ВІЙСЬКОВИЙ ІНСТИТУТ ТЕЛЕКОМУНІКАЦІЙ ТА ІНФОРМАТИЗАЦІЇ імені ГЕРОЇВ КРУТ

Інформаційних технологій

(назва факультету)

Комп’ютерних інформаційних технологій

(повна назва кафедри)

ПОЯСНЮВАЛЬНА ЗАПИСКА

до курсової роботи

на тему: “ Розробка RESTfull додатку на основі технології Node.js для автоматизації діяльності обліку результатів виконання вправ з фізичної підготовки в підрозділі”

Виконав:

курсант 212 навчальної групи

старший солдат Олександр Мазур

Керівник:

викладач кафедри № 22

майор В.РОМАНЕНКО

Київ – 2024

АНОТАЦІЯ

Курсової роботи на тему:

“ Розробка RESTfull додатку на основі технології Node.js для автоматизації діяльності обліку результатів виконання вправ з фізичної підготовки в підрозділі”

Курсова робота містить: 54 с., 22 рис., 1 додаток, 15 джерел.

У курсовій роботі розглянуто актуальну проблему автоматизації обліку складського обліку речового майна та генерації звітів для фінансової служби військової частини. В умовах сучасних військових операцій ефективний облік і управління матеріальними ресурсами є критично важливими для забезпечення оперативної готовності та економічної ефективності. Існуючі ручні методи обліку матеріальних цінностей часто призводять до помилок, неефективного використання ресурсів та зайвих витрат часу.

Метою роботи є розробка програмного модуля, який автоматизує процеси обліку та звітності матеріальних цінностей, забезпечуючи зручний інтерфейс для користувачів і ефективні методи обробки даних. Основні вимоги до системи включають надійність, точність, безпеку даних та можливість масштабування для задоволення зростаючих потреб військової частини.

В рамках дослідження було проаналізовано існуючі рішення у сфері управління матеріальними ресурсами, визначено їхні переваги та недоліки. На основі цього аналізу було розроблено архітектуру веб-додатку для автоматизації обліку матеріальних цінностей, включаючи діаграму класів та діаграму взаємодії користувачів із системою.

Реалізовано прототип системи на платформі Node.js з використанням фреймворку Express та бази даних MongoDB. Серверна частина забезпечує надійну архітектуру для обробки даних і взаємодії з базою даних. Використання сесійного управління забезпечує ефективне управління доступом та захист даних користувачів.

# *SUMMARY*

*master's thesis on the topic:*

*"* *Software Module for Automating the Accounting of Material Assets and Report Generation for the Financial Service of a Military Unit"*

*The thesis addresses the pressing issue of automating the accounting of material assets and generating reports for the financial service of a military unit. In the context of modern military operations, efficient accounting and management of material resources are critically important for ensuring operational readiness and economic efficiency. Existing manual methods of accounting for material assets often lead to errors, inefficient use of resources, and unnecessary time expenditure.*

*The goal of this work is to develop a software module that automates the processes of accounting and reporting material assets, providing a user-friendly interface and efficient data processing methods. The main requirements for the system include reliability, accuracy, data security, and scalability to meet the growing needs of the military unit.*

*Within the framework of the research, existing solutions in the field of material resource management were analyzed, their advantages and disadvantages identified. Based on this analysis, the architecture of a web application for automating the accounting of material assets was developed, including a class diagram and a user interaction diagram with the system.*

*A prototype of the system was implemented on the Node.js platform using the Express framework and MongoDB database. The server side provides a reliable architecture for data processing and interaction with the database. The use of session management ensures effective access control and protection of user data.*

ЗМІСТ

[АНОТАЦІЯ 6](#_Toc169041676)

[*SUMMARY* 8](#_Toc169041677)

[СПИСОК УМОВНИХ СКОРОЧЕНЬ 11](#_Toc169041678)

[ВСТУП 12](#_Toc169041679)

[РОЗДІЛ 1 ОБҐРУНТУВАННЯ НЕОБХІДНОСТІ СТВОРЕННЯ *WEB* ПЛАТФОРМИ 13](#_Toc169041680)

[1.1 Процес обліку оцінювання нормативів у військових частинах та їх потреби 14](#_Toc169041685)

[1.2 Детальний аналіз потреб у системах автоматизації обліку 15](#_Toc169041690)

[Висновок до розділу 1 20](#_Toc169041704)

[2.1. Архітектура програмного рішення 21](#_Toc169041705)

[2.2 Архітектура *model-view-controller* (*MVC*) для веб-платформи 23](#_Toc169041706)

[2.3 Проектування візуальної складової застосунку 25](#_Toc169041707)

[2.4 Проектування backend частини за допомогою Node.js, Express та MongoDB 27](#_Toc169041708)

[Висновок до розділу 2 28](#_Toc169041709)

[РОЗДІЛ 3 РЕАЛІЗАЦІЯ ІНТЕРАКТИВНОЇ WEB-ПЛАТФОРМИ 30](#_Toc169041710)

[3.1 Реалізація серверної частини засобами *JavaScript*, *Node.js* та *MongoDB* 30](#_Toc169041711)

[3.2 Реалізація клієнтської частини з використанням *EJS* та *HTML*/*CSS*/*JS* 32](#_Toc169041712)

[3.3 Функціональні можливості системи інтерактивного тестування 34](#_Toc169041713)

[Висновок до розділу 3 39](#_Toc169041714)

[ВИСНОВКИ 40](#_Toc169041715)

[СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 41](#_Toc169041716)

[ДОДАТОК А 43](#_Toc169041717)

СПИСОК УМОВНИХ СКОРОЧЕНЬ

*API - Application Programming Interface*

*CRUD - Create, Read, Update, Delete*

*CSS – Cascading Style Sheets*

*Docker - Open Platform for Developing, Shipping, and Running Applications*

*ES – ECMAScript*

*HTML – Hyper Text Markup Language*

*HTTP – Hyper Text Transfer Protocol*

*JS – JavaScript*

*JSON - JavaScript Object Notation*

*JWT - JSON Web Token*

*MVC - Model-View-Controller*

*REST - Representational State Transfer*

*SPA – Single Page Application*

*UI – User interface*

*UML – Unified Modeling Language*

*URL – Uniform Resource Locator*

ООП – об’єктно-орієнтоване програмування

ПЗ – програмне забезпечення

ПК – персональний комп’ютер

СКБД – система керування базами даних

# ВСТУП

Автоматизація обліку результатів виконання вправ з фізичної підготовки та генерація звітів для фінансової служби військової частини є критично важливими елементами забезпечення оперативної готовності та економічної ефективності військових операцій. У сучасних умовах військових дій ефективне управління матеріальними ресурсами набуває все більшого значення. Проте, існуючі ручні методи обліку часто призводять до помилок, неефективного використання ресурсів та зайвих витрат часу. Такі недоліки не тільки знижують ефективність управління, але й можуть мати серйозні наслідки для фінансової стабільності військової частини.

Відповідно до цих викликів, ця робота спрямована на розробку програмного модуля, який автоматизує процеси обліку результатів виконання вправ з фізичної підготовки та генерації звітів для фінансової служби військової частини. Така система сприятиме підвищенню ефективності управління матеріальними ресурсами та забезпеченню надійної звітності.

Об’єкт роботи: процес обліку результатів виконання вправ з фізичної підготовки у військовій частині.

Предмет дослідження: інформаційні технології та системи управління матеріальними ресурсами в умовах військових операцій.

Мета дипломної роботи: розробка веб-додатку, що дозволить ефективно управляти та сприяти точному та своєчасному генеруванню звітів для фінансової служби військової частини.

Досягнення цієї мети вимагає вирішення наступних завдань:

1) Аналіз існуючих методів обліку результатів виконання вправ з фізичної підготовки та генерації звітів;

2) Розробка архітектури інформаційної системи для автоматизації обліку результатів виконання вправ з фізичної підготовки;

3) Імплементація прототипу веб-додатку на платформі Node.js із застосуванням фреймворку Express та бази даних MongoDB.

# РОЗДІЛ 1 ОБҐРУНТУВАННЯ НЕОБХІДНОСТІ СТВОРЕННЯ *WEB* ПЛАТФОРМИ

# У контексті забезпечення ефективного управління результатів виконання вправ з фізичної підготовки у військових частинах, створення веб-платформи для автоматизації обліку результатів виконання вправ з фізичної підготовки та генерації звітів для фінансової служби є вкрай необхідним. Існуючі ручні методи обліку часто призводять до помилок, неефективного використання ресурсів та зайвих витрат часу. Ці проблеми ускладнюють процес управління ресурсами і можуть мати серйозні наслідки для фінансової стабільності військової частини.

# На сучасному етапі, відсутність єдиної, інтегрованої системи, яка б дозволяла фінансовій службі військової частини ефективно управляти матеріальними цінностями, може призводити до затримок у обліку, неточностей у звітності та зниження ефективності управління ресурсами. Введення такої платформи дозволить уникнути багатьох ризиків, пов’язаних із обліком матеріальних цінностей, і значно покращить координацію між різними підрозділами військової частини.

# Створення платформи на базі веб-технологій забезпечить постійний доступ до бази даних матеріальних цінностей з будь-якого пристрою, що має підключення до інтернету, тим самим підвищуючи мобільність та гнучкість фінансових служб. Це стане особливо актуальним для оперативного управління ресурсами у польових умовах, де швидкість доступу до інформації є критично важливою.

# Таким чином, створення цієї платформи не лише відповідає актуальним вимогам часу, але й закладає фундамент для підвищення ефективності фінансових операцій у військових частинах, вдосконалюючи процеси координації та обліку матеріальних цінностей.

## Процес обліку результатів виконання вправ з фізичної підготовки у військових частинах та їх потреби

## Основним завданням в рамках обліку результатів виконання вправ з фізичної підготовки у військових частинах є забезпечення високого рівня точності, надійності та оперативної готовності. Освітній процес у таких спеціалізованих умовах є комплексом навчально-практичних заходів, спрямованих на навчання, засвоєння та розширення знань і навичок у сфері управління результатів виконання вправ з фізичної підготовки. Цей процес охоплює теоретичне навчання, практичні заняття, симуляції та тренування в реальних умовах, які імітують умови бойових дій або надзвичайних ситуацій.

## Освіта у цій сфері ґрунтується на принципах науковості, де основну увагу приділяють доказовим практикам, а також гнучкості, що дозволяє адаптувати навчальні програми під змінні умови сучасних військових конфліктів та місій. Основною метою є забезпечення персоналу необхідними компетенціями для ефективного виконання завдань у рамках управління матеріальними цінностями, зокрема оперативне реагування на потреби у ресурсах, управління запасами та здатність до швидкого прийняття рішень.

## Освітній процес для персоналу також включає вивчення міжнародних стандартів управління результатів виконання вправ з фізичної підготовки, що є необхідним для координації з міжнародними партнерами і ефективної взаємодії у міжнародних операціях. Це також включає освоєння спеціалізованих інформаційних систем, які є важливими для обміну інформацією, здійснення дистанційних консультацій та управління даними під час операцій.

## Таким чином, потреба у розробці веб-платформи для автоматизації обліку результатів виконання вправ з фізичної підготовки є логічним та необхідним кроком для підвищення ефективності освітнього процесу. Цей інструмент дозволить персоналу швидко ідентифікувати та обліковувати результатів виконання вправ з фізичної підготовки, забезпечуючи точність і швидкість обліку в умовах міжнародного контексту, що в кінцевому результаті сприятиме кращій підготовці та вищій готовності до фінансових операцій у військовій частині.

## 1.2 Детальний аналіз потреб у системах автоматизації обліку результатів виконання вправ з фізичної підготовки

## У рамках військових операцій виникає критична потреба у високоефективних інформаційних системах для автоматизації обліку результатів виконання вправ з фізичної підготовки. Розглянемо детальніше ключові вимоги до таких систем:

## Швидкість і точність. В умовах обліку результатів виконання вправ з фізичної підготовки, швидкість доступу до точної інформації є життєво важливою. Система має забезпечувати миттєвий доступ до бази даних з високою точністю обліку, уникнення будь-яких помилок може бути критичним для ефективного управління ресурсами.

## Безпека даних. З урахуванням конфіденційності інформації, система має включати сучасні методи шифрування і захисту даних. Це забезпечить захист від несанкціонованого доступу та злому, гарантуючи, що конфіденційна інформація залишається безпечною в будь-яких умовах.

## Адаптивність та гнучкість. Система повинна бути досить гнучкою, щоб адаптуватися до різноманітних стандартів обліку результатів виконання вправ з фізичної підготовки різних країн. Вона має дозволяти легке оновлення інформації та швидке внесення змін до бази даних, що відображає зміни у стандартах та протоколах.

## Масштабованість. З огляду на велику кількість учасників облікових операцій, система повинна мати високу масштабованість і здатність одночасно обслуговувати значну кількість запитів без зниження продуктивності або швидкості.

## Інтеграція з іншими системами. Важливою характеристикою є можливість інтеграції системи з іншими інформаційними системами, які використовуються в рамках військових операцій. Це дозволяє обмінюватися важливою інформацією між різними платформами та забезпечує цілісність даних.

## Інтерфейс користувача. Інтерфейс повинен бути інтуїтивно зрозумілим і простим у використанні для персоналу з різним рівнем технічної підготовки. Це мінімізує помилки у введенні даних і забезпечує ефективність використання системи в польових умовах.

## Розробка системи, яка задовольняє ці вимоги, може значно покращити результати управління результатів виконання вправ з фізичної підготовки у військових частинах, забезпечуючи швидкий і безпечний доступ до важливої інформації, необхідної для ефективного управління ресурсами. Впровадження цієї системи сприятиме створенню єдиного інформаційного простору для фінансових служб, дозволяючи миттєво обмінюватися даними про результатів виконання вправ з фізичної підготовки, їх запаси та використання.

## Також, інтеграція системи із логістичними платформами дозволить оптимізувати запаси результатів виконання вправ з фізичної підготовки у військових частинах та на місцях проведення операцій. Це забезпечить не тільки швидке надання необхідних ресурсів, але й ефективне їх використання, мінімізацію відходів та зниження витрат.

## Окрім того, система зможе надавати детальні звіти з використання ресурсів, базуючись на міжнародних стандартах та дослідженнях. Така інформація буде вкрай корисною для фінансового персоналу, який часто стикається з необхідністю приймати швидкі рішення в умовах обмеженого доступу до спеціалізованих баз даних.

## Інтеграція системи із мобільними додатками та іншими інформаційними інструментами забезпечить персоналу доступ до необхідних даних навіть у віддалених і недоступних місцях проведення операцій. Це дозволить підвищити рівень мобільності та гнучкості фінансових служб, значно підсилюючи їх здатність реагувати на екстрені ситуації.

## Загалом, впровадження такої системи стане значним кроком вперед у стандартизації та уніфікації процесів обліку матеріальних цінностей у військових частинах, забезпечуючи вищий рівень управління ресурсами та збереження фінансової стабільності.

1.3 Аналіз існуючих реалізацій систем автоматизації обліку матеріальних цінностей

У контексті управління матеріальними цінностями у військових частинах виникає необхідність у спеціалізованих інформаційних системах для автоматизації обліку. Наразі існує кілька платформ, які пропонують функціональність, подібну до необхідної для нашого проекту. Ось кілька прикладів, які можуть використовуватися для аналогічних задач:

SAP ERP - це комплексна платформа управління ресурсами, яка включає модулі для обліку матеріальних цінностей. Вона забезпечує інтеграцію з іншими системами, має високий рівень безпеки та підтримує багатомовність. На рисунку 1.1 зображений інтерфейс SAP ERP.

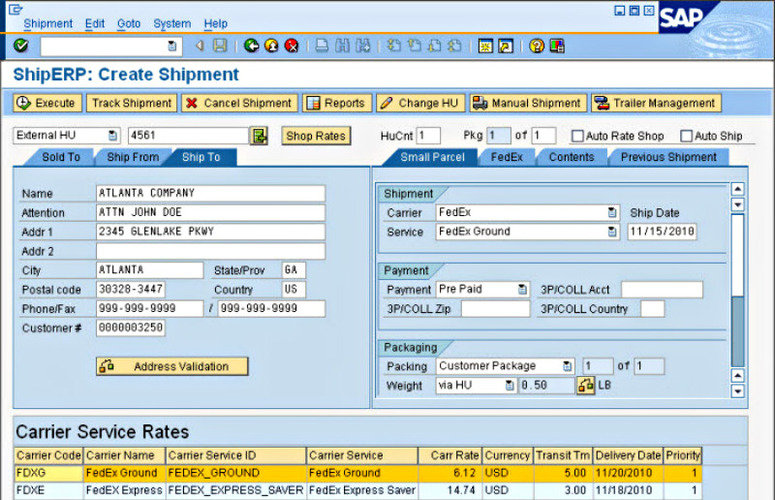


Рисунок 1.1 – Інтерфейс *SAP ERP*

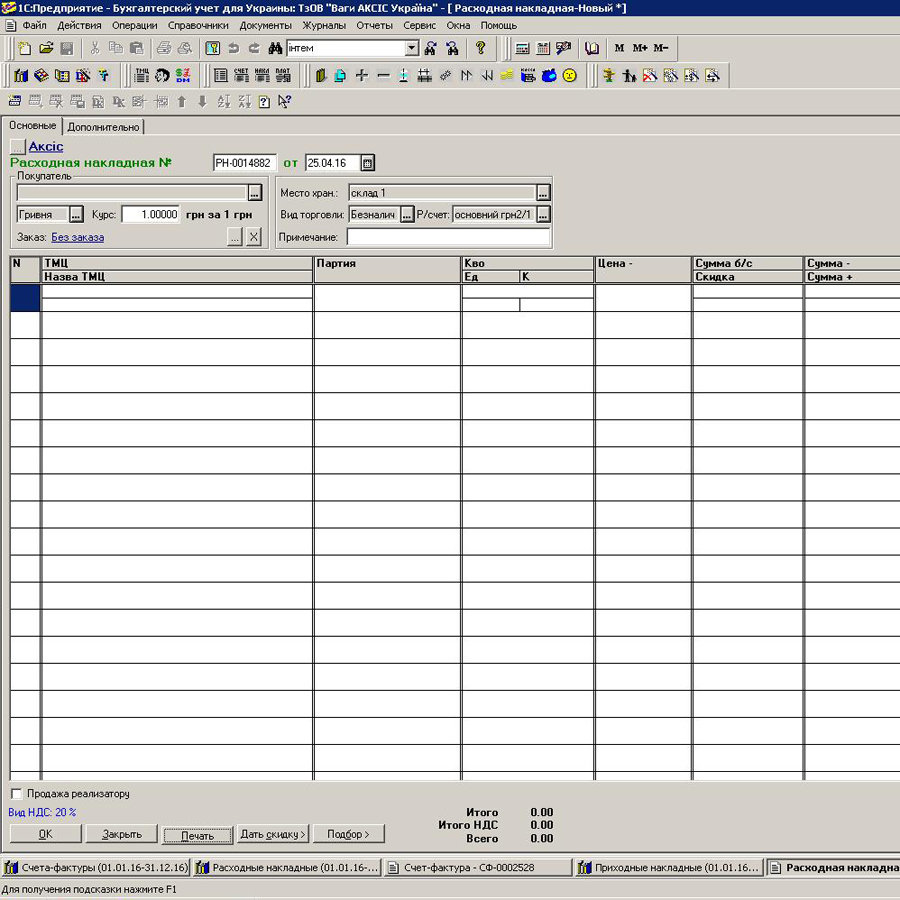
1C:Підприємство - це популярна платформа для управління ресурсами в Україні та країнах СНД. Вона забезпечує облік матеріальних цінностей, генерацію звітів та інтеграцію з іншими системами. На рисунку 1.2 зображено головне меню 1C:Підприємство.

Рисунок 1.2 – Головне меню *1С:Підприєство*

1.4. Обґрунтування створення веб-додатку з використанням Node.js, Express та MongoDB, імплементація у Docker контейнери

У контексті військових операцій існує висока потреба в надійних і доступних засобах для обліку матеріальних цінностей. Хоча інтернет-з'єднання може бути нестабільним або відсутнім у польових умовах, розробка веб-додатку, який можна також ефективно використовувати локально, забезпечує гнучкість та доступність інформації, коли це найбільш необхідно.

Локальне розгортання веб-додатку може бути реалізовано за допомогою використання Docker контейнерів, що дозволяє налаштувати і запустити систему на будь-якій сумісній машині без залежності від зовнішніх серверів. Це забезпечує високий рівень відтворюваності середовища та ізоляції, знижуючи ризики, пов'язані з варіативністю місцевих конфігурацій.

Node.js вибрано через його асинхронну природу та високу продуктивність, що є важливим для обробки великої кількості запитів у реальному часі. Express є гнучким фреймворком для Node.js, який забезпечує швидку розробку серверної частини додатку.

MongoDB використовується як система управління базами даних через її високу продуктивність, надійність та підтримку складних запитів, які можуть виникнути при роботі з великими масивами даних. MongoDB також забезпечує високий рівень сумісності з Node.js та може бути легко інтегрована з Docker, що забезпечує легкість розгортання та масштабування. Імплементація веб-додатку у *Docker* контейнери надає низку переваг:

- Ізоляція: Кожен аспект додатку може бути ізольованим у своєму контейнері, зменшуючи взаємозалежність компонентів та спрощуючи управління.

- Відтворюваність: *Docker* забезпечує однакові умови розгортання на різних машинах, що дуже важливо для консистенції поведінки додатку.

- Масштабованість: З *Docker* можна легко масштабувати систему, додаючи або видаляючи контейнери відповідно до потреб користувача.

- Безпека: Ізоляція в *Docker* також сприяє безпеці, оскільки збої або вразливості в одному контейнері не поширюються на інші.

## Ці технічні рішення дозволяють забезпечити надійну, безпечну і ефективну платформу для військових частин, гарантуючи доступність критично важливої інформації незалежно від зовнішніх обставин.

## Висновок до розділу 1

Розробка веб-додатку для автоматизації обліку матеріальних цінностей та генерації звітів для фінансової служби військової частини є критично важливою задачею, з огляду на потребу швидкої ідентифікації та обліку матеріальних цінностей в умовах військових операцій. Цей додаток дозволить фінансовому персоналу ефективно взаємодіяти між різними підрозділами, забезпечуючи точність та швидкість обліку.

Аналіз існуючих платформ виявив ряд обмежень, включно з залежністю від стабільного інтернет-з'єднання та необхідністю висококваліфікованого персоналу для їхнього налаштування та обслуговування. Водночас, використання спеціалізованих систем, таких як Node.js, Express та MongoDB, дозволяє забезпечити високу продуктивність, безпеку даних та можливість інтеграції з іншими системами.

Імплементація веб-додатку в рамках Docker контейнерів надає значні переваги, такі як ізоляція середовищ, гарантована відтворюваність, легке масштабування та зручність у розгортанні. Це не тільки підвищує безпеку і надійність додатку, але й забезпечує його автономну роботу в умовах обмеженого або відсутнього інтернет-з'єднання, що є критично важливим для польових умов та віддалених районів.

Завершення цього розділу підкреслює необхідність подальшого розроблення веб-додатку для автоматизації обліку матеріальних цінностей, здатного вирішити проблеми існуючих рішень та забезпечити потреби фінансових служб військових частин.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ІНТЕРАКТИВНОЇ WEB-ПЛАТФОРМИ *ДЛЯ ОБЛІКУ МАТЕРІАЛЬНИХ ЦІННОСТЕЙ*

У контексті потреби підтримки фінансових служб військових частин під час обліку матеріальних цінностей, виявлено суттєві недоліки існуючих інформаційних систем. Ці виклики обумовили необхідність створення нової, більш адаптивної інтерактивної веб-платформи. Проект автоматизації обліку матеріальних цінностей має на меті вирішення цих проблем за допомогою сучасних технологій веб-розробки.

Для розробки веб-додатку було обрано технологічний стек, що включає Node.js, Express, MongoDB, HTML, CSS та JavaScript. Цей вибір зумовлений наступними перевагами:

- Node.js та Express: Забезпечують потужні можливості для створення надійних та масштабованих серверних додатків. Express включає модулі, що дозволяють ефективно управляти маршрутизацією та взаємодією з клієнтською частиною*.*

- MongoDB: Використання MongoDB дозволяє забезпечити гнучкість та масштабованість у роботі з базами даних, особливо при обробці великих обсягів інформації.

- HTML, CSS та JavaScript: Ці технології є стандартом для створення сучасних інтерактивних веб-інтерфейсів, забезпечуючи гнучкість у дизайні та взаємодії з користувачем.

## 2.1. Архітектура програмного рішення

Архітектура програмного забезпечення – це спосіб структурування програмної системи, що є абстракцією елементів у певних фазах роботи системи. Система може бути структурована на різних рівнях абстракції та мати кілька фаз роботи, кожна з яких має свою незалежну архітектуру.

Архітектура програмного забезпечення є дослідженням, що виявляє оптимальний спосіб поділу системи на взаємопов'язані частини. Вона включає вивчення способів, якими ці частини взаємодіють та комунікують, розробку окремих компонентів і способи збереження цієї інформації через формальні або неформальні записи.

Проектування архітектури веб-додатку для автоматизації обліку матеріальних цінностей включає:

Модуль управління користувачами: Забезпечує аутентифікацію та авторизацію користувачів, включаючи фінансових працівників та адміністраторів.

Модуль обліку матеріальних цінностей: Відповідає за обробку запитів на облік та забезпечення доступу до бази даних матеріальних цінностей.

Цей розділ задає курс на детальне проектування та реалізацію веб-додатку, який стане надійним інструментом для фінансових служб, забезпечуючи їх необхідними інструментами для ефективної роботи.

Архітектура проекту виглядатиме наступним чином:

Клієнт - *HTML*/*CSS*/*JavaScript*: Відповідає за структуру та стилізацію інтерфейсу користувача. *JavaScript* забезпечує інтерактивність інтерфейсу. *EJS*: Використовується для динамічного рендерингу *HTML* сторінок на стороні сервера, забезпечуючи швидку відповідь і оновлення даних без перезавантаження сторінки.

Сервер - *Node.js:* Основа серверної частини, що забезпечує розробку та управління бізнес-логікою. Express.js: Фреймворк для Node.js, який контролює HTTP запити і відповіді, відповідаючи за маршрутизацію та взаємодію з клієнтською частиною. Passport.js: Бібліотека для Node.js, яка забезпечує безпеку додатку, включаючи аутентифікацію та авторизацію. Mongoose: Бібліотека для Node.js, яка відповідає за взаємодію з базою даних MongoDB, включаючи CRUD операції.

База даних - *MongoDB*: Система управління базами даних, яка зберігає всі дані про користувачів, матеріальні цінності та іншу важливу інформацію.

Ця архітектура забезпечує масштабованість та гнучкість додатку, дозволяючи легко розширювати та модифікувати його функціональність залежно від потреб користувачів та нових вимог.

## 2.2 Архітектура *model-view-controller* (*MVC*) для веб-платформи

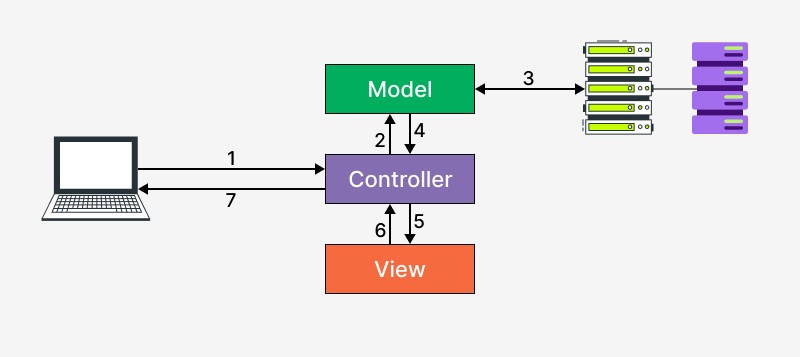
Для розробки інтерактивної веб-платформи було прийнято рішення використовувати архітектуру модель-вид-контролер (MVC). Цей підхід дозволяє ефективно розділити логіку програми на три основні компоненти, що спрощує управління кодом та підтримку додатку. Основна мета використання MVC полягає у забезпеченні високої гнучкості та легкої масштабованості проекту.

Рисунок 2.1 – Архітектура MVC

Модель (Model):

Визначає основну бізнес-логіку веб-додатку та взаємодіє з базою даних. У контексті обліку матеріальних цінностей, модель відповідає за зберігання та управління даними про матеріальні цінності, інформацію про користувачів та інші важливі бізнес-процеси. Зберігання даних відбувається у базі даних MongoDB, що забезпечує високу продуктивність та надійність.

Вид (*View*):

Реалізує інтерфейс користувача, що відображається у веб-браузері. HTML, CSS та JavaScript використовуються для створення сучасного та відгукового дизайну, що забезпечує високу зручність користувачів.

Контролер (*Controller*):

Обробляє вхідні запити від користувачів, керує моделями, ініціює необхідні зміни у моделях та відправляє дані до виду. Використання Express у якості фреймворку для контролерів дозволяє ефективно управляти роутингом, обробкою запитів та відповідей.

Основні переваги використання *MVC* включають:

- Чітке розділення відповідальності: Кожен компонент системи (модель, вид, контролер) відповідає за свої специфічні завдання, що спрощує управління кодом та знижує ризик помилок.

- Гнучкість у тестуванні: Компоненти можна тестувати незалежно один від одного, що підвищує якість та надійність програмного продукту.

- Спрощення колаборації: *MVC* дозволяє розробникам працювати одночасно над різними частинами проекту без взаємного втручання, що є особливо важливим для великих команд.

Для підкреслення переваг *MVC*, було зроблено порівняння з іншими популярними архітектурами:

- Монолітична архітектура: Традиційно використовується для створення єдиного, неділимого блоку програми, що може ускладнити масштабування та оновлення окремих частин системи. На відміну від монолітичних систем, *MVC* дозволяє окремо масштабувати та оновлювати кожен компонент, забезпечуючи більш ефективне управління ресурсами.

- Серверний рендеринг (наприклад, *ASP.NET Web Forms*): Хоча цей підхід дозволяє швидко розробляти додатки, він часто призводить до тісного зв'язування між логікою представлення та бізнес-логікою, що ускладнює тестування та модифікацію. *MVC* вирішує цю проблему за допомогою чіткого розділення логіки та інтерфейсу.

- *SPA*-архітектура (*Single Page Application*): Хоча *SPA* надає більш динамічний користувацький інтерфейс, управління станом та синхронізація між клієнтом і сервером можуть бути складними. MVC забезпечує більш строгий контроль над даними та взаємодією між сервером і клієнтом.

## 2.3 Проектування візуальної складової застосунку

Розробка візуальної частини платформи є ключовою для забезпечення інтуїтивно зрозумілого та ефективного інтерфейсу користувача. Використання HTML, CSS і JavaScript дозволяє створити динамічні, адаптивні та доступні веб-сторінки, які ефективно взаємодіють із бекендом розробленого веб-додатку.

Рисунок 2.2 – *HTML*, *CSS* і *JavaScrip*t

HTML (*HyperText Markup Language*) створює структурний каркас веб-сторінки. У контексті *Pharmaceutical Thesaurus*, *HTML* використовується для розміщення елементів інтерфейсу, таких як форми пошуку, списки лікарських засобів та інформаційні блоки. *EJS* розширює можливості *HTML*, дозволяючи інтегрувати дані моделі безпосередньо у *HTML*-шаблони, що забезпечує динамічне оновлення вмісту без перезавантаження сторінки.

*CSS* (*Cascading Style Sheets*) використовується для оформлення *HTML*-елементів, визначаючи їх візуальний стиль та розташування. Для підвищення продуктивності та гнучкості стилізації веб-платформи використовується *LESS*, препроцесор *CSS*, який дозволяє використовувати змінні, міксини, вкладені правила та інші покращення. *LESS* спрощує управління стилями, роблячи код легшим для підтримки та розширення.

*JavaScript* відіграє важливу роль у розробці інтерактивних аспектів веб-додатку. Він використовується для додавання динамічних ефектів, обробки подій на сторінці та інтеракції з користувачем. Наприклад, *JavaScript* може бути використаний для обробки введення користувача в формах пошуку лікарських засобів, відправлення запитів до сервера і отримання відповідей без перезавантаження сторінки, що забезпечує швидку та плавну реакцію інтерфейсу.

*EJS* є веб-шаблонізатором, спеціально розробленим для роботи у веб-додатках на *Java*. Він ідеально підходить для інтеграції з *Spring* *Framework*, зокрема з *Spring MVC*, де він може слугувати як потужний двигун для генерації веб-відповідей на основі серверних даних і логіки.

- Шаблони як природні шаблони: *EJS* використовує "природні шаблони" підход. *HTML* шаблони створюються таким чином, що вони залишаються валідними *HTML*-файлами навіть після додавання специфічних атрибутів *EJS*, що дозволяє їм бути переглянутими в браузері як звичайні сторінки ще до їх обробки на сервері.

- Стандартна і зручна розмітка: *EJS* дозволяє використовувати стандартну *HTML* розмітку для створення шаблонів, додаючи свої атрибути для динамічних даних. Це означає, що розробники можуть легко працювати з шаблонами, знаючи лише *HTML* та базові концепції *EJS*.

- Використання фрагментів: *EJS* підтримує використання фрагментів шаблонів, що дозволяє визначити повторно використовувані компоненти інтерфейсу, такі як заголовки, навігаційні панелі, футери тощо, і включати їх в інші шаблони. Це сприяє уникненню дублювання коду і полегшує підтримку великих проектів.

- Прив'язка даних: *Java* об'єкти, які передаються з контролерів у шаблони, можуть бути використані в *EJS* через простий синтаксис прив'язки даних. Наприклад, якщо об'єкт `person` переданий у моделі, його властивості можуть бути відображені у шаблоні за допомогою атрибутів, таких як *`<%= person.name %>`*.

- Умовні висловлювання та цикли: EJS підтримує умовні висловлювання та цикли в шаблонах, що дозволяє розробникам створювати більш динамічні сторінки. Наприклад, можна використовувати *`<% if() {}%>`* для умовного відображення елементів або *`* *<% for () {} %>`* для ітерації по колекціях.

## 2.4 Проектування backend частини за допомогою Node.js, Express та MongoDB

У рамках проектування інтерактивної веб-платформи для реалізації backend компонентів обрано Node.js у поєднанні з Express та базою даних MongoDB. Цей набір технологій вибрано завдяки їх зрілості, продуктивності та підтримці великої кількості користувацьких операцій та даних, що є критично важливим для системи обліку матеріальних цінностей.

*Node.js:* Забезпечує високопродуктивну, асинхронну обробку запитів.

*Express:* Фреймворк для Node.js, який спрощує управління маршрутизацією та обробкою HTTP запитів.

*MongoDB:* *Потужна, масштабована база даних, яка підтримує документо-орієнтовану модель зберігання даних.*

Переваги *MongoDB*:

- Висока продуктивність та масштабованість.

- Гнучкість у роботі з даними.

- Підтримка складних запитів.

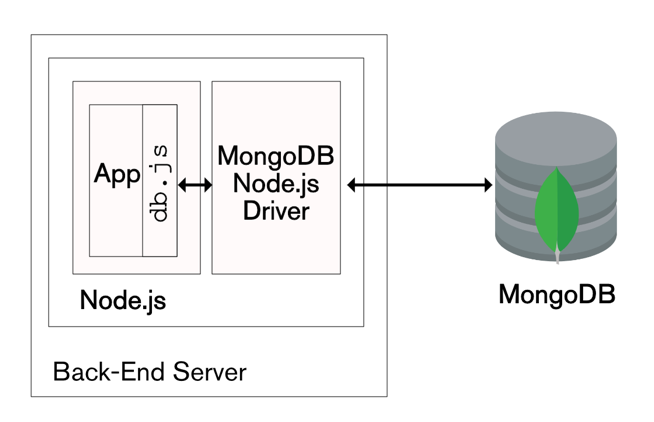


Рисунок 2.3 – Архітектура JavaScript та MongoDB

Ця інтеграція між *Java*Script, та *MongoDB* не лише забезпечує розробку з високим ступенем абстракції та контролем, але й гарантує стійкість, безпеку та високу продуктивність завдяки використанню передових практик програмування та баз даних.

Взаємодія між *Java*Script, та *MongoDB* є ключовим елементом у розробці стійких та ефективних *backend*-систем. Ця інтеграція використовується для створення масштабованих веб-додатків, які вимагають високої продуктивності обробки даних та роботи з базою даних. Нижче описано, як ці технології співпрацюють для реалізації потужних рішень.

## Висновок до розділу 2

У другому розділі було здійснено глибокий аналіз технологій, архітектурних підходів і взаємодії між компонентами для розробки інтерактивної веб-платформи, що забезпечує облік матеріальних цінностей у військових частинах. З огляду на особливості проекту та вимоги до функціональності і безпеки, було обрано стек технологій Node.js з використанням Express для серверної частини та MongoDB для бази даних. Це дозволяє забезпечити не тільки високу продуктивність та масштабованість, але й гнучкість та безпеку даних.

У рамках проектування було розроблено детальну архітектуру веб-додатку, включаючи управління користувачами, облік та адміністрування матеріальних цінностей. Особлива увага приділялася візуальному дизайну інтерфейсу, використовуючи HTML, CSS та JavaScript для створення інтуїтивно зрозумілого та легко навігованого користувацького інтерфейсу, що відіграє ключову роль у забезпеченні ефективної взаємодії з кінцевими користувачами під час критичних облікових операцій.

Комбінація цих технологій та методик дизайну не тільки сприяє зручності роботи з платформою, але й гарантує здатність системи адаптуватися до постійно змінних вимог обліку матеріальних цінностей. Завдяки гнучкості архітектури та можливостям швидкої модифікації функціоналу, платформа може легко

інтегруватися з іншими системами та додатками, що розширює можливості використання даних для аналітичних та стратегічних цілей. Це забезпечує потужний інструмент для підтримки фінансових служб військових частин під час облікових операцій.

# РОЗДІЛ 3 РЕАЛІЗАЦІЯ ІНТЕРАКТИВНОЇ WEB-ПЛАТФОРМИ

## 3.1 Реалізація серверної частини засобами *JavaScript*, *Node.js* та *MongoDB*

Початковою точкою для реалізації веб-платформи є налаштування серверної частини з використанням Node.js та Express.js. Код налаштовує Express-додаток, підключається до бази даних MongoDB та визначає необхідні налаштування, такі як шлях до шаблонів, обробники сесій та маршрутизацію.(рис.3.1).

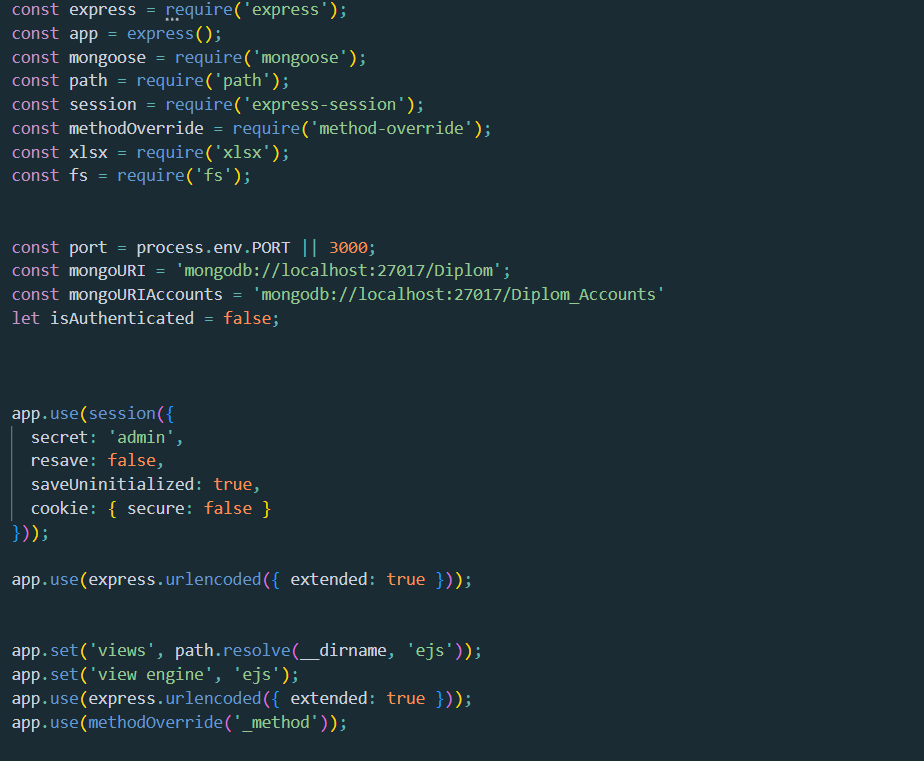


Рисунок 3.1 – Лістинг коду створення основного вебсервера з використанням *Express.js*:

Для забезпечення безпеки та аутентифікації користувачів, код визначає схему користувача Mongoose та налаштовує маршрути для входу та реєстрації. (рис.3.2)*.*

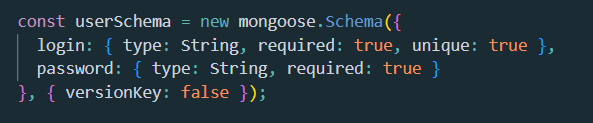


Рисунок 3.2 – Лістинг створення моделі користувача.

Після успішної аутентифікації, користувачі можуть переходити до головної сторінки та переглядати колекції документів у базі даних MongoDB. Код визначає маршрути для отримання списку колекцій, відображення документів у кожній колекції, а також додавання, редагування та видалення документів. (рис.3.3).

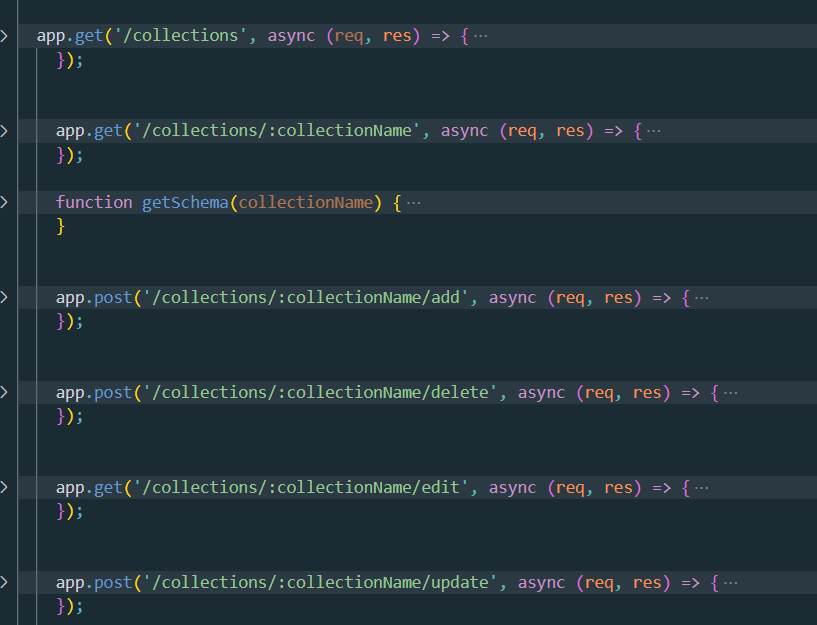


Рисунок 3.3 – Лістинг коду для відображення, редагування, додавання та видалення елементів.

Крім того, код містить функціональність для експорту документів з колекції у форматі XLSX за допомогою бібліотеки xlsx. (рис.3.4.)

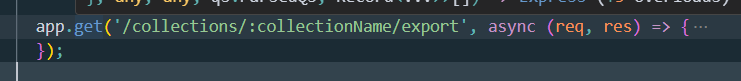


Рисунок 3.4 – Лістинг коду експорту докуменів з колекції.

## 3.2 Реалізація клієнтської частини з використанням *EJS* та *HTML*/*CSS*/*JS*

На клієнтській стороні використовується шаблонізатор EJS для динамічного відображення контенту та взаємодії з користувачем. Нижче наведено приклад шаблону EJS для відображення документів у колекції та обробки дій редагування та видалення*.* Основним завданням клієнтської частини є відображення інформації про матеріальні цінності та їх детальні характеристики. Реалізація цих завдань включає кілька ключових аспектів:

Створення структури сторінок: Використання *HTML* для розмітки забезпечує створення структурного вигляду веб-сторінок. *EJS* дозволяє вставляти дані з сервера безпосередньо в *HTML* шаблони, що сприяє динамічному відображенню контенту (рис.3.5.).

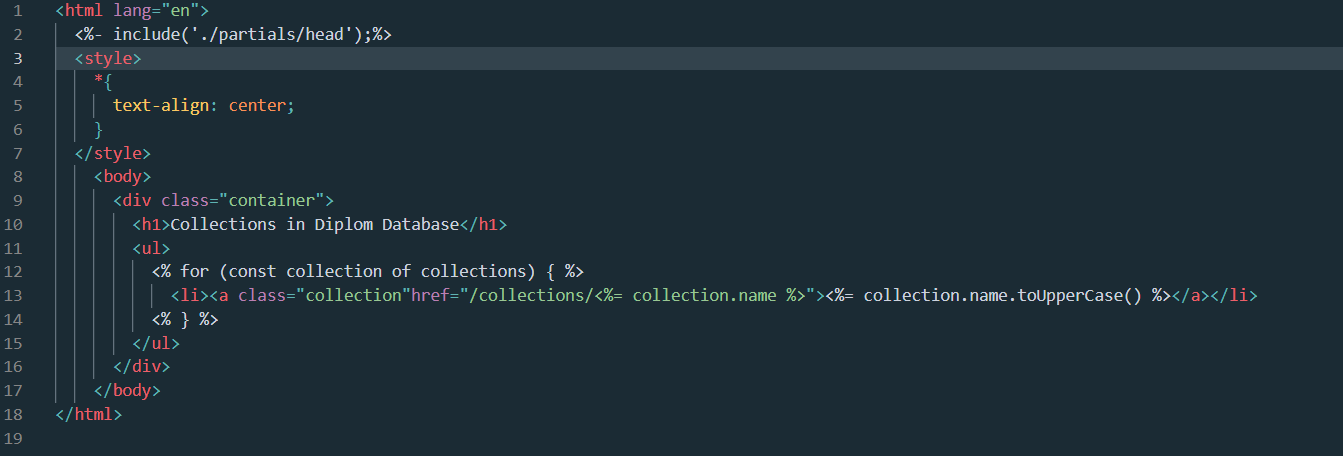


Рисунок 3.5 – Лістинг коду *HTML* шаблону, що сприяє динамічному відображенню контенту

Стилізація веб-інтерфейсу: *CSS* використовується для детального оформлення сторінок, включаючи колір, відступи, шрифти та інші візуальні елементи, які забезпечують привабливий та зрозумілий інтерфейс для користувачів.

Взаємодія користувача: *JavaScript* застосовується для додання інтерактивності на сторінках, таких як форми для введення даних, таби з інформацією, динамічні списки, анімації та інші елементи, які потребують взаємодії з користувачем (рис.3.6.).



Рисунок 3.7 – Лістинг коду для динамічного видалення документа.

Реалізація функціональності на стороні клієнта: *EJS* сприяє зв'язуванню даних на стороні сервера з *HTML*-елементами, що значно спрощує процес оновлення вмісту сторінки без повного її перезавантаження. Це підвищує продуктивність веб-додатка та покращує користувацький досвід.

В результаті, використання *EJS* у поєднанні з *HTML*, *CSS* та *JavaScript* дозволяє розробникам створювати високофункціональні та взаємодійні веб-інтерфейси, які ефективно інтегровані з серверною логікою, написаною на *JavaScript* за допомогою *Node.js*. Це забезпечує не тільки високу продуктивність веб-платформи, але й зручність розробки та підтримки, що є ключовими при реалізації складних систем, таких як інтерактивні веб-додатки для управління медичними препаратами.

## 3.3 Функціональні можливості системи інтерактивного тестування

Перед початком використання програмного продукту необхідно пройти процедуру авторизації (рис.3.8.) або реєстрації та вибрати з якої країни користувач родом і в якій зараз знаходиться (рис.3.9.).

Після успішного входу в систему, користувач потрапляє до меню, де має можливість використовувати функціонал сайту (рис.3.10.).

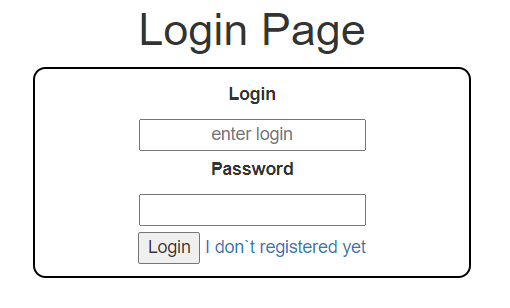


Рисунок 3.8 – Вікно авторизації

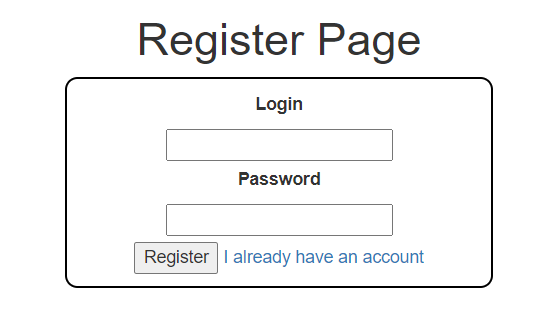


Рисунок 3.9 – Вікно реєстрації

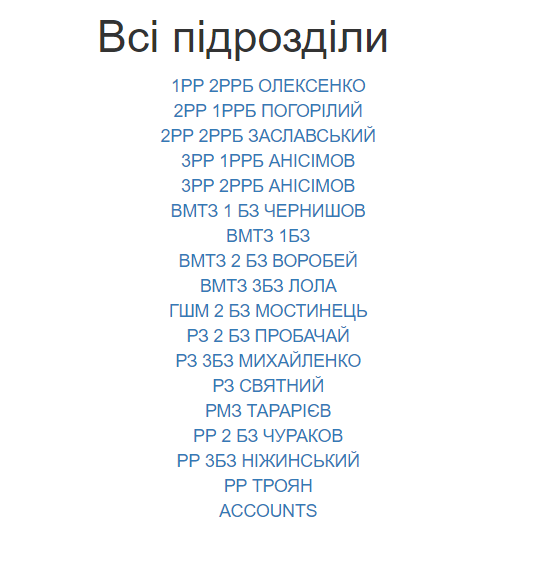


Рисунок 3.10 – Меню вибору підрозділу

Обравши з меню будь-який підрозділ, користувач потрапляє на сторінку цього підрозідлу (рис.3.11.), де може переглянути інформацію про майно підрозділу, додати документ, видалити документ, перейти на сторінку редагування, сформувати звіт для фінансової служби військової частини.



Рисунок 3.11 – Сторінка *майна підрозділу*.

Натиснувши на іконку Олівця, користувач потрапляє на сторінку редагування документу, де може змінити будь-яку інформацію(рис.3.12.).

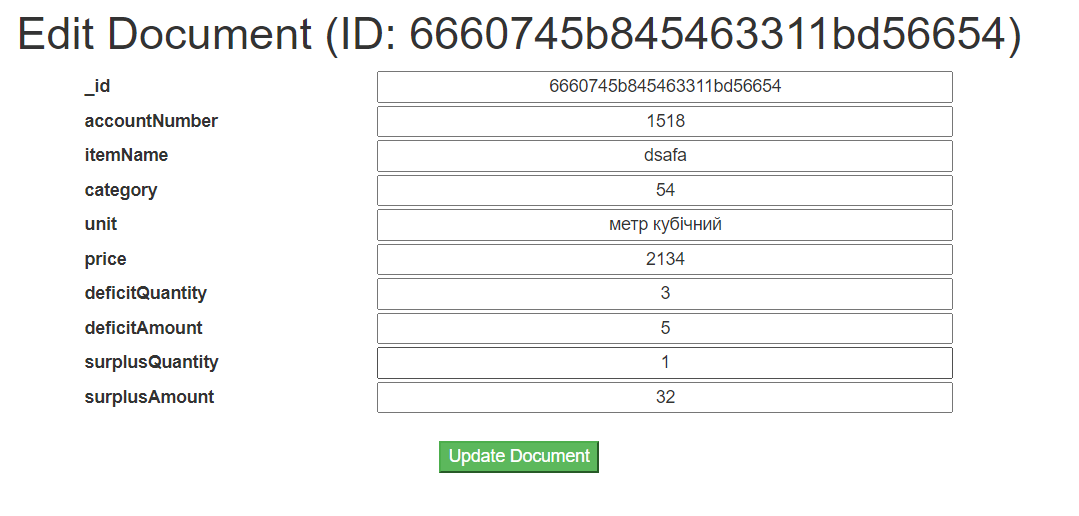


Рисунок 3.12 – Сторінка  *Редагування документа.*

Використовуючи форму на сторінці підрозділу користувач може додати елемент до бази(рис.3.13.).

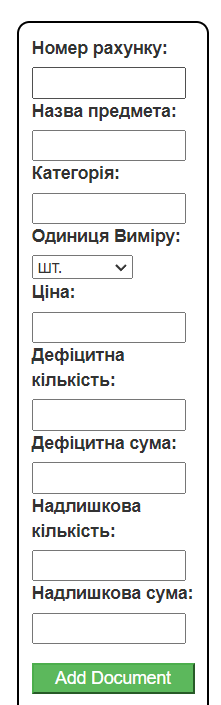


Рисунок 3.13 – *Форма додавання документів.*

Натиснувши на іконку Хрестика, користувач видаляє елемент(рис.3.14.).



Рисунок 3.17 – Видалення елементу

## Висновок до розділу 3

Завершення третього розділу підсумовує ефективну реалізацію серверної та клієнтської частини інтерактивної веб-платформи для автоматизації обліку матеріальних цінностей та генерації звітів для фінансової служби військової частини. Використання *JavaScript*, *Node.js* та *MongoDB* для серверної частини дозволило створити надійну, безпечну та легко масштабовану систему, яка забезпечує ефективне управління даними і відповідає сучасним вимогам до розробки програмного забезпечення.

Клієнтська частина, реалізована з використанням *EJS* та стандартних технологій *HTML*, *CSS*, і *JavaScript*, забезпечила гнучке та інтуїтивно зрозуміле взаємодія з користувачем. Це дозволило втілити динамічний та зручний інтерфейс, що сприяє залученню користувачів та їхньому зручному взаємодії з платформою.

Загалом, реалізація інтерактивної веб-платформи в третьому розділі підтвердила виправданість вибору використаних технологій та підходів, закладаючи міцний фундамент для подальшого розвитку і оптимізації системи.

# ВИСНОВКИ

Ця дипломна робота була спрямована на розробку інтерактивної веб-платформи для управління даними про колекції документів, використовуючи сучасні технології Node.js, Express, MongoDB, а також клієнтську частину на основі EJS, HTML, CSS та JavaScript. Проект вирішив низку задач, спрямованих на поліпшення доступу до інформації та зручність обробки даних, забезпечивши користувачам ефективний інструмент для управління колекціями документів.

Серверна частина, реалізована за допомогою Node.js та Express, забезпечила надійну архітектуру для обробки даних і взаємодії з базою даних MongoDB, що дозволило створити масштабовану і безпечну систему. Використання сесійного управління забезпечило ефективне управління доступом та захист даних користувачів.

Клієнтська частина, розроблена з використанням EJS та стандартних веб-технологій, запропонувала користувачам інтерактивний і зрозумілий інтерфейс. Це значно спростило процеси пошуку та обробки документів, забезпечивши користувачам детальну інформацію про кожну колекцію. Використання EJS сприяло тісній інтеграції з серверною логікою, що дозволило ефективно відображати дані в реальному часі.

Основною перевагою розробленої системи є її гнучкість та можливість адаптації до різних вимог користувачів, забезпечуючи можливість легкого оновлення та розширення функціоналу. Реалізація функціоналу експорту даних у форматі Excel надала додаткову цінність, дозволивши користувачам легко зберігати та аналізувати дані за допомогою знайомих інструментів.

Таким чином, проект демонструє успішне використання сучасних технологічних рішень для вирішення актуальних проблем у сфері управління документами, підтверджуючи значення технологій у сприянні доступності та ефективності інформаційних послуг.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. *Ethan Brown. Learning Node.js Development: Design and build scalable and maintainable web applications using Node.js. Packt Publishing, 2019.*
2. *Brad Traversy. Node.js, Express, MongoDB & More: The Complete Bootcamp 2021. Udemy, 2021.*
3. *Azat Mardan. Practical Node.js: Building Real-World Scalable Web Apps. Apress, 2018.*
4. *Evan Hahn. Express in Action: Writing, building, and testing Node.js applications. Manning Publications, 2016.*
5. *Valeri Karpov. Professional Node.js: Building JavaScript-Based Scalable Software. Wrox, 2013.*
6. *Simon Holmes and Clive Harber. Getting MEAN with Mongo, Express, Angular, and Node. Manning Publications, 2019.*
7. *Steve Holzner. MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly Media, 2019.*
8. *Kyle Simpson. You Don't Know JS Yet: Scope & Closures. O'Reilly Media, 2020.*
9. *Marijn Haverbeke. Eloquent JavaScript: A Modern Introduction to Programming. No Starch Press, 2018.*
10. *Addy Osmani. Learning JavaScript Design Patterns. O'Reilly Media, 2012.*
11. *Eric Elliott. Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Moderns JS Libraries. O'Reilly Media, 2014.*
12. *David Flanagan. JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language. O'Reilly Media, 2020.*
13. *Samer Buna. Node.js Design Patterns: Design and implement production-grade Node.js applications using proven patterns and techniques. Packt Publishing, 2019.*
14. *Miguel Grinberg. Flask Web Development: Developing Web Applications with Python. O'Reilly Media, 2018. (в якості додаткового джерела для розширення перспективи щодо веб-розробки).*
15. *John Sonmez. Soft Skills: The software developer's life manual. Manning Publications, 2015. (для підвищення навичок продуктивності та ефективності розробника).*

# ДОДАТОК А

Лістинг основних методів backend частини

const express = require('express');

const app = express();

const mongoose = require('mongoose');

const path = require('path');

const session = require('express-session');

const methodOverride = require('method-override');

const xlsx = require('xlsx');

const fs = require('fs');

// Define port and MongoDB connection details

const port = process.env.PORT || 3000;

const mongoURI = 'mongodb://localhost:27017/Diplom';

const mongoURIAccounts = 'mongodb://localhost:27017/Diplom\_Accounts'

let isAuthenticated = false;

// Session configuration

app.use(session({

secret: 'your\_secret\_key', // Replace with a strong secret key

resave: false,

saveUninitialized: true,

cookie: { secure: false } // Set to true if using https

}));

app.use(express.urlencoded({ extended: true }));

// Set templates folder path and view engine (replace if using a different engine)

app.set('views', path.resolve(\_\_dirname, 'ejs'));

app.set('view engine', 'ejs');

app.use(express.urlencoded({ extended: true }));

app.use(methodOverride('\_method'));

// Define mongoose schema for user accounts

const userSchema = new mongoose.Schema({

login: { type: String, required: true, unique: true },

password: { type: String, required: true }

}, { versionKey: false });

// \*\*\*\*\* Login and Registration Routes \*\*\*\*\*

app.post('/login', async (req, res) => {

try {

const connection = await mongoose.connect(mongoURIAccounts);

const db = connection.connection.db;

const collections = await db.listCollections().toArray();

collections.sort((a, b) => a.name.localeCompare(b.name));

const collectionName = collections[0].name;

console.log('MongoDB connected');

// Validate collectionName

if (!collectionName) {

return res.status(400).json({ message: 'Missing collection name' });

}

const { login, password } = req.body;

// Create Mongoose model dynamically (if needed)

const User = mongoose.model(collectionName, userSchema);

// Check if user exists with the provided login

const user = await User.findOne({ login });

if (!user) {

return res.status(401).json({ message: 'Invalid username' }); // Specific error message

}

// Check if password is valid

if (password!==await user.password) {

return res.status(401).json({ message: 'Invalid password' }); // Specific error message

}

// Successful login logic (e.g., create session, redirect)

isAuthenticated = true;

req.session.isLoggedIn = true;

req.session.user = user; // Store user data in session (optional)

res.redirect('/collections');

connection.disconnect(); // Redirect to main page after successful login

} catch (error) {

console.error('Error logging in user:', error);

res.status(500).send('Error logging in user');

}

});

// Registration route (POST /register)

app.post('/register', async (req, res) => {

try {

const connection = await mongoose.connect(mongoURIAccounts);

const db = connection.connection.db;

const collections = await db.listCollections().toArray();

collections.sort((a, b) => a.name.localeCompare(b.name));

const collectionName = collections[0].name;

const User = mongoose.model(collectionName,userSchema)

const { login, password } = req.body;

const newUser = new User({ login, password });

const savedUser = await newUser.save();

if (savedUser) {

console.log('User registered successfully');

// Redirect to the login page after successful registration

return res.redirect('/login');

} else {

console.error('Error registering user');

res.status(500).send('Error registering user');

}

} catch (error) {

if (error.name === 'MongoServerError' && error.code === 11000) {

console.error('Error registering user:', error.message);

return res.status(400).render('register', {title:'User with same login already exist, try another one'});

}

else{

console.error('Error registering user:', error);

res.status(500).send('Error registering user');

}

}

});

// Main page route (protected - requires login)

app.get('/', (req, res) => {

res.render('index', { title: 'Main Page' }); // Replace with your main page content

});

// Login page route (public)

app.get('/login', (req, res) => {

res.render('login', { title: 'Login Page' }); // Replace with your login form

});

// Registration page route (public)

app.get('/register', (req, res) => {

res.render('register', { title: 'Register Page' }); // Replace with your registration form

});

// Collections Page

app.get('/collections', async (req, res) => {

try {

if(isAuthenticated){

const connection = await mongoose.connect(mongoURI);

    const db = connection.connection.db;

    const collections = await db.listCollections().toArray();

    collections.sort((a, b) => a.name.localeCompare(b.name));

    res.render('collections', { title: 'Collection Page', collections });

    await connection.disconnect();

}else{

res.render('login',{title:"Login Page"})

}

  } catch (error) {

    console.error('Error:', error);

    res.status(500).send('Error listing collections');

  }

});

// Page for each collection

app.get('/collections/:collectionName', async (req, res) => {

  try {

const connection = await mongoose.connect(mongoURI);

    const db = connection.connection.db;

    const collectionName = req.params.collectionName;

const collection = db.collection(collectionName); // Get the collection object

    const documents = await collection.find().toArray(); // Sort ascending by \_id

if(isAuthenticated){

    res.render('unit', { title: collectionName, documents });

}else{

res.render('login',{title:"Login Page"})

}

    await connection.disconnect();

  } catch (error) {

    console.error('Error:', error);

    res.status(500).send('Error listing collections');

  }

});

function getSchema(collectionName) {

  const schema = new mongoose.Schema({

    accountNumber: { type: String, required: true },

    itemName: { type: String, required: true },

    category: { type: Number, required: true },

    unit: { type: String, required: true },

    price: { type: Number },

    deficitQuantity: { type: Number, default: 0 },

    deficitAmount: { type: Number, default: 0 },

    surplusQuantity: { type: Number, default: 0 },

    surplusAmount: { type: Number, default: 0 }

  }, { versionKey: false });

  return schema;

}

// In your route handler

app.post('/collections/:collectionName/add', async (req, res) => {

  try {

    const connection = await mongoose.connect(mongoURI);

    const db = connection.connection.db;

    const collectionName = req.params.collectionName;

    // Get the schema and model for the collection

    const Item = mongoose.models[collectionName] || mongoose.model(collectionName, getSchema(collectionName));

    const { accountNumber,

      itemName,

      category,

      unit,

      price,

      deficitQuantity,

      deficitAmount,

      surplusQuantity,

      surplusAmount

    } = req.body;

    const document = new Item({

      accountNumber,

      itemName,

      category,

      unit,

      price,

      deficitQuantity,

      deficitAmount,

      surplusQuantity,

      surplusAmount

    });

    await document.save();

    res.redirect(`/collections/${collectionName}`);

    console.log(`successful added in ${collectionName}`);

    await connection.disconnect();

  } catch (error) {

    console.error('Error adding document:', error);

    res.status(500).send('Error adding document');

  }

});

app.post('/collections/:collectionName/delete', async (req, res) => {

try {

const connection = await mongoose.connect(mongoURI);

const db = connection.connection.db;

const collectionName = req.params.collectionName;

const documentId = req.body.id; // Access the submitted ID from req.body

const collection = db.collection(collectionName);

const result = await collection.deleteOne({ \_id: new mongoose.Types.ObjectId(documentId) });

if (result.deletedCount === 1) {

res.redirect(`/collections/${collectionName}`); // Redirect to collection list

} else {

res.status(404).json({ message: 'Document not found' });

}

await connection.disconnect();

} catch (error) {

console.error('Error deleting document:', error);

res.status(500).json({ message: 'Error deleting document' });

}

});

app.get('/collections/:collectionName/edit', async (req, res) => {

try {

const connection = await mongoose.connect(mongoURI);

const db = connection.connection.db;

const collectionName = req.params.collectionName;

const documentId = req.query.id; // Access document ID from query string

// Clear cached model before creating a new one (if needed)

delete mongoose.models[collectionName];

const collection = db.collection(collectionName);

const document = await collection.findOne({ \_id: new mongoose.Types.ObjectId(documentId) });

if (!document) {

return res.status(404).json({ message: 'Document not found' });

}

res.render('edit', { title: `Edit Document (ID: ${documentId})`, collectionName, document });

await connection.disconnect();

} catch (error) {

console.error('Error fetching document:', error);

res.status(500).send('Error fetching document');

}

});

app.post('/collections/:collectionName/update', async (req, res) => {

  try {

    const connection = await mongoose.connect(mongoURI);

    const db = connection.connection.db;

    const collectionName = req.params.collectionName;

    const documentId = req.body.id;

    const collection = db.collection(collectionName);

    const schema = getSchema(collectionName);

    const Model = mongoose.model(collectionName, schema);

    const updateData = req.body;

    delete updateData.id;

    const result = await Model.findByIdAndUpdate(documentId, updateData, { new: true });

    if (!result) {

      return res.status(404).json({ message: 'Document not found' });

    }

    res.redirect(`/collections/${collectionName}`);

    console.log(`Document (ID: ${documentId}) updated successfully`);

    await connection.disconnect();

  } catch (error) {

    console.error('Error updating document:', error);

    res.status(500).send('Error updating document');

  }

});

app.get('/collections/:collectionName/export', async (req, res) => {

try {

// Connect to MongoDB

const connection = await mongoose.connect(mongoURI);

const db = connection.connection.db;

const collectionName = req.params.collectionName;

// Get documents from the collection

const collection = db.collection(collectionName);

const documents = await collection.find().toArray();

// Load the provided workbook

const filePath = (path.resolve(\_\_dirname,'звіт.xls'))

const workbook = xlsx.readFile(filePath);

// Get the first worksheet

const worksheetName = workbook.SheetNames[0];

const worksheet = workbook.Sheets[worksheetName];

// Define the starting row for the table

let rowIndex = 17; // Assuming the table starts at row 2

// Map document properties to columns

const propertyToColumn = {

accountNumber: 'A',

category: 'B',

itemName: 'C',

unit: 'D',

price:'E',

deficitQuantity: 'F',

deficitAmount: 'G',

surplusQuantity: 'H',

surplusAmount: 'I'

};

// Write data to the table

for (const document of documents) {

let colIndex = 0;

for (const [property, column] of Object.entries(propertyToColumn)) {

const cellAddress = `${column}${rowIndex}`;

const value = document[property] !== undefined ? document[property] : 'empty';

worksheet[cellAddress] = { v: value, t: typeof value === 'number' ? 'n' : 's' };

colIndex++;

}

rowIndex++;

}

// Update the range of the worksheet

worksheet['!ref'] = xlsx.utils.encode\_range({

s: { r: 1, c: 0 },

e: { r: rowIndex - 1, c: Object.keys(propertyToColumn).length - 1 }

});

// Write the modified workbook to a new file

const outputFilePath = path.join(\_\_dirname, 'updated\_inventory.xlsx');

xlsx.writeFile(workbook, outputFilePath);

// Send the updated file as response

res.download(outputFilePath, 'updated\_inventory.xlsx', (err) => {

if (err) {

console.error('Error sending file:', err);

res.status(500).send('Error sending file');

}

fs.unlinkSync(outputFilePath); // Clean up the file after sending it

});

await connection.disconnect();

} catch (error) {

console.error('Error exporting documents:', error);

res.status(500).send('Error exporting documents');

}

});

// Start the server and listen for connections

app.listen(port, () => {

console.log(`Server listening on port ${port}`);

});  
  
<!DOCTYPE html>

<html lang="en">

<%- include('./partials/head');%>

<%- include('./partials/styleUnit');%>

<body>

<div>

<a id="to\_collections" href="/collections">Back to Collections Page</a>

<h1>Documents in Collection: <%= title.toUpperCase() %></h1>

<div class="collections">

<div class="table">

<table>

<thead>

<tr>

<% for (const prop in documents[0]) { %>

<th><%= prop %></th>

<% } %>

<th>Actions</th>

</tr>

</thead>

<tbody>

<% for (const document of documents) { %>

<tr>

<% for (const prop in document) { %>

<td><%= document[prop] %></td>

<% } %>

<td>

<span class="edit-trigger" data-document-id="<%= document.\_id %>"></span>

<span class="delete-trigger" data-document-id="<%= document.\_id %>"></span>

</td>

</tr>

<% } %>

</tbody>

</table>

</div>

<div class="buttons">

<form action="/collections/<%= title %>/add" method="POST">

<%- include('./partials/input');%>

<button class="btn-success">Add Document</button>

</form>

<form action="/collections/<%= title %>/export" method="GET">

<button>Save file in xlsx</button>

</form>

</div>

</div>

</div>

<script>

const deleteTriggers = document.querySelectorAll('.delete-trigger');

const editTriggers = document.querySelectorAll('.edit-trigger');

deleteTriggers.forEach(trigger => {

trigger.addEventListener('click', (event) => {

const documentId = event.target.dataset.documentId;

const collectionName = '<%= title %>';

// Create a form element dynamically

const form = document.createElement('form');

form.method = 'POST';

form.action = `/collections/${collectionName}/delete`;

// Add a hidden input field with document ID

const hiddenInput = document.createElement('input');

hiddenInput.type = 'hidden';

hiddenInput.name = 'id';

hiddenInput.value = documentId;

form.appendChild(hiddenInput);

// Submit the form programmatically

document.body.appendChild(form);

form.submit();

document.body.removeChild(form); // Remove the temporary form

});

});

editTriggers.forEach(trigger => {

trigger.addEventListener('click', (event) => {

const documentId = event.target.dataset.documentId;

const collectionName = '<%= title %>';

window.location.href = `/collections/${collectionName}/edit?id=${documentId}`;

});

});

</script>

</body>

</html>