucket: VITE_STORAGE_BUCKET,
  messagingSenderId: VITE_MESSAGING_ID,
  appId: VITE_APP_ID,
};

export const app = initializeApp(firebaseConfig);
export const auth = getAuth(app);
export const db = getFirestore(app);
```

Рисунок 3.2 – Ініціалізація Firebase застосунку

Отримання змінних середовища:

За допомогою import.meta.env отримуються змінні середовища, які містять конфіденційну інформацію про конфігурацію Firebase. Ці змінні містяться у відокремленому файлі .env, який не включений у вихідний код та міститься лише на сервері або в локальному середовищі розробки.

Використання змінних середовища є рекомендованим підходом при розробці програмного забезпечення, особливо тоді, коли ви працюєте з конфіденційними даними або іншою конфігураційною інформацією, яка може змінюватися в залежності від середовища виконання (наприклад, розробка, тестування, продуктивне середовище).

Змінні середовища дозволяють зберігати конфіденційні дані, такі як API ключі або секрети, в окремому файлі, який не включений до версій керування, такого як Git. Це забезпечує більшу безпеку, оскільки ця конфіденційна інформація не потрапляє у відкритий доступ та не зберігається у відкритому коді. Також це дозволяє зручно керувати конфігурацією додатку для різних середовищ, таких як розробка, тестування та продуктивне середовище. Ми можемо легко змінювати значення змінних середовища відповідно до потреб кожного середовища без необхідності змінювати сам код додатку.

Використання змінних середовища дозволяє нам використовувати один і той же код додатку у різних середовищах без необхідності змінювати його. Це спрощує процес розробки та тестування, оскільки нам не потрібно вручну змінювати конфігураційні дані під час переходу від одного середовища до іншого. А також це дозволяє уникнути помилок, які можуть виникнути через введення конфіденційних даних безпосередньо у вихідний код. Це зменшує ризик неправильного використання або витоку конфіденційної інформації.

Створення об'єкта конфігурації Firebase:

У цьому кроці створюється об'єкт firebaseConfig, який містить всю необхідну конфігураційну інформацію для налаштування підключення до Firebase. Ця інформація міститься у змінних середовища, отриманих на попередньому кроці. Розглянемо кожне поле більш детальніше

apiKey: Цей ключ використовується для ідентифікації вашого додатку в Firebase. Він є унікальним для кожного додатку і використовується для авторизації доступу до Firebase сервісів.

authDomain: Це доменне ім'я або URL-адреса, яка використовується для аутентифікації користувачів у вашому додатку. Firebase використовує це для перенаправлення користувачів на правильну сторінку для аутентифікації.

projectId: Це ідентифікатор вашого проекту в Firebase. Використовується для ідентифікації конкретного проекту серед інших в адміністративному інтерфейсі Firebase.

storageBucket: Це ім'я або URL-адреса вашого Firebase Cloud Storage. Використовується для доступу до збережених файлів у вашому додатку, таких як зображення, відео тощо.

messagingSenderId: Цей ключ використовується для ідентифікації вашого додатку у Firebase Cloud Messaging. Використовується для надсилання пуш-сповіщень та повідомлень користувачам.

appId: Це ідентифікатор додатку, який використовується в Firebase. Він ідентифікує конкретний додаток серед інших додатків у Firebase проекті.

Кожен з цих ключів має свою унікальну роль у налаштуванні та забезпеченні правильної роботи вашого додатку з Firebase. Їх комбінація дозволяє Firebase ідентифікувати та авторизувати ваш додаток, забезпечує доступ до необхідних сервісів Firebase та надає можливість зберігання та обміну даними у безпечному та ефективному середовищі.

Ініціалізація з'єднання з Firebase:

За допомогою функції initializeApp ініціалізується з'єднання з Firebase, використовуючи отриману конфігурацію firebaseConfig. Після цього з'єднання стає доступним для використання усіма іншими службами Firebase.

Отримання посилань на різні служби Firebase:

В цьому кроці використовуються функції getAuth, getFirestore та getStorage для отримання посилань на різні служби Firebase. Кожна з цих функцій приймає як аргумент з'єднання з Firebase, отримане під час ініціалізації додатку, та повертає об'єкт, який представляє відповідну службу Firebase.

Отже, кожен крок у цьому коді важливий для належної ініціалізації та налаштування з'єднання з Firebase, щоб забезпечити правильну роботу додатку з Firebase сервісами.

Згодом ми будуємо за допомогою класового підходу сервіси задля того, щоб звертатися до кожного сервісу окремо та мати змогу розділити логіку відповідного до кожного сервісу. Наприклад, щоб сервіс Invoce мав змогу керувати виключно колонкою invoce та її методами. Приклад сервісу Invoce бачимо на рисунку 3.3

3.3.2 Застосування абстрактних класів та сервісів при написанні запитів до Firebase API

Як можна помітити, я застосовую абстрактні класи для побудови моїх сервісів. Це обумовлено тим, що це дозволяє мені відповідно розділити мою логіку на незалежні частини.

Абстрактні класи в TypeScript - це класи, які можуть містити абстрактні (неперевизначені) методи, які повинні бути реалізовані в похідних класах.

Абстракція - це ключовий принцип об'єктно-орієнтованого програмування (ООП), який дозволяє приховати деталі реалізації та представити лише ті характеристики об'єкта, які є необхідними для зовнішнього світу. Абстрактні класи є інструментом для вираження абстракції, оскільки вони дозволяють визначати загальні концепції.

```
export class InvoceService {
  static async createNewInvoce({
    actDate,
    actNumber,
    colorist,
    data,
    client,
    carBrand,
    totalPrice,
    discount,
    service,
    note,
    countOfServices,
    weight,
    servicePrice,
  }: ICreateNewInvocieProps): Promise<IInvoceDto> {
    const number = (await this.getNewInvoceRowNumber()) + 1

    const id = uuidv4();
    const newDoc = {
      id,
      number,
      invoceNumber: actNumber,
      date: actDate.format('DD/MM/YYYY'),
      client,
      colorist,
      carBrand: carBrand,
      price: totalPrice,
      discount,
      service,
      note,
      countOfServices,
      createdDate: dayjs().format('YYYY-MM-DD HH:mm:ss'),
      products: data,
      isAprroved: false,
```

Рисунок 3.3 - Приклад сервісу Invoce

Ці класи дозволяють структурувати код шляхом визначення загальної функціональності, яку можливо застосовувати незалежно один від одної в різних частинах коду. Вони дозволяють також узагальнити спільні властивості та методи відповідно до свого призначення що дозволяє уникнути повтору коду та додає унікальності йому. Забезпечують також класи більшу структурованість коду та підтримку коду.

Перевагами абстрактних класів є те, що вони створюють загальну архітектуру певної частини програми та забезпечують її стабільність роботи та розширюваність.

Абстрактні класи є потужним інструментом для створення структурованого, гнучкого та легко розширюваного коду в програмуванні. Вони допомагають уникнути дублювання коду, сприяють поліморфізму та забезпечують зручний інтерфейс для взаємодії з об'єктами різних класів.

Задля кращого розуміння, що саме в нас відбувається в коду при написанні сервісів, які звертаються до Firebase API розглянемо декілька методів накладних. Прикладом одного з таких методів буде метод Create New Invoce на рисунку 3.4

```
export class InvoceService {
  static async createNewInvoce({
    actDate,
    actNumber,
    colorist,
    data,
    client,
    carBrand,
    totalPrice,
    discount,
    service,
    note,
    countOfServices,
    weight,
    servicePrice,
  }: ICreateNewInvoceProps): Promise<IInvoceDto> {
    const number = (await this.getNewInvoceRowNumber()) + 1;

    const id = uuidv4();
    const newDoc = {
      id,
      number,
      invoceNumber: actNumber,
      date: actDate.format('DD/MM/YYYY'),
      client,
      colorist,
      carBrand: carBrand,
      price: totalPrice,
      discount,
      service,
      note,
      countOfServices,
      createdDate: dayjs().format('YYYY-MM-DD HH:mm:ss'),
      products: data,
      isAprroved: false,
      weight,
      servicePrice,
    };

    await DocumentsService.createNewDocument(actNumber, actDate, DOCUMENT_TYPE.INVOCE, colorist, []);
    const doc = await addDoc(collection(db, 'invoces'), newDoc);

    return { ...newDoc, docID: doc.id };
  }
```

Рисунок 3.4 - Повний лістинг коду методу Create New Invoce

3.3.3 Опис декількох запитів до створення, оновлення та видалення записів в Firebase NoSQL Базі даних

На вхід функція приймає параметри, які містять дані для створення відповідно нової накладної, це такі дані як дата акту, номер, колорист який створив накладну, ім’я клієнта його знижка та марка машини, загальна вартість, послугу та ціну за послугу. Потім функція викликає метод getNewInvoceRowNumber для отримання наступного номера рядка рахунку. Отриманий номер збільшується на одиницю та використовується для створення нового рахунку. Наступним кроком використовується uuidv4 для генерації унікального ідентифікатора. Після цього створюється новий об'єкт документа newDoc, який містить всі дані про нову накладну, включаючи дату, номер, клієнта, колориста, загальну вартість та інші деталі.

Згодом ми звертаємося до окремого сервісу DocumentService який створює документ щодо накладної та буде містити її основну інформацію.

Наступний найважливішим кроком йде звертання до Firebase API на створення нової накладної в базі даних. Цей пункт розглянемо більш детальніше.

addDoc є методом Firebase Firestore, який використовується для додавання нового документа до колекції у базі даних. Цей метод приймає два параметри: посилання на колекцію, до якої потрібно додати документ, і сам документ, який потрібно додати.

collection є функцією Firebase Firestore, яка використовується для створення посилання на конкретну колекцію в базі даних. Вона приймає два параметри: посилання на базу даних (db у цьому випадку) та назву колекції, до якої потрібно звернутися.

db є посиланням на об'єкт бази даних Firestore, який ініціалізується та імпортується на початку коду. Він використовується для звернення до бази даних та виконання операцій з даними.

Invoces - це назва колекції в базі даних Firestore, до якої буде доданий новий документ. В даному випадку, новий документ буде доданий до колекції 'invoces'.

Тепер розглянемо інший приклад на оновлення даних продукту в БД представлений на рисунку 3.5

```
static async updateProduct(prevProduct: IProductDto, newProduct: IAcceptProductTableRowProps): Promise<IProductDto> {
  let weightTogether = 0;

  if (typeof newProduct.weightTogether === 'number' && typeof prevProduct.weightTogether === 'number') {
    weightTogether = newProduct.weightTogether + prevProduct.weightTogether;
  }
  const count = prevProduct.count + newProduct.count;
  const buyPrice = prevProduct.buyPrice > newProduct.buyPrice ? prevProduct.buyPrice : newProduct.buyPrice;
  const overPrice = prevProduct.overPrice > newProduct.overPrice ? prevProduct.overPrice : newProduct.overPrice;
  const capacity = (prevProduct.capacity as number) > 0 ? (prevProduct.capacity as number) : 1;

  const newPriceTogether = capacity * count * buyPrice * overPrice;

  await updateDoc(doc(db, 'products', prevProduct.docID), {
    count,
    buyPrice,
    weightTogether,
    overPrice,
    priceTogether: newPriceTogether,
  });

  return {
    ...newProduct,
    count,
    buyPrice,
    weightTogether,
    overPrice,
    priceTogether: newPriceTogether,
    docID: prevProduct.docID,
    group: prevProduct.group,
    provider: prevProduct.provider,
    note: newProduct.note ? newProduct.note : prevProduct.note,
  };
}
```

Рисунок 3.5 - Повний лістинг коду методу Update Product

Отже, якщо розглянути цей код більш детальніше можемо сказати, що функція updateProduct приймає два параметри: prevProduct, який є об'єктом попереднього продукту, та newProduct, який є об'єктом нового продукту. Обидва параметри мають тип IProductDto, що, очевидно, є інтерфейсом для опису продукту. В функції спочатку виконуються різні обчислення на основі даних з обох продуктів. Наприклад, обчислюється загальна вага (weightTogether), кількість (count), ціна покупки (buyPrice), ціна продажу (overPrice) та загальна ціна (newPriceTogether).

Після того як всі обчислення завершено, за допомогою методу updateDoc оновлюється документ в колекції 'products' у базі даних Firestore. Цей метод

приймає три параметри: посилання на документ, який потрібно оновити, та об'єкт з новими значеннями для полів документа.

Перший параметр методу updateDoc - це посилання на документ, який потрібно оновити. У випадку використовується функція doc, яка приймає три параметри: Посилання на базу даних (db),

Назву колекції, до якої належить документ ('products'),

Ідентифікатор (ID) документа, який потрібно оновити (поле prevProduct.docID).

Другий параметр методу updateDoc - це об'єкт, який містить нові значення для оновлення. Цей об'єкт повинен містити ключі, що відповідають полям документа, які потрібно оновити, та їх нові значення. У даному випадку це об'єкт з полями count, buyPrice, weightTogether, overPrice та priceTogether.

Після виклику методу updateDoc Firebase Firestore автоматично оновлює документ у базі даних з новими значеннями.

Ключове слово await використовується перед викликом updateDoc, оскільки цей метод є асинхронним. Воно забезпечує те, що виконання коду буде чекати на завершення оновлення документа у базі даних перед продовженням виконання наступних операцій.

Функція повертає об'єкт IProductDto, який містить оновлені дані про продукт. Цей об'єкт містить нові значення, обчислені в процесі виконання функції, а також ідентифікатор документа docID з попереднього продукту.

Тепер розглянемо ще один метод Firebase API це видалення певного запису в NoSQL базі даних. Цей метод представлений на рисунку 3.6

```
81
82    static async deleteColorist(docId: string, userID: string) {
83      await deleteDoc(doc(db, 'colorists', docId));
84      await UserService.deleteUser(userID);
85      return docId;
86    }
87
```

Рисунок 3.6 - Повний лістинг коду методу Delete Colorist

Проаналізуємо метод, що видаляє запис колориста. Функція deleteColorist отримує два параметри: docId - ідентифікатор документа колориста, який потрібно видалити, та userID - ідентифікатор користувача, який пов'язаний з цим колористом.

Для видалення документа з колекції 'colorists' у базі даних Firestore використовується метод deleteDoc. Він приймає посилання на документ, який потрібно видалити, яке створюється за допомогою функції doc, та виконує видалення цього документа. Після виклику deleteDoc відбувається видалення документа з бази даних.

Після видалення документа колориста з бази даних, викликається метод deleteUser з класу UserService для видалення користувача, пов'язаного з цим колористом. Це призначено для підтримки консистентності даних: якщо колорист видаляється, його пов'язаний користувач також має бути видалений.

3.3.4 Використання асинхроного коду в JavaScript\TypeScript

Під час написання усіх запитів до бази даних Firebase я використовував асинхроний код для повної маніпуляції даними.

JavaScript є одним з найпопулярніших мов програмування для розробки веб-додатків, і він використовується для створення веб-сторінок з інтерактивними елементами, які реагують на дії користувачів. Однак, JavaScript також підтримує асинхронне програмування, що дозволяє виконувати операції, які займають час, такі як мережеві запити або операції з файлами, без блокування виконання інших операцій. [21]

Для написання асинхроного коду ми маємо декілька підхідів це Callback, Promise та Async\Await.

Callback - це функція, яка передається як аргумент до іншої функції і викликається пізніше, після завершення виконання асинхронної операції.

Зворотні виклики (callbacks) є одним із фундаментальних концептів асинхронного програмування в JavaScript. Дозволяють вони виконувати

асинхронні операції та визначати дії, які потрібно виконати після завершення цих операцій.

Основними принципами калбеків є передача функції як аргументу: У JavaScript функції можуть бути передані як аргументи в інші функції. Така функція, що передається, називається зворотним викликом.

Виклик після завершення асинхронної операції: Зворотний виклик зазвичай викликається після завершення асинхронної операції, наприклад, отримання даних з сервера або завантаження файлу. [22]

Обробка результату або помилки: Зазвичай зворотній виклик приймає параметри, які представляють результат асинхронної операції або помилку, якщо така виникла.

Перевагами калбеків є простота використання: Зворотні виклики дозволяють створювати асинхронний код без потреби в розумінні промісів або async/await. Гнучкість: Калбеки можуть бути використані для будь-яких асинхронних операцій, не обмежуючися лише використанням вбудованих методів.

Проте недоліками є Callback Hell: Глибоке вкладення калбеків може призвести до погано читабельного та важкого для обслуговування коду. Підвищена складність помилок: Важко управляти помилками в калбеках, особливо коли їх багато в складному коді. Відсутність контролю над потоком: Калбеки можуть викликатися в непередбачуваний момент, що може призвести до складнощів у керуванні потоком виконання. Приклад використання підходу калбеків представлений на рисунку 4.7 [23-24]

Проміси (Promises) в JavaScript - це об'єкти, які представляють результат асинхронної операції, яка може бути виконана або відхилена в майбутньому. Вони дозволяють створювати асинхронний код, який є більш зрозумілим та простим для обслуговування, особливо у випадках, коли виникає багато послідовних або паралельних асинхронних операцій.

```
205
206  function fetchData(callback) {
207    setTimeout(() => {
208      const data = 'Data fetched';
209      callback(null, data); // Передача результату та null в якості помилки
210    }, 1000);
211  }
212
213  fetchData((error, data) => {
214    if (error) {
215      console.error('Error:', error);
216    } else {
217      console.log('Data:', data);
218    }
219  });
```

Рисунок 4.7 - Приклад Використання Калбеків

Основними принципами промісів є його створення, за допомогою конструктора Promise, який приймає функцію з двома аргументами resolve та reject.Обробка результатів, після створення промісу ми можемо приєднати обробники для успішного виконання та випадку помилки then та catch методи відповідно.

Перевагою промісів є те, що проміси дозволяють писати асинхронний код, який є більш зрозумілим та легким для обслуговування. Ланцюження промісів дозволяє уникнути глибокого вкладення зворотних викликів, що відомо як callback hell. За допомогою методу .catch() можна легко обробити помилки, що сталася під час виконання асинхронної операції [25-26]

Недоліками промісів є те, що для деяких початківців проміси можуть бути складними для розуміння через потребу використання ланцюжків .then() та .catch().Проміси зазвичай викликаються автоматично, що може призвести до несподіваної поведінки у деяких випадках. Проміси підтримуються не всіма старими браузерами, тому може знадобитися поліфіл для підтримки у старих версіях. Приклад використання промісів представлений на рисунку 3.8.

async та await - це синтаксичний цукор над промісами, який робить асинхронний код більш зрозумілим та легким для розуміння. Вони були додані

в ECMAScript 2017 (ES8) і стали популярними засобами для написання асинхронного коду в JavaScript.

Рисунок 3.8 - Приклад використання промісів

Функція, яка має ключове слово async перед її оголошенням, завжди повертає проміс. Це дозволяє використовувати await всередині цієї функції.

Ключове слово await може використовуватися всередині async функцій для очікування результату проміса. Виконання коду зупиняється, доки проміс не буде вирішено або відхилений.

Перевагами async\await є те, що async/await дозволяє писати асинхронний код, який виглядає подібно синхронному, що робить його більш зрозумілим та легким для розуміння. Ланцюження async/await дозволяє уникнути глибокого вкладення зворотних викликів, що відомо як callback hell. За допомогою блоку try/catch можна легко обробляти помилки в асинхронному коді, що робить його більш безпечним та стабільним. async/await дозволяє легко ланцювати проміси разом за допомогою ключового слова await, що робить код більш структурованим та зрозумілим.

Проте недоліками async\await є те, що використання await блокує виконання коду, тому необхідно уникати довгих операцій, які можуть

заблокувати виконання програми. Деякі старі версії браузерів можуть не підтримувати async/await, тому для використання цих функцій вам може знадобитися поліфіл.

Кожен з трьох підходів має свої переваги та недоліки, і вибір між ними залежить від конкретного випадку використання, уподобань та вимог проекту. У більшості випадків async/await є більш зручним та елегантним підходом, який робить асинхронний код більш зрозумілим та простим для обслуговування. Однак, слід врахувати підтримку браузерів та особливості вашого проекту при виборі між цими підходами.

Під час створення запитів до бази даних я використовував в основному підхід async\await оскільки для мене, він є доволі зручним, зрозумілим та легким під час написання коду та подальшого масштабування його.

3.4 Опис безпеки даних та взаємодії колориста з застосунком

У сучасному цифровому світі, де дані стають найціннішим ресурсом, питання безпеки даних виявляється критично важливим для різноманітних сфер діяльності. Збільшення обсягів і складності обробки інформації, що відбувається в організаціях будь-якої форми і розміру, підкреслює необхідність ретельного та системного підходу до захисту цих даних.

У цьому контексті важливо розглядати не лише захист цих даних від несанкціонованого доступу, але й їхню ефективну управління та використання. Взаємодія колористів з системами управління відносинами з клієнтами (CRM) виявляється ключовим фактором в забезпеченні якісної обробки та захисту даних.

Як було вже обумовлено вище, задля взаємодії з певними частинами застосунку колористу необхідно мати власний код доступу, який він згодом буде застосовувати під час взаємодії з CRM системою

Як приклад, задля створення накладної, колористу слід використати його код доступу для підтвердження особи. Приклад сторінки де колористу слід вести код для створення накладної наведено на рисунку 3.9

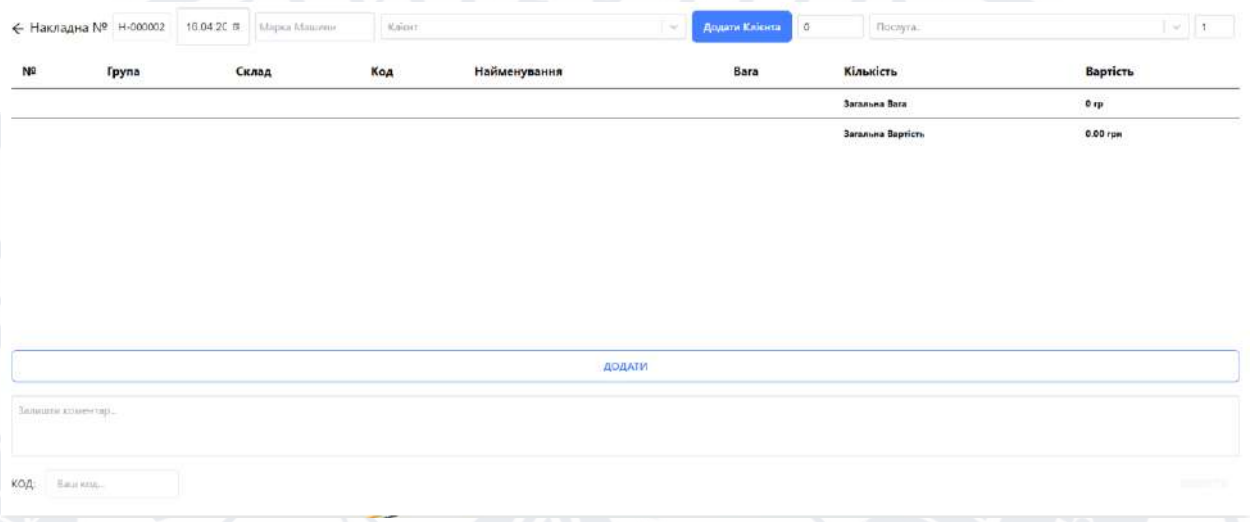


Рисунок 3.9 - Приклад сторінки створення накладної

Під час створення накладної, колорист має обрати відповідні продукти, які використовуються в замовленні. Також повинен вказати клієнта, знижку для клієнта, від якої залежить остаточна ціна. Згодом обирає послугу, в даному випадку це може бути підбір за номером, підбір по авто, підбір по лючку та габаритним деталям. Кожен підбір має унікальний ключ id та свою коштовність, усе це описано в таблиці financialCategories. Дану таблицю та решту таблиць БД продемонстровано на рисунку 3.10.

Також вказується марка машини за якою робиться підбір. Згодом вводиться унікальний код, який перевіряє чи присутній колорист в системі з таким кодом та записує накладну відповідно до колориста, по цьому коду. Приклад таблиці де присутні дані колористів наведено на рисунку 3.11

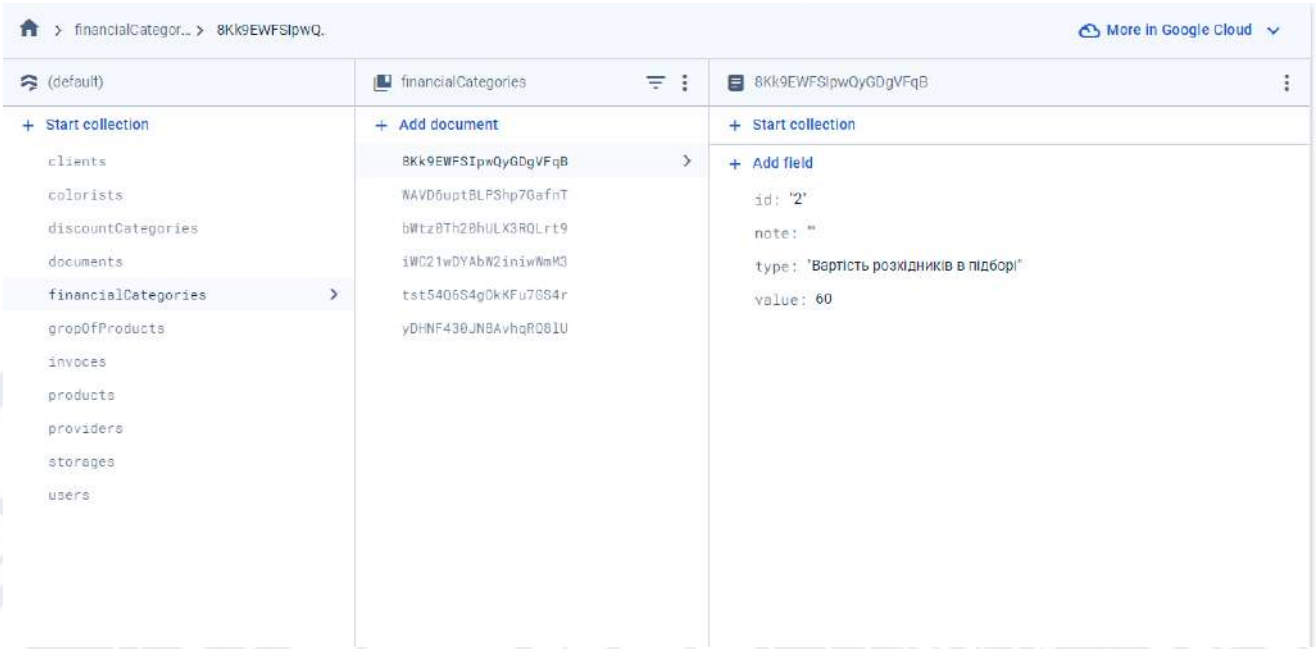


Рисунок 3.10 - Приклад таблиці financialCategories в Firebase

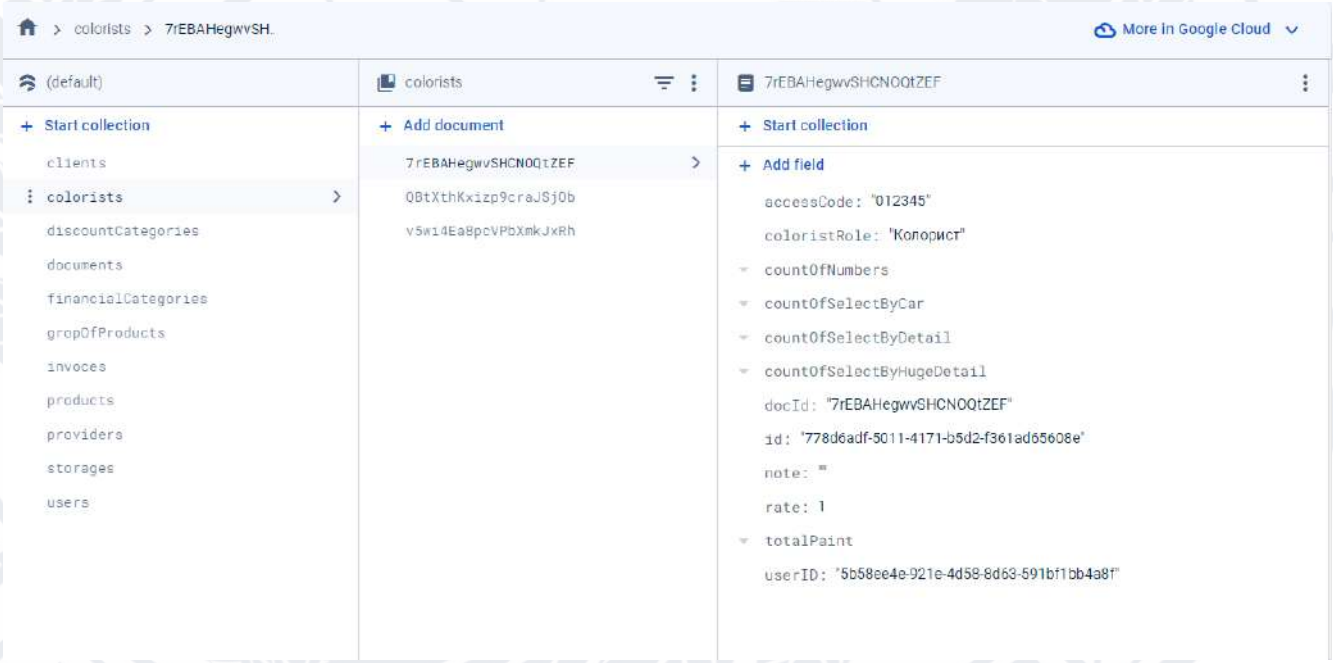


Рисунок 3.11 - Приклад таблиці colorists в Firebase

```
549  const getColoristByAccessCode = async (code: string, allColorists: IColorsitDto[]) => {
550    const colorist = allColorists.find(item => item.accessCode === code);
551    if (colorist) {
552      return await UserService.getUserById(colorist.userID);
553    }
554    return null;
555  };
556
```

```
24    static async getUserById(id: string): Promise<IUser> {
25      const querySnapshot = await getDocs(collection(db, 'users'));
26      let userFromDb: IUser | null = null;
27
28      querySnapshot.forEach(doc => {
29        const data = doc.data() as IUser;
30        if (data.userID === id) userFromDb = { ...data, docId: doc.id };
31      });
32
33      if (!userFromDb) throw new Error();
34      return userFromDb;
35    }
36
```

Рисунок 3.12 - Реалізація отримання колориста за його унікальним кодом

Цей код складається з двох функцій. Перша функція getColoristByAccessCode призначена для отримання колориста за його кодом доступу. Друга функція getUserById використовується для отримання користувача за його ідентифікатором. Давайте розглянемо кожну частину коду більш детально.

Функція getColoristByAccessCode:Ця функція є асинхронною і отримує два параметри: code (код доступу) і allColorists (масив об'єктів, що представляють колористів).

Вона використовує метод find, щоб знайти колориста в масиві allColorists, який має вказаний accessCode.

Якщо колориста знайдено, викликається метод UserService.getUserById для отримання даних користувача за userID колориста і повертається результат. Якщо колориста не знайдено, повертається null.

Функція UserService.getUserById: Ця функція також є асинхронною і приймає ідентифікатор користувача (id). Вона використовує функції Firestore для отримання даних користувача з колекції 'users' за id.

Якщо користувача знайдено, дані отримуються з об'єкту doc, додається ідентифікатор документа doc.id і повертається об'єкт userFromDb. Якщо користувача не знайдено, викликається виняток.

Загалом, цей код призначений для отримання даних користувача на основі коду доступу колориста, шляхом перевірки коду доступу колориста та отримання його userID, а потім використання цього userID для отримання даних користувача з бази даних Firestore. Важливо враховувати, що код може викидати виняток, якщо не буде знайдено відповідного користувача в базі даних.

Після усіх цих дій, якщо колориста знайдено, накладну буде створено з відповідними товарами та даними, які було внесено до цього. Приклад сторінки де будуть наявні усі ці накладні наведено на рисунку 3.13

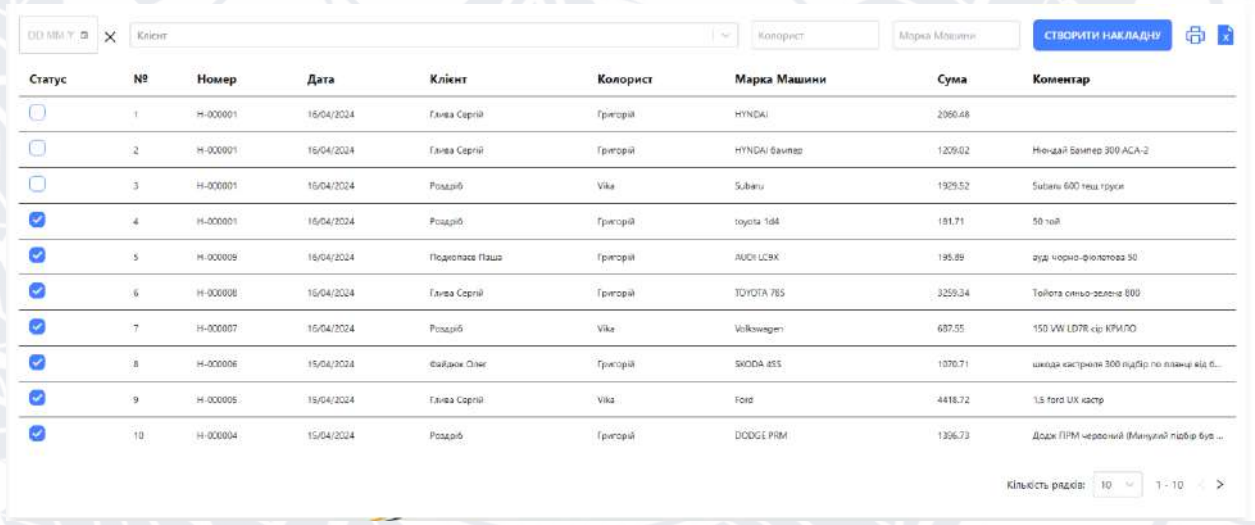


Рисунок 3.13 - Приклад сторінки зі створеними накладними

Після того, як колорист створив накладну, він має змогу її видалити. Видаляти колорист накладні може лише ті, які не є затвердженими. Приклад функціональності з видалення накладної наведено на рисунку 3.14

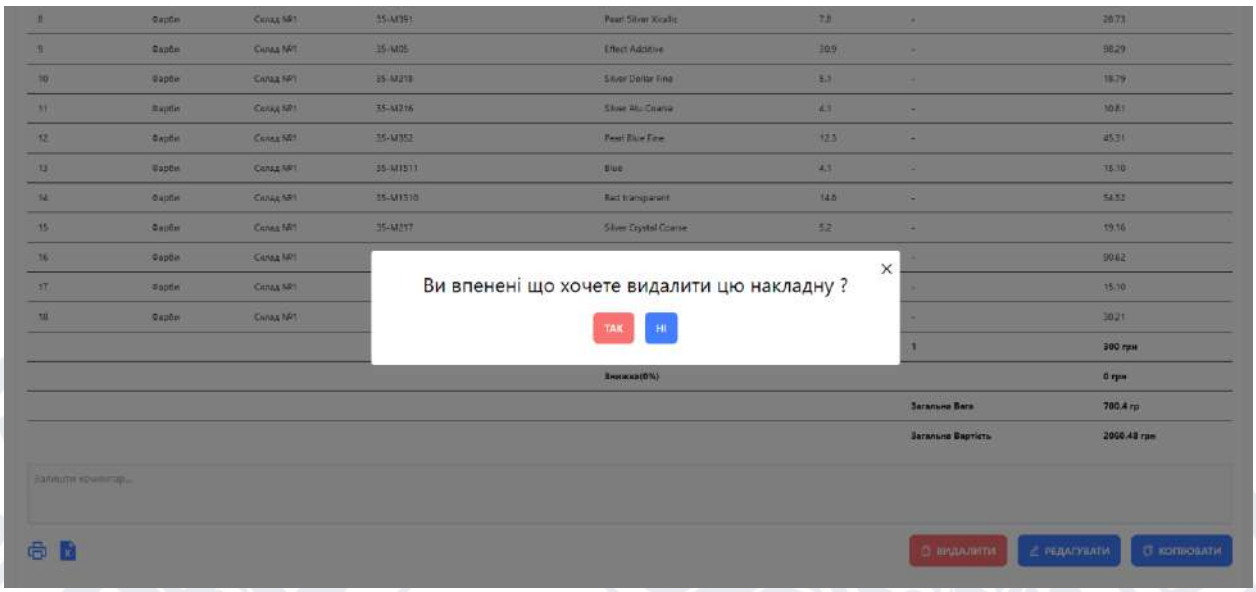


Рисунок 3.14 - Модальне вікно з підтвердженням видалення накладної

Можливість видалення накладних, лише ті, які були затверджені було зроблено задля того, щоб згодом не отримати мутацію даних в застосунку, адже від затверджених накладних безспосередньо залежить й нарахування заробітньої плати колориста. Приклад методу з видалення накладних наведено на рисунку 3.15

```
281  static async deleteInvoce(invoceDocId: string, documentNumber: string) {
282    await deleteDoc(doc(db, 'invoces', invoceDocId));
283    await DocumentsService.deleteDocument(documentNumber);
284    return invoceDocId;
285  }
```

Рисунок 3.15- Код з видалення накладної

Цей код відображає функцію deleteInvoce, яка видаляє накладну та документ, пов'язаний з цією накладною, за їхніми ідентифікаторами.

await deleteDoc(doc(db, 'invoces', invoceDocId));: Ця стрічка коду видаляє документ з колекції "invoices" за його ідентифікатором invoceDocId. Вона використовує функцію doc для отримання посилання на документ у Firestore, передаючи ім'я колекції та ідентифікатор документа. Функція deleteDoc виконує видалення цього документа з бази даних.

await DocumentsService.deleteDocument(documentNumber);: Ця стрічка коду викликає функцію deleteDocument з сервісу DocumentsService, передаючи номер документа для видалення.

Ця функція, відповідає за видалення документа з іншої таблиці.

return invoceDocId;:Після успішного видалення обох документів, функція повертає invoceDocId, що є ідентифікатором видаленого рахунку.

Функція з видалення накладної доступна лише для колориста, який має відповідні права – Супер Колорист, коли користувач з правами просто Колориста немає змоги цього зробити.

Також користувач може скопіювати відповідну накладну задля створення нової накладної на основі вибраної або ж відредагувати існуючу. Для останнього варіанту, колористу знову необхідно буде вказати свій код ідентифікатор задля підтвердження особи, або ж код того колориста, на якого буде створено накладну.

Ця функція призначена для редагування рахунку (інвойсу) у базі даних. Вона отримує об'єкт з параметрами для редагування, такими як дата акту, номер акту, клієнт, марка автомобіля, загальна ціна, знижка, послуга, примітка та інші. Спочатку функція витягує всі документи з колекції "invoices" у базі даних та знаходить той, який потрібно редагувати, використовуючи його унікальний ідентифікатор. Потім вона створює новий об'єкт з оновленими даними рахунку. Після цього вона оновлює відповідний документ у базі даних з новими даними. На виході функція повертає оновлений рахунок разом з його унікальним ідентифікатором. Ця функція є важливою для забезпечення актуальності даних та можливості редагування рахунків з подальшим оновленням інформації у системі.

Також слід зазначити, що редагувати накладні, також може лише колорист з статусом Супер Колорист й тільки ті накладні, які не є затвердженими, знову таки з метою збереження цілісності даних.

```
static async editInvoce({
  actDate,
  actNumber,
  data,
  client,
  carBrand,
  totalPrice,
  discount,
  service,
  note,
  id,
  number,
  createdDate,
  colorist,
  countOfServices,
  weight,
  servicePrice,
}: IEditInvocesProps): Promise<IInvoceDto> {
  const docs = await getDocs(collection(db, 'invoces'));
  let docId = '';

  docs.forEach(el => {
    const invoce = el.data() as IInvoceDto;
    if (invoce.id === id) docId = el.id;
  });
  });

  const editInvoce = {
    id,
    number,
    invoceNumber: actNumber,
    date: actDate.format('DD/MM/YYYY'),
    client,
    colorist,
    carBrand,
    price: totalPrice,
    discount,
    service,
    note,
    products: data,
    createdDate,
    countOfServices,
    isAprroved: false,
    weight,
    servicePrice,
  };
    servicePrice,
  };

  await updateDoc(doc(db, 'invoces', docId), editInvoce);

  return { ...editInvoce, docID: docId };
}
```

Рисунок 3.16 - Код з редагуванням накладної

Як ще одним прикладом взаємодії користувача з застосунком, а також безпекою, продемонструємо вкладку “Склади”, де колорист має змогу виконувати усі необхідні операції на товарами в складах, такі як додавааня товару, видалення його, редагування, переміщення з одного складу в інший, списати товар та виконати переоблік. Приклад сторінки склади наведено на рисунку 4.17

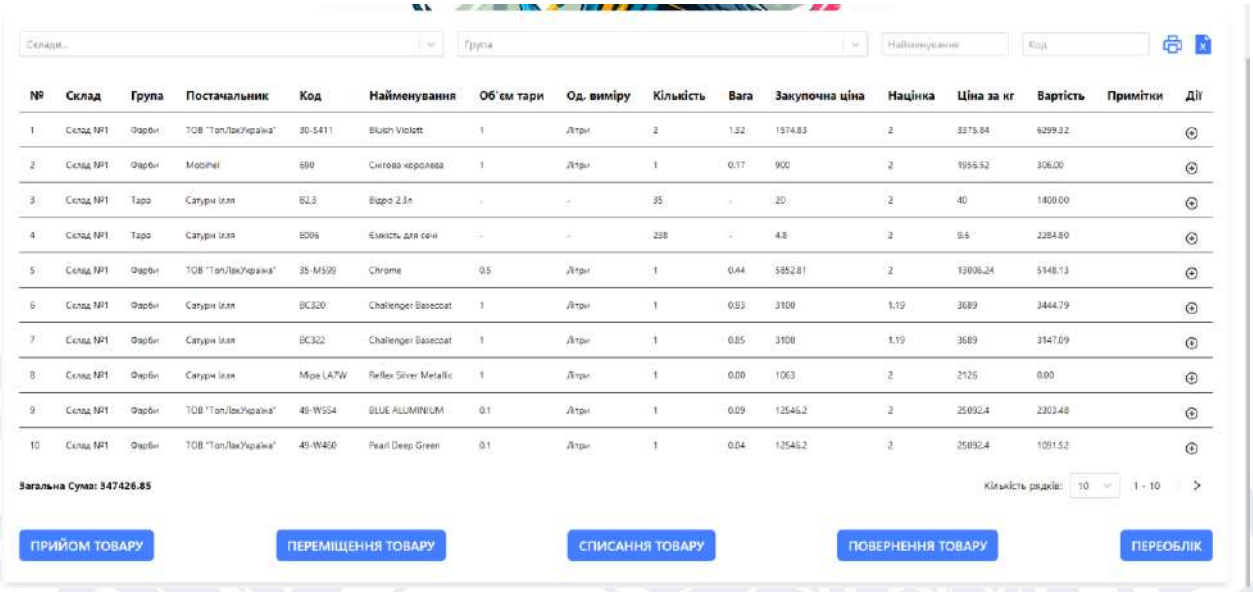


Рисунок 3.17 - Приклад сторінки «Склади»

Проаналізуємо наступний код, який дозволяє нам додати новий товар до складу

```
static async addNewProducts(
  data: IAcceptProductTableRowProps[],
  actDate: dayjs.Dayjs,
  actNumber: string,
  performer: IUser,
  note: string
) {
  const products: IProductDto[] = [];
  const res = await getDocs(collection(db, 'products'));
  const allProducts: IProductDto[] = [];
  res.forEach(el => allProducts.push({ ...(el.data() as IProductDto), docID: el.id }));

  for (let product of data) {
    const sameProduct = getEqualProduct(allProducts, product);
    if (sameProduct) {
      const updateProduct = await this.updateProduct(sameProduct, product);
      products.push(updateProduct);
    } else {
      const newProduct = await this.addNewProduct(product);
      products.push(newProduct);
    }
  }

  await DocumentsService.createNewDocument(actNumber, actDate, DOCUMENT_TYPE.ACT, performer, products, note);

  return products;
}
```

```
static async addNewProduct(data: IAcceptProductTableRowProps): Promise<IProductDto> {
  let provider;
  let group;

  if (typeof data.provider === 'object') {
    provider = data.provider as IProviderDto;
  } else {
    provider = await StorageService.createNewProvider(data.provider as string);
  }

  if ((data.group as IProductGroupDto).id) {
    group = data.group as IProductGroupDto;
  } else {
    group = await StorageService.createNewGroupProduct(data.group as string);
  }

  const newDoc = {
    ...data,
    provider,
    group,
  };

  const createdDoc = await addDoc(collection(db, 'products'), newDoc);

  if (createdDoc) {
    return { ...newDoc, docID: createdDoc.id };
  }
  throw new Error('Помилки при додаванні нового продукту');
}
```

Рисунок 3.18 - Приклад коду з додаванням нового продукту до складу

Цей код належить до класу, який містить статичний метод addNewProducts, який відповідає за додавання нових продуктів. Давайте розберемо його детально. На вхід addProducts функція приймає декілька аргументів такі як, нові продукти, дата створення акту про додавання продуктів, номер акту, користувач, який відповідно створив цей акт та нотатки. Згодом ми отримуємо усі продукти які наявні в NoSQL Firebase, далі проходимося по кожному продукту, який плануємо додати та перевіряємо чи цей продукт новий чи вже був доданий раніше подібний. У разі, якщо продукт вже існує його кількість, вага та остаточна ціна – інкремінтуються, якщо ж продукт новий – він просто додається в базу даних за допомогою методу addNewProduct. В цій функції ми перевіряємо чи існує вже вказаний постачальник чи його слід створити, це додано задля того, щоб користувач міг динамічно створювати нових постачальник, так само пророблюємо з групою товару. Після усіх кроків ми врешті додаємо продукт до Firestore. Приклад з можливістю додати новий товар на сторінку наведено на рисунку 3.19.

Рисунок 3.19 а) Приклад створення товару; частина 1

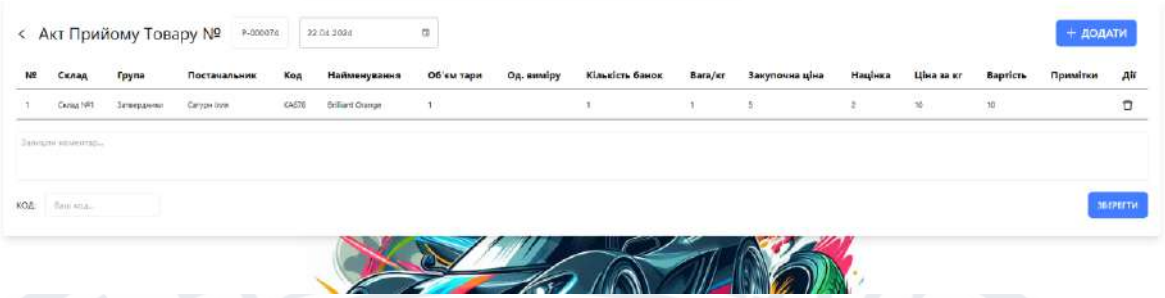


Рисунок 3.19 а) Приклад створення товару; частина 2

Коли колорист впевнений у правильності акту та наявності товару – він вводить відповідний код доступу, який ідентифікує його та дає змогу створити відповідний акт з певними товарами – після усіх цих кроків, товари з актом буде додано до NoSQL Firebase.

Як було зазначено вище – колорист має також зможу переміщати товар з одного складу на інших, це розраховано на майбутню перспективу задля клієнтів, в яких декілька складів й яким потрібно контролювати відповідну наявність на певному складі. Приклад даної функціональності з переміщенням товару наведено на рисунку 3.20.

```
186   static async updateProductsStoragePlace(
187     newPlaceProps: IUpdateProductsStorageProps[],
188     actNumber: string,
189     actDate: dayjs.Dayjs,
190     performer: IUser,
191     note: string
192   ) {
193     const products: IProductDto[] = await Promise.all(
194       newPlaceProps.map(async el => {
195         return await this.productUpdate(el.product.docID, {
196           ...el.product,
197           storage: el.newPlace,
198         });
199       })
200     );
201     await DocumentsService.createNewDocument(actNumber, actDate, DOCUMENT_TYPE.ACT, performer, products, note);
202   }
```

Рисунок 3.20 - Приклад оновлення місцерозташування товару

Як можемо помітити в функції updateProductStoragePlace – ми отримуємо декілька аргументів, такі як нове місцерозташування, номер акту, дата створення акту, власний акту та нотатки. Згодом ми проходимось по кожному товару та

оновлюємо йому його місцерозташування. Приклад сторінки де можна переміщувати товар наведено на рисунку 3.21

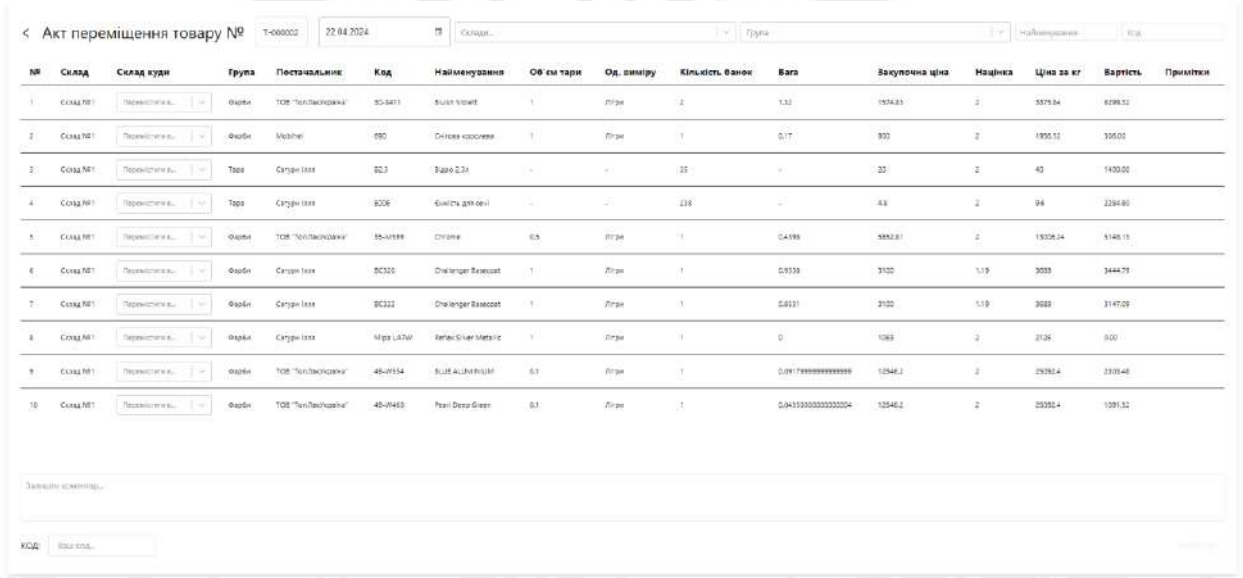


Рисунок 3.21 - Приклад сторінки з переміщенням товару

Знову таки, задля підтвердження особи – колорист мусить використати код доступу, після чого певні товару буде переміщено.

Час від часу, трапляється таке, що колорист просто забуває записати відповідні зміни в програму після контактування з клієнтом, тому наявність фізична на складі може не збігатися з тою, що вказана в програмі, тому для цього було створено також можливість провести акт переобліку, де колорист має змогу вказати контрольну вагу товару, його кількість, додати певну примітку, або ж взагалі видалити товар зі складу. Приклад реалізації такої функціональності наведено на рисунку 3.22

```
241  static async udpateProductInvenotry(
242    newData: IUpdateProductInventoryProps[],
243    actNumber: string,
244    actDate: dayjs.Dayjs,
245    performer: IUser,
246    note: string
247  ) {
248    const products = await Promise.all(
249      newData.map(async el => {
250        if (el.isDelete) {
251          await deleteDoc(doc(db, 'products', el.product.docID));
252          return {
253            ...el.product,
254            count: el.newCount,
255            weightTogether: el.newTotalWeight,
256            priceTogether: el.newTotalPrice,
257            note: el.newNote,
258          };
259        } else {
260          return await this.productUpdate(el.product.docID, {
261            ...el.product,
262            count: el.newCount,
263            weightTogether: el.newTotalWeight,
264            priceTogether: el.newTotalPrice,
265            note: el.newNote,
266          });
267        }
268      })
269    );
270
271    await DocumentsService.createNewDocument(actNumber, actDate, DOCUMENT_TYPE.ACT, performer, products, note);
272  }
273
```

Рисунок 3.22 - Приклад коду з переобліком товарів

Аналізуючи цей код, можемо зробити висновок, що updateProductInventory - один з абстрактних методів класу Products, який відповідно проводить переоблік для певної групи продуктів, які відповідно користувач обрав. Як на вхід, функція приймає масив з новими продуктами, номер акту, дату, колориста який створив цей акт та нотатки. Згодом ми проходимось по кожному продукту та оновлюємо продукт або ж видаляємо його, якщо колорист це зазначив перед цим, після усього цього створюємо акт з відповідними продуктами. Приклад сторінки де можливо здійснити переоблік товару наведено на рисунку 3.23

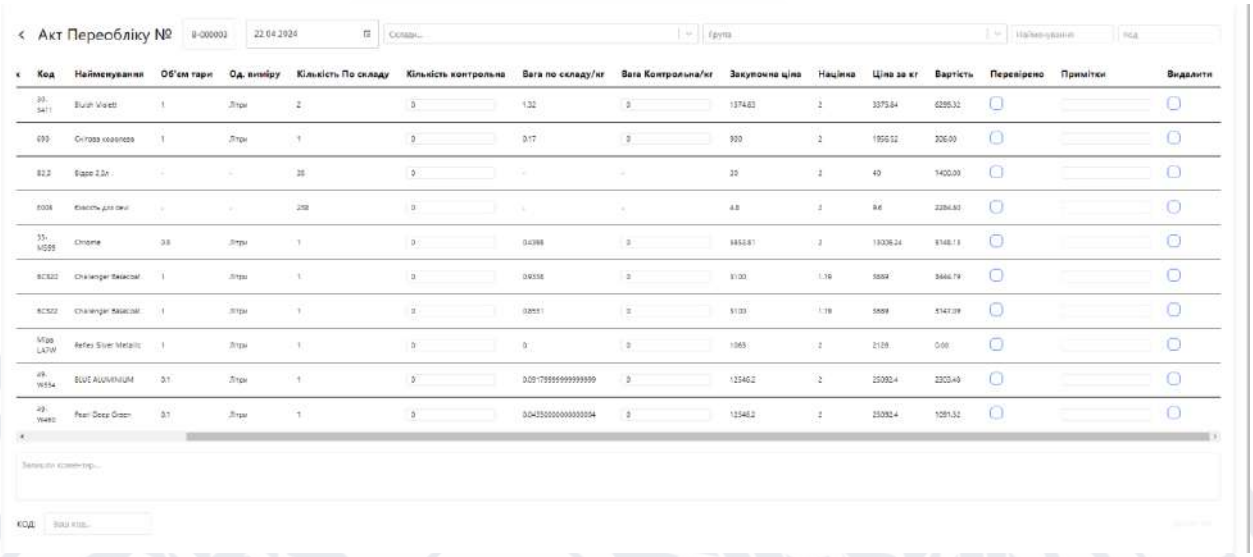


Рисунок 3.23 - Приклад сторінки з переобліком товару

ВИСНОВКИ

В результаті роботи поставлена мета була досягнута за рахунок розробки CRM системи з використанням NoSQL баз даних. Було проведено детальний аналіз актуальності розробки CRM-системи для колористів, що займаються підбором фарб для автомобілів. Дослідження показало, що у сучасних умовах високої конкуренції індивідуальний підхід до клієнтів є критично важливим. Розроблено детальний проект архітектури бази даних, яка відповідає потребам зберігання та обробки даних про клієнтів, історії замовлень, інформації про фарби та було реалізовано CRM система за допомогою NoSQL Firebase.

В роботі було проаналізовано сучасні методи та інструменти розробки CRM-систем, зокрема зосереджено увагу на використанні NoSQL баз даних для зберігання та обробки інформації. В рамках роботи було розглянуто існуючі рішення на ринку CRM-систем, їх переваги та недоліки, а також можливість адаптації цих рішень для специфічних потреб колористів. На основі проведеного аналізу було розроблено концептуальну модель CRM-системи для колористів з використанням NoSQL баз даних, яка враховує всі специфічні вимоги та особливості даної професійної діяльності.

Основною метою даної дипломної роботи було впровадження новітної технології NoSQL Firebase в CRM систему та дослідження перспектив розвитку даної технології, її особливості використання, переваги та недоліки й поріняння з конкурентами. Нам вдалося більш детальніше дізнатися з приводу тонкощів налаштування Firestore, особливості використання саме NoSQL баз даних в застсосунках та переваги таких підходів. Ми дослідили такі поняття як CRM система, яка є стратегіє управління взаєминами з клієнтами, NoSQL бази дані що є альтернативою традиційних SQL баз даних.

Також, більш детальніше розібрали екосистему NoSQL Firebase в даному випадку впровадження в CRM систему. Розібрано зручні та доступні API для взаємодії з базою даних з можливістю запису, видалення, редагування та читання будь яких даних.

Розглянули більш детальніше архітектуру проекту, структуру бази даних, основні аспекти побудови схем для NoSQL баз даних, а також структуру запитів до БД за допомогою Firebase API.

Використання NoSQL баз даних, зокрема Firebase, у веб проектах, має свої переваги, такі як швидкість розробки, зручність взаємодії та низькі витрати на утримання проекту. Аналіз показав, що такий підхід є актуальним та ефективним для сучасних веб-застосунків. Також, використання таких баз даних у сучасних веб-застосунках відкриває широкі можливості для розробників і бізнесу. Firebase вирізняється своєю зручністю у використанні та багатофункціональністю, що робить його привабливим вибором для реалізації різноманітних проектів Усі його переваги стають ключовим фактором у розробці сучасних веб-застосунків, дозволяючи розробникам швидко створювати функціональні та масштабні продукти з низькими витратами на утримання.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Why is NoSQL ? стаття URL: https://www.mongodb.com/nosql-explained (останнє звернення: 03.05.2024)

2. Introduction to NoSQL стаття URL: https://www.geeksforgeeks.org/introduction-to-nosql/ (останнє звернення: 03.05.2024)

3. ЩО ТАКЕ NOSQL БАЗИ ДАНИХ? стаття URL: https://freehost.com.ua/ukr/faq/wiki/chto-takoe-bazi-dannih-nosql/ (останнє звернення: 03.05.2024)

4. Що таке NoSQL? стаття URL: https://www.oracle.com/cis/database/nosql/what-is-nosql/ (останнє звернення: 03.05.2024)

5. Характеристики NoSQL баз даних стаття URL: https://javarush.com/ua/quests/lectures/ua.questhibernate.level19.lecture01 (останнє звернення: 03.05.2024)

6. NoSQL - переваги та недоліки нереляційних баз даних стаття URL: https://qagroup.com.ua/publications/nosql-perevagy-ta-nedoliky-nereliatcijnykh-baz-danykh/ (останнє звернення: 03.05.2024)

7. NoSQL vs SQL, й до чого тут теорема CAP стаття URL: https://dou.ua/lenta/articles/nosql-vs-sql/ (останнє звернення: 03.05.2024)

8. SQL vs. NoSQL: The Differences Explained + When to Use Each стаття URL: https://www.coursera.org/articles/nosql-vs-sql (останнє звернення: 03.05.2024)

9. NoSQL — no party. Що треба знати про нереляційні бази даних стаття URL: https://highload.today/uk/blogs/nosql-no-party-shho-treba-znati-pro-nerelyatsijni-bazi-danih/ (останнє звернення: 03.05.2024)

10. Офіційна документація Firebase. стаття URL: https://firebase.google.com/docs/firestore (останнє звернення: 03.05.2024)

11. Firestore: A Deep Dive into Firebase's NoSQL Database стаття URL: https://appmaster.io/blog/firestore-nosql-database (останнє звернення: 03.05.2024)

12. Is Firebase a NoSQL Database? стаття URL: https://www.lido.app/firebase/firebase-nosql (останнє звернення: 03.05.2024)

13. Comparing Firebase vs MongoDB стаття URL: https://www.mongodb.com/resources/compare/firebase-vs-mongodb (останнє звернення: 03.05.2024)

14. Firebase як бекенд для будь-яких застосунків, та як використовувати Firebase-сервіси стаття URL: https://dou.ua/forums/topic/44058/ (останнє звернення: 03.05.2024)

15. Introducing Firebase Data Connect стаття URL: https://firebase.blog/posts/2024/05/introducing-firebase-data-connect/ (останнє звернення: 03.05.2024)

16. FIRESTORE VS. REALTIME DATABASE: WHICH PERFORMS BETTER? стаття URL: https://estuary.dev/firestore-vs-realtime-database/ (останнє звернення: 03.05.2024)

17. Using Firebase for NoSQL DB стаття URL: https://medium.com/@juans_gro/using-firebase-for-nosql-db-779cb71f9c3c (останнє звернення: 03.05.2024)

18. Firebase: Realtime Database стаття URL: https://www.javatpoint.com/firebase-realtime-database (останнє звернення: 03.05.2024)

19. Benefits of using Firebase стаття URL: https://medium.com/@laxaar/10-benefits-of-using-firebase-1db8062e9f5b (останнє звернення: 03.05.2024)

20. Firebase vs. MySQL: Battle of the Databases стаття URL: https://www.integrate.io/blog/firebase-vs-mysql/ (останнє звернення: 03.05.2024)

21. Callback function документація URL: https://developer.mozilla.org/en-US/docs/Glossary/Callback_function (останнє звернення: 03.05.2024)

22. Що таке callback функції в JavaScript стаття URL: https://medium.com/@hexlet/%D1%87%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-callback-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F-%D0%B2-javascript-d8466b43f651 (останнє звернення: 03.05.2024)

23. JavaScript Callback Functions – What are Callbacks in JS and How to Use Them стаття URL: https://www.freecodecamp.org/news/javascript-callback-functions-what-are-callbacks-in-js-and-how-to-use-them/ (останнє звернення: 03.05.2024)

24. Using promises документація URL:https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises (останнє звернення: 03.05.2024)

25. Asynchronous JavaScript – How to Use Promises in Your JS Code стаття URL: https://www.freecodecamp.org/news/how-to-use-promises-in-javascript/ (останнє звернення: 03.05.2024)

26. How to Use Async/Await in JavaScript – Explained with Code Examples стаття URL: https://www.freecodecamp.org/news/javascript-async-await/ (останнє звернення: 03.05.2024)

27. Шмалюх В.А, Павлов Д.Л. Використання NoSQL баз даних в сучасних веб застосунках. Матеріали Всеукраїнської науково-практичної інтернет-конференції «Молодь в науці: дослідження, проблеми, перспективи (МН-2024)» 2024», Вінниця, 2024. URL: https://conferences.vntu.edu.ua/index.php/mn/mn2024/paper/view/21768 (дата звернення 21.05.2024)

28 Перелік національних стандартів для Перелік Національних стандартів України для створення, впровадження та супроводження автоматизованих і інформаційних систем http://nbuv.gov.ua/node/1469. (дата звернення 21.05.2024)

29. Уніфікована система організаційно-розпорядчої документації. Вимоги до оформлювання документів. ДСТУ 4163-2003; чинний від 01.09.2003. URL: http://zakon3.rada.gov.ua/rada/show/v0055609-03. (дата звернення 21.05.2024)

30. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлювання: ДСТУ 3008-2015 введ. 01.07.2017. К.: ДП «УкрНДНЦ», 2016. 26 с.

31. Інформація та документація Бібліографічне посилання Загальні положення та правила складання ДСТУ 8302:2015 введ. 01.07.2016. К.: ДП «УкрНДНЦ», 2016. 16 с.

32. Що таке стандарт ISO? URL: https://onmedu.edu.ua/shho-take-standart-iso-9001/. (дата звернення 21.05.2024)

33. ICO. URL: https://www.ukrcsm.kiev.ua/index.php/en/services-ua/standard-ua/inter-org-standardua/inter-org-iso-ua. (дата звернення 21.05.2024)

34. МЕЖДУНАРОДНЫЙ СТАНДАРТ ISO 9000. Cert academy. 2015. URL: http://isomanagement.com/wp-content/uploads/2018/09/ISO-9000-2015.pdf. (дата звернення 21.05.2024)

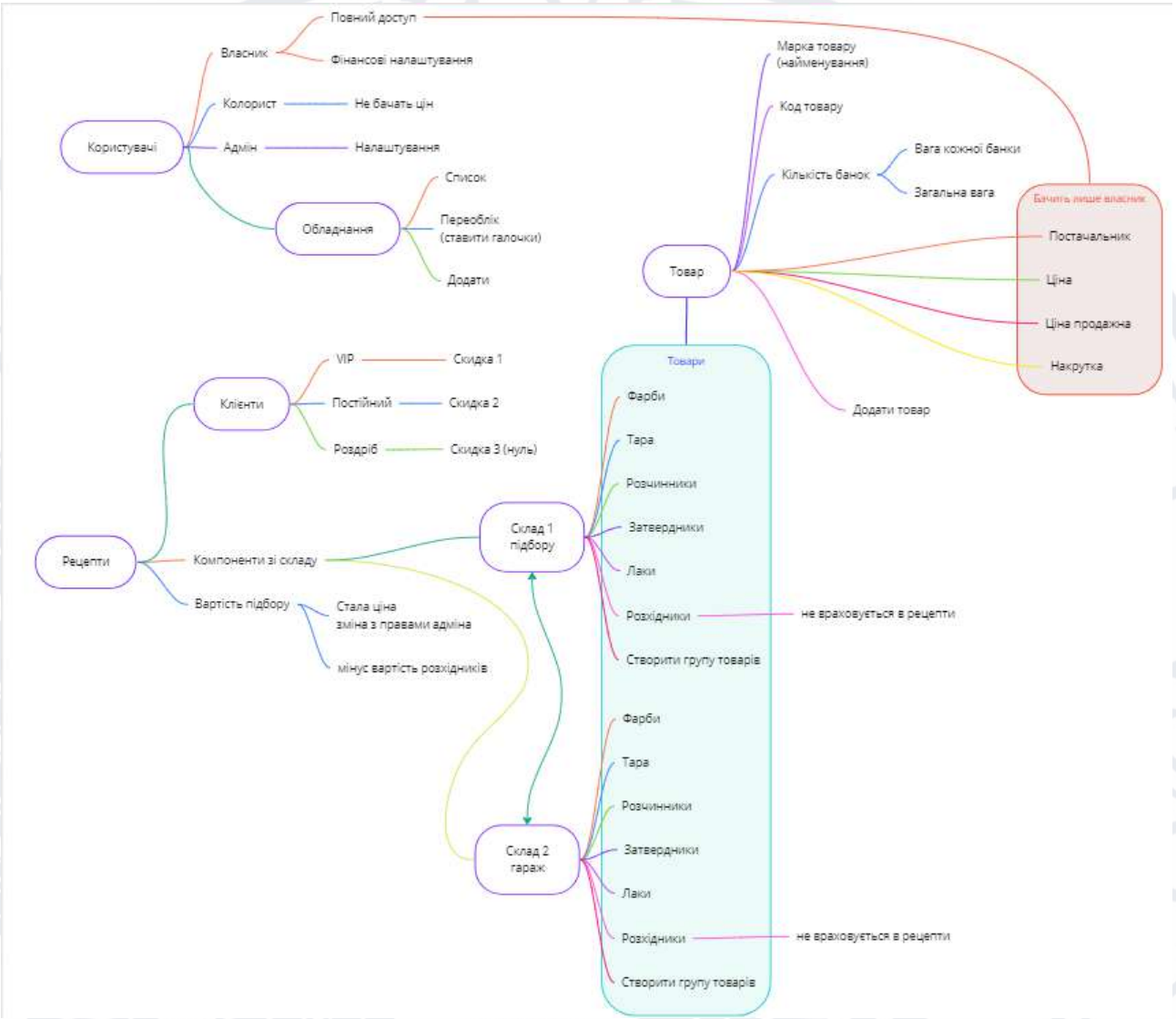
35. ДСТУ ISO/IEC/IEEE 16326:2015 Розроблення систем та програмного забезпечення. Процеси життєвого циклу. Керування проектами (ISO/IEC/IEEE 16326:2009, IDT) URL : http://online.budstandart.com/ua/catalog/docpage.html?id_doc=67052. (дата звернення 21.05.2024)

36. Методичні рекомендації до виконання, оформлення та захисту кваліфікаційної (бакалаврської) роботи здобувачами спеціальності 122 «Комп'ютерні науки» освітньої програми «Комп'ютерні науки». О. В. Зелінська, Т. В. Січко, Н. А. Потапова, К. О. Якубич. Вінниця: ДонНУ імені Василя Стуса, 2024. 27 с.

ДОДАТКИ

ДОДАТОК А

Загальний структура CRM системи



ДОДАТОК Б

Лістинг основних функцій

```
import { db } from '@/config/firebase.config';
import { approveInvoceHelper, approveInvoceColoristHelper } from
'@/helpers/invoce.helper';
import { invoceTotalPriceHelper } from '@/helpers/invoce.helper';
import { ColoristService } from '@/services/Colorist.service';
import { DocumentsService } from '@/services/Documents.service';
import { ProductsService } from '@/services/Products.service';
import { IClientDto, IInvoceDto } from '@/types/dto';
import { DOCUMENT_TYPE } from '@/types/enum';
import { GetInvoceSearchParams, ICreateNewInvoceProps,
IEditInvoceProps, PaginationTypes } from '@/types/interface';
import dayjs from 'dayjs';
import {
  addDoc,
  collection,
  deleteDoc,
  doc,
  getCountFromServer,
  getDocs,
  limit,
  orderBy,
  query,
  startAfter,
  updateDoc,
  where,
} from 'firebase/firestore';
import { v4 as uuidv4 } from 'uuid';
import { FinancesService } from '@/services/Finances.service';

export class InvoceService {
```

```
static async createNewInvoce({
  actDate,
  actNumber,
  colorist,
  data,
  client,
  carBrand,
  totalPrice,
  discount,
  service,
  note,
  countOfServices,
  weight,
  servicePrice,
}: ICreateNewInvocеProps): Promise<IInvoceDto> {
  const number = (await this.getNewInvoceRowNumber()) + 1;

  const id = uuidv4();
  const newDoc = {
    id,
    number,
    invoceNumber: actNumber,
    date: actDate.format('DD/MM/YYYY'),
    client,
    colorist,
    carBrand: carBrand,
    price: totalPrice,
    discount,
    service,
    note,
    countOfServices,
    createdDate: dayjs().format('YYYY-MM-DD HH:mm:ss'),
    products: data,
```

```
      isAprroved: false,
      weight,
      servicePrice,
    };

    await DocumentsService.createNewDocument(actNumber, actDate,
DOCUMENT_TYPE.INVOCE, colorist, []);
    const doc = await addDoc(collection(db, 'invoces'), newDoc);

    return { ...newDoc, docID: doc.id };
  }

  static async getInvocesCount() {
    try {
      const res = (await getCountFromServer(collection(db,
'invoces'))).data();
      return res.count;
    } catch (err) {
      throw err;
    }
  }

  static async getAllInvoces({
    lastVisible = null,
    pageSize = 10,
  }: PaginationTypes): Promise<{ invoces: IInvoceDto[];
newLastVisible: string; counts: number }> {
    let lastVisibleElement = null;

    if (lastVisible) {
      const req = query(collection(db, 'invoces'), where('id', '==',
lastVisible));
      const res = await getDocs(req);
```

```
      lastVisibleElement = res.docs[res.docs.length - 1];
    }

    const q = lastVisibleElement
      ? query(collection(db, 'invoces'), orderBy('createdDate',
'desc'), startAfter(lastVisibleElement), limit(pageSize))
      : query(collection(db, 'invoces'), orderBy('createdDate',
'desc'), limit(pageSize));

    const res = await getDocs(q);

    let invoces: IInvoceDto[] = [];
    let newLastVisible = null;

    res.forEach(el => {
      invoces.push({ ...(el.data() as IInvoceDto), docID: el.id });
    });

    newLastVisible = invoces[invoces.length - 1].id;

    for (let i = 0; i < invoces.length; i++) {
      invoces[i].number = i + 1;
    }
    const counts = await this.getInvocesCount();

    return { invoces, newLastVisible, counts };
  }

  static async getNewInvoceRowNumber() {
    const invoces = await getDocs(query(collection(db, 'invoces'),
orderBy('createdDate', 'asc'), limit(1)));
    let invoce: IInvoceDto | null = null;
```

```
    invoces.forEach(el => (invoce = el.data() as IInvoceDto));

    if (invoce) {
      const el = invoce as IInvoceDto;
      return el.number;
    }
    return 0;
  }

  static async editInvoce({
    actDate,
    actNumber,
    data,
    client,
    carBrand,
    totalPrice,
    discount,
    service,
    note,
    id,
    number,
    createdDate,
    colorist,
    countOfServices,
    weight,
    servicePrice,
  }: IEditInvoceProps): Promise<IInvoceDto> {
    const docs = await getDocs(collection(db, 'invoces'));
    let docId = '';

    docs.forEach(el => {
      const invoce = el.data() as IInvoceDto;
      if (invoce.id === id) docId = el.id;
```

```
    });

    const editInvoce = {
      id,
      number,
      invoceNumber: actNumber,
      date: actDate.format('DD/MM/YYYY'),
      client,
      colorist,
      carBrand,
      price: totalPrice,
      discount,
      service,
      note,
      products: data,
      createdDate,
      countOfServices,
      isAprroved: false,
      weight,
      servicePrice,
    };

    await updateDoc(doc(db, 'invoces', docId), editInvoce);

    return { ...editInvoce, docID: docId };
  }

  static async approveInvoce(invoce: IInvoceDto) {
    const invoces = await getDocs(collection(db, 'invoces'));
    let docID = '';

    invoces.forEach(el => {
      const invoceFromDB = el.data() as IInvoceDto;
```

```
      if (invoceFromDB.id === invoce.id) docID = el.id;
    });

    const { isValid, changedProducts } = await
approveInvoceHelper(invoce);
    if (isValid) {
      console.log(changedProducts);
      const { coloristDocId, newNumberCount, newSelectCountByCar,
newSelectCountByDetail, totalPaint, newSelectCountByHugeDetail } =
        await approveInvoceColoristHelper(invoce);

      if (coloristDocId) {
        await ColoristService.updateColoristCounts(
          coloristDocId,
          totalPaint,
          newNumberCount,
          newSelectCountByCar,
          newSelectCountByDetail,
          invoce.colorist.userID,
          newSelectCountByHugeDetail
        );
      }

      await Promise.all(
        changedProducts.map(
          async ({ id, newTotalPrice, newWeight, newCount }) =>
            await          ProductsService.productInvoceChange(id,
newTotalPrice, newCount, newWeight)
        )
      );

      await updateDoc(doc(db, 'invoces', docID), {
        isAprroved: true,
```

```
      });
    } else {
      throw new Error('Створення неможливе - сума одного із продуктів перевищує наявність на складі');
    }
  }

  static async getInvoce(id: string): Promise<IInvoceDto | null> {
    const invoces = await getDocs(query(collection(db, 'invoces'), where('id', '==', id), limit(1)));
    let invoce: IInvoceDto | null = null;
    invoces.forEach(el => (invoce = { ...(el.data() as IInvoceDto), docID: el.id }));
    return invoce || null;
  }

  static async getInvoceIdByDocumentId(docId: string): Promise<string | null> {
    //TODO:REWORK THIS ASAP;
    const doc = await DocumentsService.getOneDocument(docId);
    const docNumber = doc?.documentNumber || '';

    const invoces = await getDocs(query(collection(db, 'invoces'), where('invoceNumber', '==', docNumber), limit(1)));
    let invoce: IInvoceDto | null = null;
    invoces.forEach(el => (invoce = el.data() as IInvoceDto));

    return invoce!.id || null;
  }

  static async updateInvoceTotalPrice({ clientId }: { clientId?: string }): Promise<IInvoceDto[]> {
    const q = clientId
```

```
      ? query(collection(db, 'invoces'), where('isAprroved', '==',
false), where('client.id', '==', clientId))
      : query(collection(db, 'invoces'), where('isAprroved', '==',
false));
    const res = await getDocs(q);
    const invoces: IInvoceDto[] = [];
    res.forEach(el => invoces.push(el.data() as IInvoceDto));

    console.log(invoces);
    return await Promise.all(
      invoces.map(async el => {
        if (!el.isAprroved) {
          const invoceValue = await
FinancesService.getFinancialValue(el.service);
          const newTotalPrice = invoceTotalPriceHelper(el.price,
el.client.category.value, invoceValue, el.countOfServices);

          await updateDoc(doc(db, 'invoces', el.docID), {
            price: newTotalPrice,
          });
          return { ...el, totalPrice: newTotalPrice };
        }
        return el;
      })
    );
  }

  static async updateInvocesClient(newClient: IClientDto) {
    const res = await getDocs(query(collection(db, 'invoces'),
where('client.id', '==', newClient.id), where('isAprroved', '==',
false)));
    let invoces: IInvoceDto[] = [];
    res.forEach(el => invoces.push({ ...(el.data() as IInvoceDto),
docID: el.id }));
```

```
    for (let invoce of invoces) {
      if (!invoce.isAprroved) {
        await updateDoc(doc(db, 'invoces', invoce.docID), {
          client: newClient,
        });
      }
    }
  }
  static async deleteInvoce(invoceDocId: string, documentNumber:
string) {
    await deleteDoc(doc(db, 'invoces', invoceDocId));
    await DocumentsService.deleteDocument(documentNumber);
    return invoceDocId;
  }
  static async getInvoceByFilters({ client = '', carBrand = '', date
= '', colorist = '' }: GetInvoceSearchParams) {
    try {
      const res = await getDocs(collection(db, 'invoces'));
      let invoces: IInvoceDto[] = [];
      res.forEach(el => invoces.push(el.data() as IInvoceDto));
      invoces = invoces.filter(el => {
        return (
          el.client.fio.includes(client) &&
          el.carBrand.includes(carBrand) &&
          el.date.includes(date) &&
          el.colorist.firstName.includes(colorist)
        );
      });
      return invoces;
    } catch (err) {
      throw err;
    }
```

ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВАСИЛЯ СТУСА
ALMA MATER
NOMEN EST OMEN
ІМ'Я ЗОБОВ'ЯЗУЄ

ДОДАТОК В

Декларація щодо унікальності текстів роботи

та невикористання матеріалів інших авторів без посилань

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)