МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Ігор ШЕЛЕХОВ
_____
(підпис)
червня 2023 р.

КОМПЛЕКСНА КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавр

зі спеціальності 122 - Комп'ютерних наук,
освітньо-професійної програми «Інформатика»
на тему: «Інформаційне і програмне забезпечення системи керування контентом інтернет-магазину. Серверна частина»
здобувача групи ІН – 94-1 Бончуков Іван Геннадійович

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Іван БОНЧУКОВ
(підпис)

Керівник,
Кандидат наук, доцент Ігор ШЕЛЕХОВ _____
(підпис)

Суми – 2023

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
Ігор ШЕЛЕХОВ
_____
(підпис)

ЗАВДАННЯ НА КОМПЛЕКСНУ КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавра

зі спеціальності 122 - Комп'ютерних наук, освітньо-професійної програми «Інформатика»
здобувача групи ІН-94-1 Бончукова Івана Геннадійовича

1. Тема роботи: «Інформаційне і програмне забезпечення системи керування контентом інтернет-магазину. Серверна частина.

затверджую наказом по СумДУ від *«01» червня 2023 р. № 0475-VI*

2. Термін здачі здобувачем кваліфікаційної роботи *до 09 червня 2023 року*

3. Вхідні дані до кваліфікаційної роботи _____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Огляд технологій, які використовуються для керування контентом інтернет-магазинів 3) Розробка інформаційної системи керування контентом інтернет магазину. 4) Аналіз результатів.*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ (підпис) Керівник _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
2	*Огляд технологій, що використовуються для керування контентом інтернет-магазинів*		
3	*Розробка інформаційної системи керування контентом інтернет магазину*		
4	*Аналіз отриманих результатів*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ (підпис) Керівник _____ (підпис)

АНОТАЦІЯ

Записка: 57 стр., 8 рис., 16 додаток, 6 використаних джерел.

Обґрунтування актуальності теми роботи – Тема кваліфікаційної роботи є актуальною, оскільки присвячена розв’язанню важливої практичної задачі адміністрування та управління інтернет-магазином.

Об’єкт дослідження — процес управління контентом інтернет магазину.

Мета роботи — розробка інформаційної системи ауправління контентом інтернет магазину.

Методи дослідження — алгоритми аналізу та обробки даних.

Результати — розроблено інформаційну систему, яка зберігає та оброблює дані про покупців, їх закази, формує на основі цього статистику.

ІНФОРМАЦІЙНЕ І ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ КЕРУВАННЯ КОНТЕНТОМ ІНТЕРНЕТ-МАГАЗИНУ. СЕРВЕРНА ЧАСТИНА.

ЗМІСТ

ВСТУП

В сьогоденних реаліях, коли світ інформаційних технологій стрімко розвивається, кожного дня з’являються нові технології, а наше суспільство стрімкими темпами цифровізується, дуже важливим аспектом нашого життя є сфера інтернету.

Це важливо, оскільки в цілому наш світ тримається на наступних речах: комунікація, комерція та інновації. Без інтернету людям було б набагато важче комунікувати, обмінюватися інформацією, тощо. Зверніть увагу наскільки залежить темп розвитку технологій від того, наскільки у людей в різних куточках світу є можливість обмінюватися думками та ідеями. Це можна зрозуміти просто переглянувши дані по темпам розвитку промисловості кілька стоень років тому, і зараз.

Без простих та доступних способів комунікації було також складно розвивати світову економіку, раніше у людей не було навіть думки, що за допомогою декількох простих дій можливо буде придбати собі товар з іншого куточка нашої неосяжної Землі. Сьогодні така можливість є, і все це за допомогою потужного та стрімкого розвитку інформаційних технологій, звичайно, не можливо уявити таке без розвитку й інших сфер нашого життя: промисловості, транспортних засобів, тощо. Але це знову ж таки без потужних засобів комунікування між людьми цього б всього не було, або було але в набагато менших масштабах.

Станом на зараз існує безліч різноманітних систем під буквально будь-які людські потреби, ми можемо дистанційно здавати сессію, купувати товари з будь-якої точки світу, знаходити потрібну інформацію для навчання, роботи, знаходити людей схожих за інтересами і це все не виходячи з дому.

Люди сьогодні не можуть уявити своє буття без смартфонів, комп’ютерів, тощо оскільки все це є основним засобом комунікування, що робе наше життя

набагато більш простішим. Звичайно навіть станом на сьогодні не весь світ цифровізований, але людство швидкими темпами йде саме до цього.

З всього вищесказаного зрозуміло, що для того, щоб все це і надалі розвивалось та підтримувалось потрібні спеціалісти які будуть підтримувати всю цю надскладну систему.

Наразі також існує надвеликий попит на всілякі інтернет-магазини, системи адміністрування товарів, оскільки в наслідок розвиненої системи комунікування у продавців виникає великий попит на їх товари, потрібно адмініструвати товари, закази, оптимально зберігати контакти своїх покупців, розширювати клієнтську базу, тощо.

В результаті проведеної бакалаврської роботи буде розроблено повністю працездатну систему, що може використовуватися, для полегшення життя людям, що займаються продажами. Оскільки при великому товарообігу, потрібно оптимально розпоряджатися своїм часом та часом своїх клієнтів, для того, щоб твій бізнес був ще більш продуктивним.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Загальна інформація про серверне програмне забезпечення

Розвиток веб-додатків в тому вигляді якому ми його знаємо почався в 1990 році, з тих часів на момент написання даної роботи прошло вже 33 роки. Багато чого змінилося з тих часів.

Веб того часу був представлений в більшості зі звичайних HTML-документів за допомогою яких люди у яких був доступ до комп’ютерів могли обмінюватися різноманітною інформацією між собою.

HTML – це мова гіпертекстової розмітки, що розроблено під роботу з веб-браузерами. В основному використовується для створення простих сторінок, які можна переглядати через браузер, крім цього він надає можливість посилатися на інші сторінки, ресурси, тощо.

Працювало це все на основі HTTP-протоколу, що являє собою прикладний рівень моделі OSI. HTTP – це протокол, що дозволяє отримувати певні дані, як наприклад, вже було вище сказано HTML-сторінки, цей протокол є так би мовити, базою, дідом сьогоднішніх систем обміну інформацією, на сьогодні всі поступово від нього відмовляються у зв’язу з появою нових протоколів, таких як HTTPS та SSH, оскільки цей протокол має певні недоліки пов’язані в першу чергу з безпекою. HTTP це в першу чергу, про клієнт-серверну взаємодію, що означає, те що запроси до сервера на отримання певних даних робиться самим адресатом, зазвичай в його ролі виступає браузер, але завжди є виключення. В основі цього діалогу лежить два основних поняття: запит(request) та відповідь(request). Запит це повідомлення відправлене клієнтом, в основному використовують наступні:

- GET – запит на отримання даних з сервера
- POST – запит на відправленя даних на сервер

Підсумуючи можно сказати що Інтернет на базовому рівні являє собою систему обміну документами, які зв'язані між собою гіперпосиланнями.

Але світ не стояв на місці, тому окрім простої передачі файлів, потрібно

було якось її упорядковувати, приводити до більш-менш пристойного вигляду.

Для цього було розроблено CSS, каскадну таблицю стилів, що являє собою засіб декорування HTML-сторінок. Це означає, що тепер окрім звичайного тексту чорного тексту на чорному фоні, сторінки можуть представляти собою більш пристойну за виглядом річ.

Звичайно, що маючи такий потужний засіб обміну інформацією, люди почали активно його розвивати. Окрім розміщення інформації її також потрібно десь зберігати, сортувати, тощо. Тут ми підходимо до поступового розділення вебу на дві основні частини: frontend та backend, перший потрібен для представлення користувачам даних, другий для їх обробки та збереження.

Дані в свою чергу поділяються на: динамічні та статичні. В чому різниця? Розглянемо наступний приклад: коли ми заходимо на сторінку де окрім нічого крім тексту немає, це статичні дані, динамічні можна спостерігати сьогодні набагато частіше, зайдіть в будь-яку соціальну мережу та напишіть коментар. Цей коментар пройде довгий путь де він буде відправлений на сервер у вигляді POST-запиту, де буде перевірений на валідність, та доданий до бази даних.

Ще одним фактором того, що веб-розробка поділилася на дві відокремленні частини стало те, що обсяг HTML-сторінок які потрібно було розміщувати на сайтах становився все більшим, тому почали приходити до того, що набагато оптимальніше (якщо ми говоримо про сайти з великою кількістю контенту) буде використовувати Singlepage-додатки та асинхроні запити. Тепер замість кількох сотень HTML-сторінок, використовується одна, яка динамічно змінюється в залежності від того, що відправе сервер на клієнт. Це удобніше, оскільки тепер можна відправляти невеликий за обсягом JSON-об’єкт.

1.2 Архітектура серверного програмного забезпечення

До сьогоднішнього дня архітектура серверного програмного забезпечення невплинно ускладнювалася. Це пов’язано з тим, що потрібно працювати зі все більшими обсягами данних та проводити багато різноманітних операцій над ними.

Основою будь-якого серверного додатку є в першу чергу ядро, в якому описується вся бізнес-логіка. Бізнес-логіка це в першу чергу певний комплекс процесів, правил, алгоритмів, операцій і таке інше що використовується в межах певної парадігми вирішення задач.

Окрім ядра, також активно використовуються бази даних. База даних – це певна вибірка даних, що внутрішньо пов’язана між собою. Окрім збереження даних вона призначена також для їх корегування, обробки і т.д. Якщо казати більш конкретно, то база даних(якщо ми говоримо про реляційні) являє собою набір таблиць в яких данні строго структуровані. Зазвичай бази даних використовують для обробки великої кількості інформації, на сьогодні не один інтернет-магазин, корпоративний сайт, соціальну мережу неможливо уявити без використання баз даних.

Окрім звичайних, реляційних баз даних, також існують, так звані, документоорієнтовані – нереляційні БД. Вони використовуються, коли нам потрібно зберігати хаотичні, малозв’язані між собою дані.

Для керування базами даних використовують СУБД. СУБД (Система Управління Базою Даних) – це програмне забезпечення, що виконує роль посередника між базою даних та тим, хто нею користується. На сьогодні існує багато різних СУБД, із основних можно виділити: MySQL, Postgresql, MoongoDB, MariaDB та Oracle. СУБД обирається для кожного проєкта індивідуально, оскільки кожна з них має як свої переваги, так і недоліки. Тому правильний вибір СУБД займає не останнє місце в проєктуванні серверного програмного забезпечення.

Окрім зберігання великих обсягів інформації, також може виникнути потреба в зберіганні і навпаки, малого обсягу даних, до яких потрібен швидкий доступ. Для цього використовують кєш. Кєш – це структура, що оперує даними на рівні ключ-значення, де кожен окремий ключ є унікальним. Така структура дозволяє надавати доступ до даних майже моментально. Зазвичай кєш

використовують коли потрібно зберігати, наприклад сесії користувачів на сайті, JWT-токени та інші дані до яких потрібно часто звертатися.

Коли ядро додатку стає занадто великі, виникає проблема підтримки такого програмного забезпечення, оскільки коли одна частина програми залежить від іншої, при збої в роботі однієї частини може зламатися в принципі вся програма. Тому, у великих серверних додатках використовують мікросервісну архітектуру. Така архітектура дозволяє відокремити кожну частину програми на окремі сервіси, а це в свою чергу збільшує можливості пдітримки та розширення веб-додатків. Наприклад, один сервіс відповідає за авторизацію, інший за роботу з БД, третій за новосну стрічку і так далі.

1.3 Архітектура MVC

При створенні веб додатків, потрібно чітко відрізняти логіку однієї частини програми від іншої, оскільки це сприяє більш простому масштабуванню, та підтримці додатку. Тому було розроблено архітектурний патерн MVC, що дозволяє розробникам програмного забезпечення створювати прості але в той же час потужні додатки завдяки чітко виставленій архітектурі.

MVC (Model, View, Controller) – це архітектурний патерн, що являє собою шаблон розділення даних(інформації), інтерфейсу та бізнес логіки. MVC оперує наступними термінами: модель, вид, контролер. Кожний з цих елементів має свою зону відповідальності. Першу популярність даний шаблон набрав коли був інтегрований в Struts та Ruby on Rails.

Основною ідеєю цього патерну є чітке розділення відповідальності за різні функціональні частини веб-додатків. Тепер розглянемо окремо кожний елемент.

Модель – це певний набір правил та даних, що представляють собою певну парадигму управління додатком. Будь-який сучасний веб додаток, це структура що складається з даних які оброблюються певним чином. Модель надає контролеру інформацію, що певним чином структурована та уніфікована, Модель також можна назвати ядром цього партерну оскільки вона містить в собі всю логіку по обробці даних нашого додатку. Модель єдиний елемент який може

напряму контактувати з базами даних. Розробники виходячи з того, які саме задачі перед ними ставляться створюють модель, що буде ці самі задачі максимально ефективно виконувати.

Контролер – це елемент який виконує роль менеджера. Головною задачею контролера є розподіл задач які потрібно виконувати та подальша передача їх на потрібну модель. Тобто, контролер є посередником між користувачем та моделлю, що дозволяє чітко розмежувати зони відповідальності. В основному контролер це максимально тонкий прошарок, що оперує запитами користувача та перенаправляє певний запит на певну модель.

Вид – це елемент партерну, що використовується для подання користувачу потрібних даних, що були отримані із моделі.

Зазвичай, веб-додатки розроблюють таким чином, що під кожний запит у нас є свій контролер, модель та вид, що будуть оброблювати цей самий запит. Наведемо приклад: у нас є користувач якому потрібно отримати інформацію про наявні товари на сайті. По-першу він робе GET-запит на сервер, по певній адресі, наприклад: “http://example.com/products” , далі якщо така адреса оброблюється контролером він передає управління моделі, яка в свою чергу робить запит до бази даних та витягує з неї потрібну інформацію, далі управління знову передається контролеру, який відправляє отримані дані на вид. Готово! Користувач бачить перед собою список наявних товарів.

1.4 Постановка задачі

Мета цієї роботи – це створення серверного додатку, для керування контентом інтернет магазину.

Зміст програми полягає в створенні системи, що буде, зручно адмініструвати товари, закази та користувачів сайту, робити звітність для аналізу успішності інтернет магазину.

Основні вимоги для додатку:

а) Можливість авторизації.

б) Створення товарів.

в) Створення категорій

г) Створення атрибутів товарів.

д) Статистика та звітність.

е) Розподілення ролей для користувачів

ж) Можливість завантаження зображень для товарів

з) Створення заказів користувачів

и) Стабільність роботи додатку.

к) Документація

2 ВИБІР ПРОГРАМНИХ ЗАСОБІВ

2.1 Вибір мови програмування

Мова програмування грає далеко не останню роль, коли ми говоримо про створення серверного програмного забезпечення. В першу чергу це пов’язано з тим, що мова програмування це по-перше інструмент, інструмент що призначений для вирішення певних задач.

Також важливими критеріями вибору є зручність для розробника, швидкість розробки та постійні оновлення, оскільки світ стрімко розвивається і потрібно відповідати сьогоденним стандартам.

На сьогоднішній день можно виділити наступні мови програмування з уклоном в серверну розробку:

1. PHP
2. Java
3. Python
4. C#
5. JS&Node.js

Ці мови є найбільш популярними серед розробників по версії сайту TIOBE станом на траверь 2023 року, це можна переглянути на рис. 2.1. Даний сайт спеціалізується на відстеженні тенденцій в сфері розробки, що досягається шляхом відстеження пошукових запросів в найбільш популярних пошукових системах: Google, YouTube, Yahoo!, Bing. Це дозволяє об’єктивно оцінювати тенденції серед веб-розробників.

May 2023	May 2022	Change	Programming Language	Ratings	Change
1	1		Python	13.45%	+0.71%
2	2		C	13.35%	+1.76%
3	3		Java	12.22%	+1.22%
4	4		C++	11.96%	+3.13%
5	5		C#	7.43%	+1.04%
6	6		Visual Basic	3.84%	-2.02%
7	7		JavaScript	2.44%	+0.32%
8	10	^	PHP	1.59%	+0.07%
9	9		SQL	1.48%	-0.39%
10	8	v	Assembly language	1.20%	-0.72%

Рисунок 2.1 – Рейтинг популярності мов програмування по версії TIOBE

Кожна з вищеописаних мов програмування має як свої недоліки так і переваги.

PHP – мова програмування яка була розроблена саме для серверної веб-розробки. На даний момент в світі написано майже 1\3 всіх сайтів, що досягається в першу чергу досягається швидкістю його освоєння, тому багато розробників-новачків починають свій шлях саме з нього. Самий популярний ресурс розроблений на PHP це Wordpress, що є самою затребуваною CMS на даний момент. На сьогодні практика створення чогось нового на даній мові програмування не актуально оскільки через великий попит та велику кількість часу на ринку майже все, що було можливо вже написано. Окрім, вищесказаного, у PHP також є свої недоліки як наприклад неоднорідність синтаксису.

Java – це мова універсальна мова програмування, яка має велику базу різноманітних додатків і т.д. На Java пишуть як великі enterprise-додатки як наприклад LinkedIn або пошукова система Yahoo! так і різномантні open-source проєкти. Основною перевагою даної мови програмування є в першу чергу кросплатформність, завдяки Java Runtime Machine. Також Java добре інтегрована під роботу з паттерном об’єктно орієнтованого програмування імає для цього всі необхідні інструменти. Окрім всього вищесказаного Java має велику кількість вирішених проблем, готових бібліотек та фреймворків під будь-які задачі тощо. З недоліків можна виділити те, що з точки зору сучасності синтаксис мови є

застарілим та багатослівним, що в сьогоднішніх реаліях, коли важливо швидко створювати програмні продукти є великим недоліком, окрім цього багато проєктів написано на старих версіях Java, тому підтримка таких продуктів є проблематичною.

Python – це високорівнева інтерпретована мова програмування, яка працює поверх C++. Першочергово Python розрахований на максимально швидку розробку програмного забезпечення, зазвичай його використовують разом з фреймворком Django, що вважається найбільш оптимальним вибором для надшвидкої розробки веб-додатків. Але основним недоліком Python є орієнтованість на обробку великої кількості даних, тому не завжди Python є оптимальним варіантом.

C# - дана мова починає свій шлях з мови C. Дана мова програмування також є максимально універсальною та підходить для будь-яких задач, зокрема: розробка ігр, ПЗ, бізнес-додатки. Головною перевагою мови, є в першу чергу обширна документація, що підтримується компанією Microsoft. Мова постійно розвивається та підтримується, з кожним роком стає кращою. Але все ж таки не дивлячись на все, основний упор в ній робиться на роботу з ПЗ, тому для веб-програмування C# підходить не завжди.

JavaScript – це мова програмування яка від самого початку презентувалася як мова для фронтенд розробки яка використовувалася в основному для створення простих анімацій, скриптів які виконував веб-браузер, але з годом вона переросла в більш потужний інструмент, на якому сьогодні пишуть як клієнтські так і серверні веб-додатки. З виходом стандарту ECMAS 6 JavaScript став більш схожим на інші мови програмування, як наприклад Java або C#, оскільки прибавилось багато корисних елементів як наприклад генератори, класи, механізми наслідування, асинхроні операції та інше. Для серверної веб-розробки на JavaScript використовують платформу Node.js, що надає можливість кросплотформеної веб-розробки.

Для створення програмного продукту було обрано мову JavaScript у зв’язці з TypeScript та Node.js, оскільки вона має приємний синтаксис з не самим високим рівнем входу, пакетний менеджер npm, велику кількість готових модулів під будь-яку ситуацію, а з TypeScript стає ще більш потужною мовою серверного програмування.

2.2 Вибір фреймворку

На даний момент існує багато різноманітних фреймворків для Node.js, оскільки це платформа яка активно розвивається та тільки набирає популярності. Вибір фреймворку є важливою частиною в програмування програмного забезпечення, оскільки за умови вибору правильного фреймворку розробнику буде набагато простіше створювати якісне та в майбутньому легко підтримуване програмне забезпечення.

Основними фрейморками для платформи Node.js являються:

1. Express
2. Koa.js
3. Nest.js

Розглянемо кожний з них детальніше:

Express – це база, основний та самий популярний фреймворк для створення веб-додатків на платформі Node.js на даний момент. Хоча даний фрейморк є доволі мінімалістичним, він в той же час залишається достатньо гнучким та підходе для створення майже будь-яких серйозних веб-додатків. Express має велику кількість служебних методів HTTP, middleware (проміжних обробників), окрім цього Express це по суті своїй тонкий прошарок між розробником та ядром Node.js. Це означає, що розробник має дуже гнучкий інструментарій та не залежить від внутрішній правил фреймфорку, а це в свою чергу може бути як і недоліком так і перевагою.

Koa.js – це самий малий з точки зору ваги фреймворк, що являє собою ще більш тонкий прошарок абстракції. Koa має достатньо велику спільноту, легко розширюється, а також дуже добре інтегрується з MongoDB. Також однією з

особливостей даного фреймворку є використання Pug – це мова яка дозволяє швидко створювати HTML-шаблони та може підтримувати динамічний код.

Але не зважаючи на це все через надзвичайно малу кількість інструментів цей фреймворк не завжди є оптимальним рішенням.

Для створення програмного забезпечення було обрано фреймворк Nest.js, чому саме він? По-перше це обгортка над Express, що покликана для того, щоб поставити розробника в чіткі рамки MVC патерну проєктування. По-друге, Nest.js має більш обширний функціонал, якого немає в Express, і якщо б ми обрали Express його б прийшлось би дописувати самим. Nest.js додає декларативність, тому при розробці в команді ви завжди будете чітко розуміти структуру вашого проєкту.

Окрім цього Nest.js має потужну утиліту для генерування файлів та проєктів, це @nestjs/cli. За допомогою декількох команд ми можемо створити новий проєкт, вибрати для нього потрібний нам пакетний менеджер та інтегрувати в проєкт тайпскрипт, окрім цього @nestjs/cli дозволяє створювати окремі файли контролерів, сервісів та інше, з повним переліком можна ознайомитися на рис 2.2

name	alias	description
application	application	Generate a new application workspace
class	cl	Generate a new class
configuration	config	Generate a CLI configuration file
controller	co	Generate a controller declaration
decorator	d	Generate a custom decorator
filter	f	Generate a filter declaration
gateway	ga	Generate a gateway declaration
guard	gu	Generate a guard declaration
interceptor	itc	Generate an interceptor declaration
interface	itf	Generate an interface
library	lib	Generate a new library within a monorepo
middleware	mi	Generate a middleware declaration
module	mo	Generate a module declaration
pipe	pi	Generate a pipe declaration
provider	pr	Generate a provider declaration
resolver	r	Generate a GraphQL resolver declaration
resource	res	Generate a new CRUD resource
service	s	Generate a service declaration
sub-app	app	Generate a new application within a monorepo

Рисунок 2.2 – Перелік можливостей nest generate

Також, перевагою Nest.js можно вважати шикарну інтегрованість з TypeScript. TypeScript це обгортка над мовою JavaScript, що була розроблена компанією Microsoft, і робе з JS, мову програмування зі строгою типізацією. А це важливо оскільки в серйозних веб-додатках важливо чітко відстежувати типи даних з якими ви працюєте, бо не знаючи з якими даними ви працюєте ви наражаєте себе на велику кількість помилок та неочікуваних ситуацій.

Також, не можу не звернути уваги на вибір файлового менеджера, оскільки він також грає роль в процесі розробки. Для роботи з Node.js існують наступні файлові менеджери: npm, yarn та pnpm. Для даного проєкту було обрано npm, оскільки він має велику швидкість створення файлів, підтримку великої кількості пакетів.

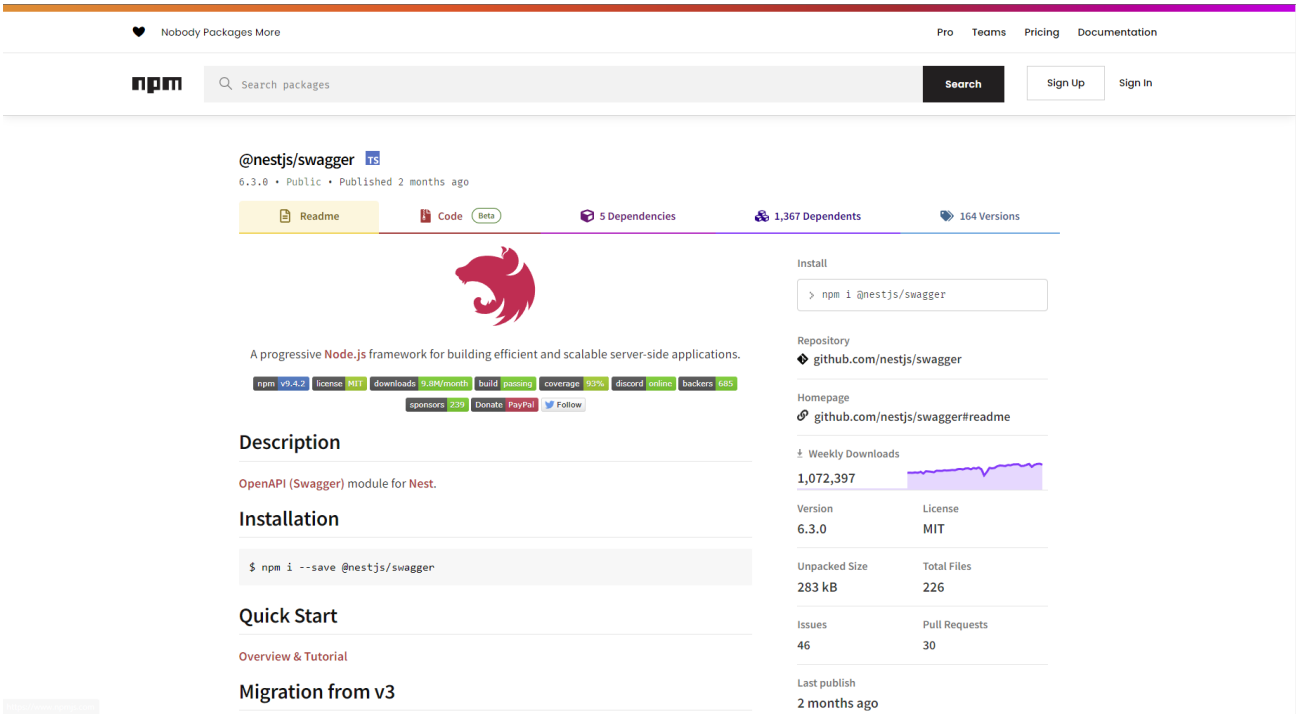

Рисунок 2.3 – Сторінка файлового менеджеру NPM

Отже, для створення серверної частини веб-додатку було обрано саме Nest.js за чітку структуру проєкту, зручну утиліту для генерування файлів та налаштування проєкту та за інтеграцію з TypeScript.

2.3 Вибір СУБД та ORM

СУБД грає важливу роль при проєктуванні програмного забезпечення, на сьогодні існує чимало різних СУБД під будь-які потреби розробників.
Окрім СУБД потрібно спершу визначитися з типом самої бази: реляційна або нереляційна (NoSQL) для даного проєкту більш оптимальним вибором буде реляційна база даних, оскільки дані які потрібно зберігати повинні бути чітко структуровані, та можуть мати складну структуру відносин між собою.

СУБД – це спеціальний набір програмного забезпечення, що слугує інтерфейсом між розробником та базою даних, який дозволяє виконувати різноманітні операції над базою даних за допомогою мови SQL. Зазвичай у кожної СУБД свій діалект SQL, але в своїй суті відрізняються вони тільки у дрібницях.

Основними СУБД для реляційних баз даних є на сьогодні є:

1. MySql
2. Oracle DB
3. Postgresql

Розглянемо кожну СУБД більш детально:

MySql – одна з найбільш популярних реляційних баз даних, яка працює на всіх сучасних платформах від Microsoft Windows до Ubuntu. Підтримує інтеграцію з багатьма мовами програмування, має високу оптимізацію що позитивно впливає на швидкість роботи та відсутності проблем з пам’яттю. Але основним недоліком є рідкі оновлення та проблеми з резервним копіюванням, та юнікодом.

Oracle DB – в першу чергу орієнтована на великий бізнес, та обробку величезної кількості інформації. Основним недоліком є в першу чергу велика вартість даного програмного продукту, та значні вимоги до апаратної частини серверного забезпечення, оскільки вона орієнтована в першу чергу на роботу з великою кількістю даних.

Для дипломного проєкту було обрано Postgresql, оскільки вона є безплатною, але не зважаючи на це вона може працювати з будь-яким об’ємом даних, має достатньо приємну розширюваність та підтримку складних структур даних таких як масиви бітові строки і т.д.

В якості ORM було обрано TypeORM, оскільки вона має простий інтерфейс, та в той же час широкі можливості для створення складних структур сутностей та має широку типізацію.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Розробка моделі системи

Створений API для системи керування контентом інтернет магазину являється сукупністю класів та моделей, які відповідають за певним функціям додатку. як наведено у додатку. Проєкт створений на JavaScript.

Для розуміння концепції було створено ERD-діаграму бази даних представлену на рис. 3.1, для того, щоб розуміти що саме потрібно для даного додатку.

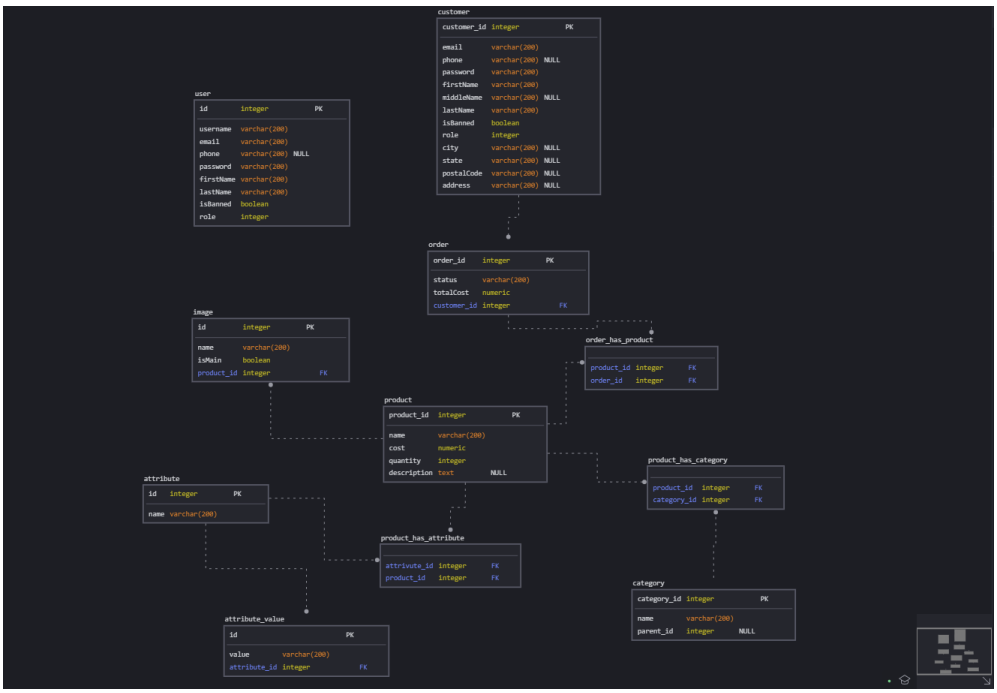

Рисунок 3.1 - ERD-діаграма бази даних додатку

Після розробки діаграми, було утворено опис ендпоінтів, що наведено на таб. 3.1, що будуть услугувати інтерфейсом по роботі з базою даних.

Таблиця 3.1 – Опис ендпоінтів додатку

Метод	Адреса	Функція
GET	/api/entity	Повернення всіх даних
GET	/api/entity/:id	Повернення конкретного запису
POST	/api/entity	Створення даних
PATCH	/api/entity /:id	Оновлення даних
DELETE	/api/entity /:id	Видалення даних

Окрім простої роботи з даними додаток також повинен мати систему авторизації, авторизація має окремий список ендпоінтів наведений в таб. 3.2. За основу авторизації та аутентифікації було взято стандарт JWT(Json Web Token), він використовується для шифрування та передачі даних авторизованих користувачів між сервером та клієнтом. Тому для коректного функціонування було додано метод для оновлення токена.

Таблиця 3.2 – Опис ендпоінтів авторизації

Метод	Адреса	Функція
POST	/api/auth/signup	Реєстрація
POST	/api/auth/signin	Аутентифікація
POST	/api/auth/refresh	Оновлення токену
POST	/api/auth/reset-request	Запрос на відновлення паролю
GET	/api/auth/redirect/:link	Переадресація на сторінку відновлення паролю
GET	/api/auth/get-my-info	Перегляд інформації про авторизованого користувача

Після опису ендпоінтів було створено новий проєкт та визначено файлову структуру яку наведено на рис. 3.2

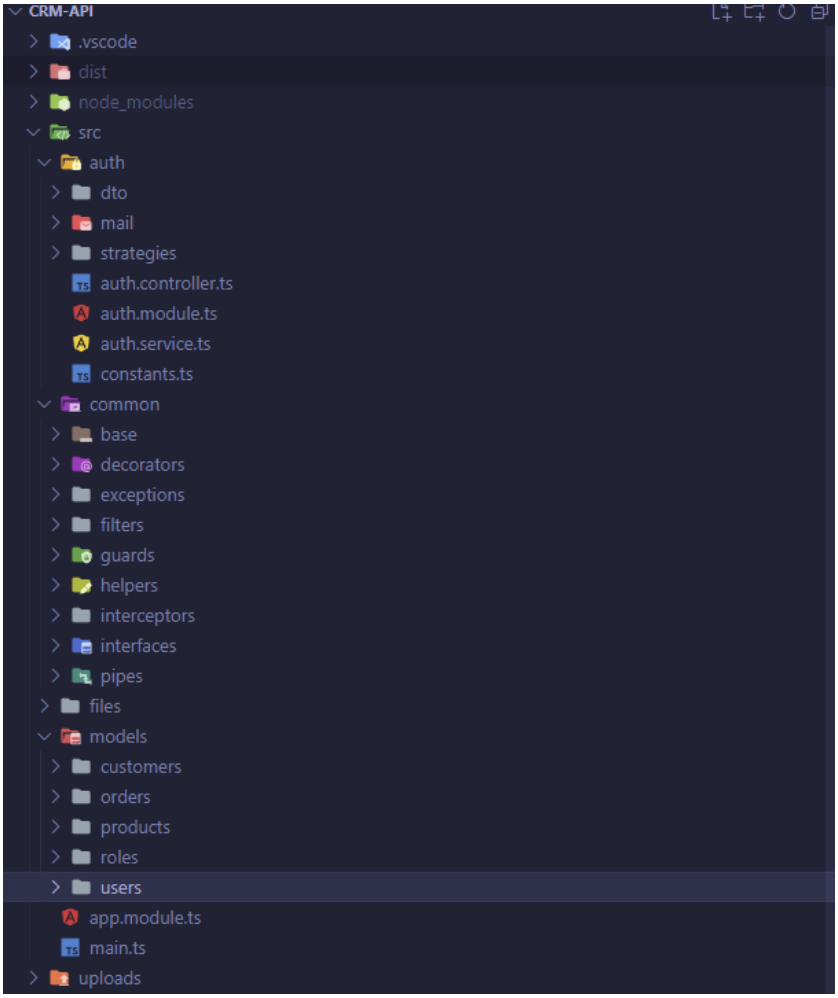

Рисунок 3.2 - Структура проєкту

3.2 Опис основних класів та методів додатку

Паттерн MVC вимагає чіткого розділення функціоналу додатку, тому основна роль по функціоналу випадає на сервіси.

Першочергово розглянемо сервіс авторизації для авторизації було використано додаткові модулі пакетного менеджера NPM, а саме:

@nestjs/jwt – який надає функціонал по створенню JWT-токенів. Налаштування даного модуля проводилось в файлі auth.module.ts

```
imports: [
 JwtModule.register({
   global: true,
   secret: jwtConstants.secret,
 }),
```

Passport/jwt - який потрібен для зручної перевірки валідності токену.

Для коректності роботи модуля створено створити клас JwtStrategy, що потрібний для налаштування первірки токену.

```
const cookieExtractor = (req: Request) => {
    let token = null;
    if (req && req.cookies) {
        token = req.cookies?.jwt;
    }
    return token;
};

export class JwtStrategy extends PassportStrategy(Strategy) {
    constructor() {
        super({
            jwtFromRequest: cookieExtractor,
            ignoreExpiration: false,
            secretOrKey: jwtConstants.secret
        })
    }

    async validate(payload: any) {
        return {userId: payload.sub, username: payload.username}
    }
}
```

Також було створено *JwtAuthGuard.* Guard – це механізм, що дозволяє створити систему авторизації для обмеження доступу до певних ендпоінтівза певними критеріями, в цьому випадку це наявність у користувача валідного JWT-токена .

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
    const isPublic =
this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
```

```
            context.getHandler(),
            context.getClass(),
          ]);
          if (isPublic) {
            return true;
          }
          return super.canActivate(context);
        }
      }
      export const IS_PUBLIC_KEY = 'isPublic';
      export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

Bcrypt – для шифрування паролю, оскільки зберігання паролю в базі даних в відкритому вигляді є не найкращою практикою з точки зору безпеки.

Клас *AuthService* містить в собі наступні методи:

Реєстрація.

```
      async signUp(email: string, username: string, password: string): Promise<User | null> {
              const existingEmail = await this.usersRepository.findOneBy({
                  email: email,
              });
              const existingUserName = await this.usersRepository.findOneBy({
                  username: username,
              });
              if(existingEmail || existingUserName) {
                  throw new BadRequestException('email or login already exist', { cause: new Error(), description: 'signUp error' });
              }
              const passwordHash = bcrypt.hashSync(password, 10);
              const doc = {
                  email: email,
                  username: username,
                  password: passwordHash,
                  role: 2, // User role
              };
```

```
            const user = this.usersRepository.create(doc);
            await this.usersRepository.save(user);
            delete user.password;
            return user;
    }
```

Аутентифікація та авторизація. Саме тут використовується модуль *Bcrypt*. Він шифрує пароль який надіслав користувач і повертає рядок з результатом назад.

```
     async signIn(username: string, password: string, res: Response):
Promise<Object | null> {
            const user = await (await this.usersRepository.findOne({ where: {
username: username }, relations: ['role'] }));
            if (!user) {
                throw new NotFoundException('user not found', { cause: new
Error(), description: 'signIn error' });
            }

            const valid = await bcrypt.compare(password, user.password);
            if (!valid) {
                throw new UnauthorizedException('incorrect login or password',
{ cause: new Error(), description: 'signIn error' });
            }
            return this.getTokens(user, res);
}
```

Оскільки повертати токен потрібно відразу в декількох місцях коду було прийняте рішення створити для цього окрему функцію.

```
         private async getTokens(user: User, res: Response): Promise<Object |
null> {
            try {
                const payload = {
                    id: user.id,
                    username: user.username,
                    role: user.role,
                }
                const access_token = await this.jwtService.signAsync(payload,
                    {
                        expiresIn: jwtConstants.expiresInAccess,
                    });
                const refresh_token = await this.jwtService.signAsync(payload,
                    {
                        expiresIn: jwtConstants.expiresInRefresh,
                    })
                res.cookie('jwt', access_token, { httpOnly: true, expires: new
Date(new Date().getTime() + 15 * 60000) });
                res.cookie('refresh_token', refresh_token, { httpOnly: true,
expires: new Date(new Date().getTime() + 60 * 60000) })
                return {
```

```
                    access_token,
                    refresh_token,
                    expires_in: jwtConstants.expiresInAccess
                };

            } catch (err) {
                throw new BadRequestException();
            }
        }
```

Оновлення токену.

```
    async refreshToken(refreshToken: string, res: Response) {
        const result = await this.jwtService.verifyAsync(refreshToken);
        if(!result) {
            throw new UnauthorizedException('invalid refresh-token');
        }
        const user = await this.userService.getUser(result.id);
        return this.getTokens(user, res);
    }
```

Запит на оновлення паролю.

```
    async resetRequest(dto: ResetPasswordDto, res: Response) {
            const existingEmail = await this.usersRepository.findOneBy({
                email: dto.email,
            });
            const existingUserName = await
this.usersRepository.findOneBy({
                username: dto.username,
            });
            if(!existingEmail || !existingUserName) {
                throw new BadRequestException('email or login not found');
            }

            const resetLink = mailConstants.reset_url + uuidv4();
            await this.mailService.sendResetMail(dto.email, resetLink);

            const payload = {
                email: dto.email,
                username: dto.username,
                resetLink: resetLink,
            }
            const reset_token = await this.jwtService.signAsync(payload,
                {
                    expiresIn: jwtConstants.expiresInReset,
            });
            res.cookie('reset_token', reset_token, { httpOnly: true,
expires: new Date(new Date().getTime() + 60 * 60000) });
            return [];
        }
```

Для відправки листа підтвердження на пошту було додано MailService, який використовує модуль nodemailer:

```
@Injectable()
export class MailService {
```

```
        constructor(private readonly mailerService: MailerService) {}

        async sendResetMail(email: string, token: string): Promise<void> {
            const to = email;
            const subject = 'Reset Password'

            await this.mailerService.sendMail({
                to,
                subject,
                text: ` Click for activation from ${process.env.API_URL}:
${token}`,
            });
        }
    }
```

Переадресація користувача на сторінку зміни паролю:

```
        async redirect(res: Response) {
            res.redirect(`${process.env.FRONT_RESET_URL}/login?recover=true`);
            return [];
        }
```

Для того, щоб у користувача був обмежений доступ та час на зміну пароля було створено *ResetTokenGuard.*

```
    @Injectable()
    export class ResetTokenGuard implements CanActivate {
      constructor(private jwtService: JwtService) {}

      canActivate(context: ExecutionContext): boolean {
        try {
        const req: Request = context.switchToHttp().getRequest();
        const resetToken = req.cookies?.reset_token;

        if (!resetToken) {
          throw new HttpException('Reset token is missing',
HttpStatus.UNAUTHORIZED);
        }
        const payload = this.jwtService.verify(resetToken);

        req.user = payload;

        return true;
        } catch (e) {
          console.log(e)
        }

      }
    }
```

Зміна паролю користувача.

```
      async changePassword(dto: ChangePasswordDto, req: Req, res: Response) {
            const resetToken = req.cookies?.reset_token;
            if (!resetToken) {
                throw new NotFoundException('reset token not found')
            }
```

```
        const payload = this.jwtService.decode(resetToken);

        const user = await this.usersRepository.findOneBy({
            email: payload['email']
        });
        if(!user) {
            throw new BadRequestException('user not found');
        }

        const passwordHash = bcrypt.hashSync(dto.password, 10);
        user.password = passwordHash;

        await this.usersRepository.save(user);
        res.clearCookie('reset_token');
        if (user) {
            res.json({ success: true });
        }
    }
```

Отримання даних авторизованого користувача.

```
    async getMyInfo(req: Req) {
        const resetToken = req.cookies?.jwt;
        const payload = this.jwtService.decode(resetToken);

        if (!payload['username']) {
            throw new BadRequestException('Incorrect token');
        }

        const user = await this.usersRepository.findOneBy({
            username: payload['username']
        });

        if(!user) {
            throw new BadRequestException('user not found');
        }
        delete user.password;
        return user;
    }
```

Для працездатності вищеописаного коду було створено сутність користувача.

```
@Entity('users')
export class User extends IBaseEntity{

  @ApiProperty({
    description: 'user login',
    example: 'username',
    type: String
  })
  @Column({ type: 'varchar', unique: true, length: 36, })
  username: string;

  @ApiProperty({
    description: 'user email',
    example: 'example@mail.com',
    type: String
  })
  @Column({ type: 'varchar', unique: true, })
  email: string;
```

```
  @ApiProperty({
    description: 'user phone number',
    example: '+380950277946',
    type: String,
    default: null
  })
  @Column({ type: 'varchar', default: null, unique: true, })
  phone: string;

  @ApiProperty({
    description: 'user password',
    example: 'efsdfER!@123m',
    type: String,
  })
  @Column({ type: 'varchar', length: 150, })
  password: string;

  @ApiProperty({
    description: 'user first name',
    example: 'Ivan',
    type: String,
    default: '',
  })
  @Column({ type: 'varchar', default: '' })
  firstName: string;

  @ApiProperty({
    description: 'user last name',
    example: 'Ivanov',
    type: String,
    default: '',
  })
  @Column({ type: 'varchar', default: '' })
  lastName: string;

  @ApiProperty({
    description: 'property indicating whether the user is banned or not',
    example: 'example@mail.com',
    type: Boolean,
    default: false
  })
  @Column({ type: 'boolean', default: false, })
  isBanned: boolean;

  @ApiProperty({
    description: 'user role, default 2 - user role',
    example: 'example@mail.com',
    type: Number,
    default: 2
  })
  @ManyToOne(() => Role, role => role.id)
  role: number;
}
```

Для більшої коректності було створено функцію яка перевіряє чи співпадають введені паролі користувача при реєстрації нового акаунта.

```
@ValidatorConstraint({name: 'Match'})
export class MatchConstraint implements ValidatorConstraintInterface {
```

```
    validate(value: any, args: ValidationArguments) {
        const [relatedPropertyName] = args.constraints;
        const relatedValue = (args.object as any)[relatedPropertyName];
        return value === relatedValue;
    }
}
```

Також налаштовано TypeOrm

```
TypeOrmModule.forRoot({
      type: 'postgres',
      host: process.env.POSTGRES_HOST,
      port: Number(process.env.PORT),
      username: process.env.POSTGRES_USER,
      password: process.env.POSTGRESS_PASSWORD,
      database: process.env.POSTGRES_DB,
      synchronize: true,
      autoLoadEntities: true,
    }),
```

Оскільки подальша розробка полягає в тому щоб створювати однотипні сервіси та контролери, для більш оптимальної розробки та подальшої відладки та розширення додатку було створено базові клас та сервіс.

По-перше створено інтерфейс корньового сервісу, для більшої декларативності.

Клас *BaseService* працює з абстрактним типом T що успадковується від *IBaseEntity*, який в свою чергу створено для більш швидкого опису сутностей, щоб кожного разу не прописувати однотипні поля які потрібні для буь-якої сутності.

```
      @Entity()
      export abstract class IBaseEntity {
          @PrimaryGeneratedColumn({ type: 'bigint'})
          id: number;

          @CreateDateColumn({ name: 'create_at' })
          createdAt: Date;

          @UpdateDateColumn({ name: 'updated_at' })
          updatedAt:Date;
      }
```

В конструкторі класу *BaseService* є можливість вказання додаткових параметрів, оскільки ми можемо працювати зі складними сутностями, у яких може бути наприклад декілька відношень з іншими. *genericRepository* це обов’язковий параметр який приймає репозиторій сутності з якою нам треба буде

працювати у дочірньому сервісі. *relationOptions* визначається масивом сутностей, що будуть використовуватися в усіх CRUD-операціях. *relatesRepositories* приймає об’єкт формату ключ-значення, ключом є рядок що відповідає назві додаткової сутності, а значенням є клас переданої сутності, це потрібно для оновлення та додавання інформації до зв’язаних сутностей. На випадок якщо при створенні нового сервісу не було задано додаткові опції, в конструкторі все виставляється на null.

```
constructor (
              private readonly genericRepository: Repository<T>,
              private readonly relationOptions?: FindManyOptions<T>,
              private readonly relatedRepositories?: {[key: string]:
Repository<any>},
              searchField?: keyof T,
          ) {
              this.options = relationOptions || {};
              this.relatedRepositories = relatedRepositories || null;
              this.searchField = searchField || 'id';
            }
```

Метод для перевірки сутностей що надходять до інших методів класу.

```
         protected async checkRelated(entity: T) {
              const relationArray = Object.keys(this.relatedRepositories);
              for (const relation of relationArray) {
                  if (entity.hasOwnProperty(relation)) {
                      const relatedEntity = entity[relation];
                      const relatedRepository =
this.relatedRepositories[relation];

                      if (Array.isArray(entity[relation])) {
                          //if relation have several items (relation - Array)
                          for (const item of entity[relation]) {
                              if (!item.id) {
                                  throw new BadRequestException(`Invalid data in
${relation} id not specified`);
                              }

                              const findOptions = {
                                  where: {
                                      id: item.id as
FindOptionsWhereProperty<NonNullable<T['id']>, any>
                                  },
                              } as FindOneOptions<T>;
                              const foundEntity =
relatedRepository.findOne(findOptions);
                              if (!foundEntity) {
                                  throw new BadRequestException(`${relation} for
${relatedEntity.id} id not found`);
                              }
                          }

                      } else {
                          if (!relatedEntity.id) {
```

```
                                throw new BadRequestException(`Invalid data in
${relation} id not specified`);
                            }
                            const findOptions = {
                                where: {
                                    id: relatedEntity.id as
FindOptionsWhereProperty<NonNullable<T['id']>, any>
                                },
                            } as FindOneOptions<T>;
                            const foundEntity =
relatedRepository.findOne(findOptions);
                            if (!foundEntity) {
                                throw new BadRequestException(`${relation} for
${relatedEntity.id} id not found`);
                            }
                        }
                    }
                }
            }
```

Метод для створення нової сутності.

```
   async create(entity: T): Promise<T> {
                try {
                    const findOptions: FindOneOptions<T> = {
                        where: {
                            [this.searchField]: entity[this.searchField],
                        } as FindOptionsWhere<T>,
                    };
                    const record = await
this.genericRepository.findOne(findOptions);

                    if(record) {
                        throw new BadRequestException('record is already exist');
                    }

                    if (this.relatedRepositories) {
                        this.checkRelated(entity);
                    }

                    const createRecord = await
this.genericRepository.create(entity);
                    const saveRecord = await
this.genericRepository.save(createRecord);

                    return saveRecord;
                } catch (e) {
                    console.log(e)
                }

            }
```

Методи для повернення записів з бази даних на клієнт.

```
async getAll(): Promise<T[]> {
    const records =  await this.genericRepository.find(this.options);
    if(records.length === 0) {
        throw new NotFoundException('table is empty');
    }
    return records;
}

async getOneById(id: number): Promise<T> {
    const findOptions = {
        where: {
          id: id as FindOptionsWhereProperty<NonNullable<T['id']>, any>
        },
        ...this.options
    } as FindOneOptions<T>;
    const record = await this.genericRepository.findOne(findOptions);
    if (!record) {
        throw new NotFoundException('record not found')
    }

    return record;
}
```

Методи видалення та оновлення записів в базі даних.

```
async update(id: number, entity: T): Promise<T> {
    const record = await this.getOneById(id);
    Object.assign(record, entity);
    return await this.genericRepository.save(record);
}

async delete(id: number): Promise<T> {
    const record = await this.getOneById(id);
    await this.genericRepository.delete(id);
    return record;
}
```

Для того, щоб у користувача був якийсь інтерфейс для взаємодіїї з сервісами було створено базовий контроллер *BaseController*

```
export class BaseController<T extends IBaseEntity> {
    constructor(private readonly IBaseService: IBaseCrudService<T>) {}
```

```
	@Get()
	@ApiResponse({ status: 200, description: 'Ok'})
	async findAll(): Promise<T[]> {
	  return this.IBaseService.getAll();
	}

	@Get(':id')
	@ApiResponse({ status: 200, description: 'Record retrieved successfully.'})
	@ApiResponse({ status: 404, description: 'Record does not exist'})
	async findById(@Param('id', new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE })) id: number): Promise<T> {
	  return this.IBaseService.getOneById(id);
	}

	@Post()
	@ApiResponse({ status: 201, description: 'The record has been successfully created.'})
	@ApiResponse({ status: 403, description: 'Forbidden.'})
	@ApiResponse({ status: 400, description: 'Bad Request.'})
	async create(@Body() entity: T): Promise<T> {
	      return this.IBaseService.create(entity);
	}

	@Patch(':id')
	@ApiResponse({ status: 400, description: 'Bad Request.'})
	@ApiResponse({ status: 200, description: 'Record deleted successfully.'})
	async update(@Param('id', new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE })) id: number, @Body() entity: T): Promise<T> {
	  return this.IBaseService.update(id, entity);
	}

	@Delete(':id')
	@ApiResponse({ status: 200, description: 'Record deleted successfully.'})
	@ApiResponse({ status: 400, description: 'Bad Request.'})
	async delete(@Param('id', new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE })) id: number) {
	  this.IBaseService.delete(id);
	}
}
```

Даний контролер описує всі базові CRUD-операції, що потрібні для роботи з даними в цьому додатку.

Для уніфікації відповідей сервера було розроблено інтерцептор – це утиліта яка виконує певні дії перед відправкою відповіді з сервера.

```
@Catch()
export class ApiResponceExceptionFilter implements ExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();

    let status = HttpStatus.INTERNAL_SERVER_ERROR;
    let message = 'Internal Server Error';
    let validationMessage: string[] = [];

    if (exception instanceof HttpException) {
      status = exception.getStatus();
      message = exception.message;

      const errors = exception.getResponse() as ValidationError[];
      if (errors) {
        validationMessage = errors['message']
      }
    }

    const apiResponse: ApiResponse = {
        data: [],
        success: false,
        faultString: message,
        faultCode: status.toString(),
        validationResult: validationMessage,
    };

    response.status(status).json(apiResponse);
  }
  private extractValidationErrors(errors: ValidationError[]): string[] {
    const validationErrors: string[] = [];

    for (const error of errors) {
      for (const constraint in error.constraints) {
```

```
            if (error.constraints.hasOwnProperty(constraint)) {
              validationErrors.push(error.constraints[constraint]);
            }
          }

          if (error.children && error.children.length > 0) {

validationErrors.push(...this.extractValidationErrors(error.children));
          }
        }
        return validationErrors;
      }
    }
```

Для роботи з файлами та можливості зберігати на сервері зображення створено FileService, Nest.js з коробки надає функціонал для роботи з файлами

```
    @Injectable()
    export class FileService implements MulterOptionsFactory {
      private readonly allowedFileTypes = ['.jpg', '.jpeg', '.png'];

      createMulterOptions(): MulterModuleOptions {
        return {
          storage: diskStorage({
            destination: './uploads',
            filename: this.generateFilename.bind(this),
          }),
          fileFilter: this.fileFilter.bind(this),
          limits: {
            fileSize: 1024 * 1024,
          },
        };
      }

      private generateFilename(req, file, callback) {
        const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
        callback(null,    file.fieldname    +    '-'    +    uniqueSuffix    +
extname(file.originalname));
      }

      private fileFilter(req, file, callback) {
        const ext = extname(file.originalname);
```

```
    if (!this.allowedFileTypes.includes(ext.toLowerCase())) {
      return callback(new BadRequestException('Invalid file type'));
    }
    callback(null, true);
  }
}
```

Ознайомитися з повним кодом програми можно в додатках.

3.3 Інструкція користувача

Для запуску програми потрібно мати встановлені postgresql, nestjs, npm.

Для запуску використовується команда *npm run start*, запуск прогарми наведено на рис 4.1

Рисунок 3.1 – Запуск програми

Для перегляду доступних маршрутів можно перейти в Swagger-документацію приклад якої наведено на рис. 3.2, посилання на яку з’явиться в консолі після запуску програми.

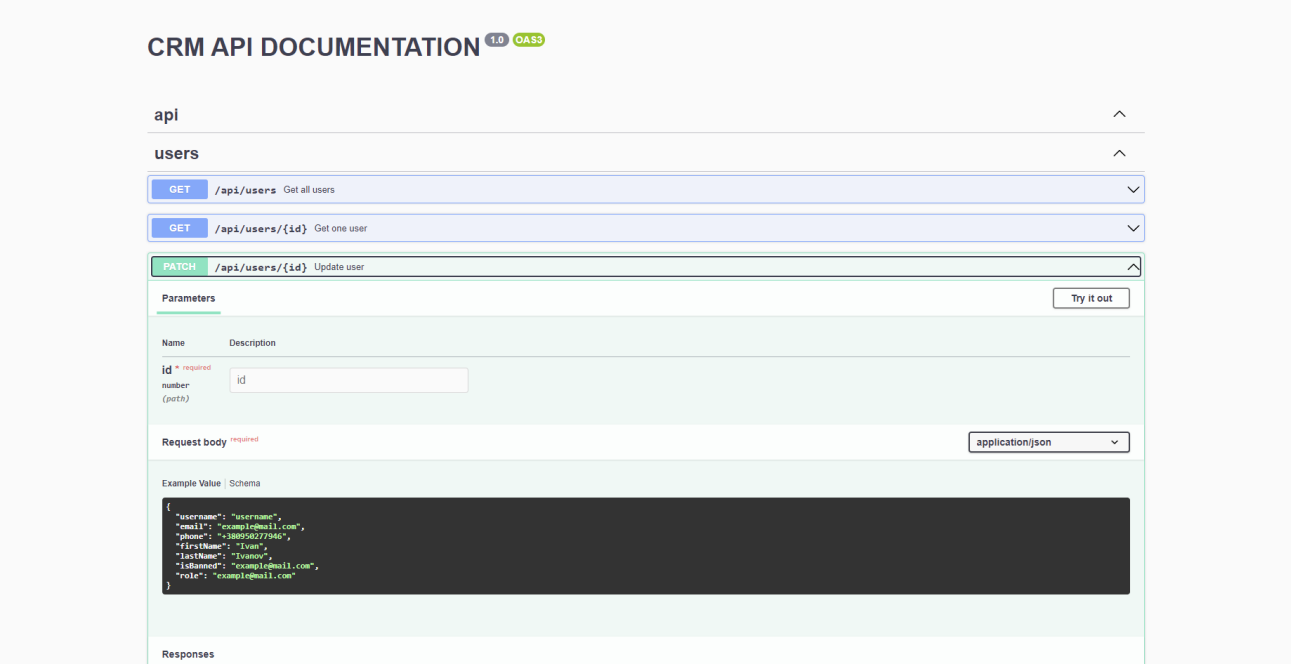

Рисунок 3.2 – Документація додатку

Для роботи з API можно використовувати будь-яке програмне забезпечення, що підтримує роботу з GET-, POST-, PATCH-, та DELETE-запитами, як на рис 3.3.

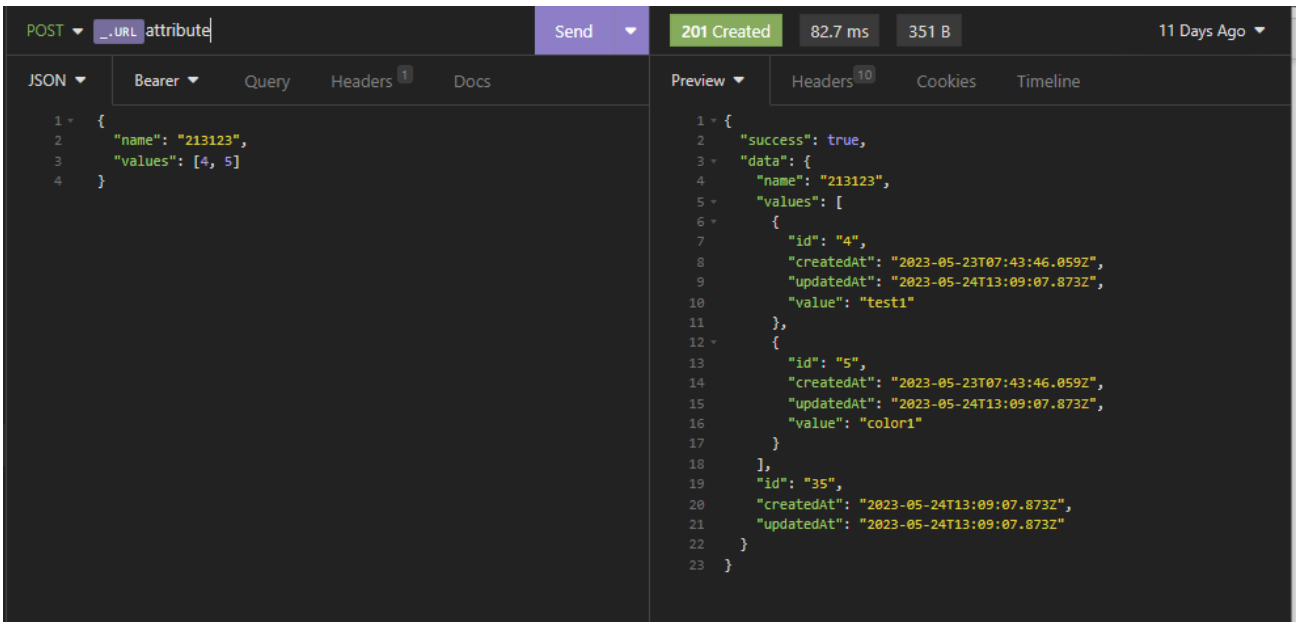

Рисунок 3.3 – Приклад запита та відповідь з сервера

ВИСНОВКИ

У результаті виконаної роботи було розроблено серверну частину системи керування контеном інтернет магазину. При цьому було розв’язано такі задачі:

1) Розроблено інформаційну модель серверної частини інтернет-магазину
2) Розроблено структуру баз даних серверної частини інтернет-магазину
3) Виконано вибір архітектури серверної частини інтернет-магазину
4) Виконано вибір мови програмування для програмної реалізації
5) Виконано вибір фреймворку
6) Виконано програмну реалізацію серверної частини інтернет-магазину
7) Проведено перевірку працездатності розробки
8) Створено інструкцію користувача

Таким чином, у результаті розробки було створено зручне REST-API для системи керування контентом інтернет магазину. Він забепезпечить стабільну та якісну роботу з даними. Розроблений додаток задовольняє всім вимогам, що були поставлені.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Каміль Мишлевіц. "Nest.js - прогресивний фреймворк Node.js." 2018

2. Томаш Трайян. Nest.js: Розробка ефективних, масштабованих та надійних серверних додатків." 2018.

3. Гергелі Немет. "Практичні шаблони та найкращі практики проектування RESTful API: Побудова ефективних RESTful API для підприємства з використанням шаблонів і REST (Hateoas)." Packt Publishing, 2019.

4. Дугальд Стір, Ехсан М. Захеді. "Практична архітектура мікросервісів: Вирівнювання принципів, практик та культури." O'Reilly Media, 2019.

5. Cech Ніл Томпсон. "Побудова безпечних та надійних систем: Кращі практики проектування, впровадження та підтримки систем." O'Reilly Media, 2020.

6. Еойн Шанагі та ін. "Мікросервіси в дії." Manning Publications, 2018.

7. Guerrero Sergio. Microservices in SAP HANA XSA: A Guide to REST APIs Using Node.js .– Apress Media LLC., 2020. — 229 p.

8. Jay Bell, Greg Magolan, David Guijarro, Adrien de Peretti, Patrick Housley. Nest.js: A Progressive Node.js Framework.– Packt, 2019. – 317 p.

9. Daniel Correa, Greg Lim Practical Nest.js: Develop clean MVC web applications. – Independently published, 2022. – 190 p.

10. Ackermann Philip. JavaScript: The Comprehensive Guide. – Rheinwerk Computing, 2022. — 982 p

11. McGregor M. JavaScript: The Hidden Parts: Building More Performant, Flexible, and Maintainable Applications (Early Release). – O’Reilly Media, 2022. — 142 p.

ДОДАТОК

main.ts

```
async function bootstrap() {
  const PORT = process.env.API_PORT || 3000;
  const api = await NestFactory.create(AppModule);

  api.use(cors({
    origin: 'http://localhost:4200',
    credentials: true
  }));
  api.use(cookieParser());

  api.setGlobalPrefix("/api");
  const config = new DocumentBuilder()
    .setTitle('CRM API DOCUMENTATION')
    .setVersion('1.0')
    .addTag('api')
    .build();
  const document = SwaggerModule.createDocument(api, config);

  SwaggerModule.setup('/api/docs', api, document);

  api.useGlobalPipes(new ValidationPipe({
    transformOptions: {
      enableImplicitConversion: true,
    },
  }));
  await api.listen(PORT, () => {
  console.log('-------------------------------');
  console.log(`Server started at ${PORT} port`);
  console.log('-------------------------------');
  console.log(`go              to              documentation              ->
http://${process.env.API_URL}:${PORT}/api/docs`);
  console.log('-------------------------------');
  });
}
bootstrap();
```

app.module.ts

```
@Module({
  providers: [
      {
        provide: APP_GUARD,
        useClass: JwtAuthGuard,
      },
      {
        provide: APP_INTERCEPTOR,
        useClass: ApiResponseInterceptor,
      },
      {
        provide: APP_FILTER,
        useClass: ApiResponceExceptionFilter,
      },
  ],
  imports: [
    ConfigModule.forRoot({
      envFilePath: '.development.env',
    }),
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: process.env.POSTGRES_HOST,
      port: Number(process.env.PORT),
      username: process.env.POSTGRES_USER,
      password: process.env.POSTGRESS_PASSWORD,
      database: process.env.POSTGRES_DB,
      synchronize: true,
      autoLoadEntities: true,
    }),
    MulterModule.registerAsync({
      useClass: FileService,
    }),
    UsersModule,
    AuthModule,
    RolesModule,
    ProductsModule,
    CustomersModule,
    OrdersModule,
    FilesModule
  ],
})
export class AppModule {
```

```
  constructor(private dataSource: DataSource) {}
}
```

user.service.ts

```
@Injectable()
export class UsersService {
    constructor(
        @InjectRepository(User)     private     readonly     usersRepository:
Repository<User>
    ) {}

    async getAllUsers() {
        const users = await this.usersRepository.find({ relations: ['role']
});
        if (!users.length)  {
            throw new NotFoundException('users not found');
        }

        users.forEach(user => delete user.password)
        return users;
    }

    async getUser(userId: number): Promise<User | null>  {
            const user = await (await this.usersRepository.findOne({ where:
{ id: userId }, relations: ['role'] }));
            if (!user) {
                throw new NotFoundException('user not found');
            }
            delete user.password;
            return user;
    }

    async updateUsers(userId: number, dto: UpdateUserDto) {
        const user = await this.getUser(userId);

        if ('email' in dto) {
            const checkEmail = await this.usersRepository.findOne({
                where: {
                    'email': dto.email
                }
            });
```

```
                if(checkEmail) {
                    if (checkEmail.id != userId) {
                        throw new BadRequestException('this email already
exist');
                    }
                }
            }
            if ('username' in dto) {
                const checkUsername = await this.usersRepository.findOne({
                    where: {
                        'username': dto.username
                    }
                });
                if(checkUsername) {
                    if (checkUsername.id != userId) {
                        throw new BadRequestException('this username already
exist');
                    }
                }
            }

            await this.usersRepository.update({
                id: userId },{
                ...dto,
                });
            delete user.password;
            return user;
        }

        async banUser(userId: number): Promise<User | null> {
            const user = await this.getUser(3);
            if (user.isBanned) {
                await this.usersRepository.update({
                    id: userId,
                    }, {
                    isBanned: false,
                    });
                    return user;
            }
            await this.usersRepository.update({
                id: userId,
                }, {
```

```
                isBanned: true,
            });
            delete user.password;
            return user;
        }

        async deleteUsers(userId: number): Promise<User | null> {
            const user = this.getUser(userId);
            await this.usersRepository.delete({ id: userId })
            return user;
        }

        async changeUserRole(userId: number, dto: ChangeRoleDto): Promise<User |
null> {
            const user = await this.getUser(userId);
            await this.getUser(userId);
            await this.usersRepository.update({
                id: userId,
              }, {
                role: dto.roleId,
              });
            delete user.password;
            return user;
        }
    } (private dataSource: DataSource) {}
    }


        @ApiOperation({summary: 'Change user role'})
        @ApiResponse({status: 200, type: User, description: 'change user role
according to the given requirements, requirements: authorization, admin role'})
        @Patch('/change-role/:id')
        changeUserRole(@Param('id', new ParseIntPipe({ errorHttpStatusCode:
HttpStatus.NOT_ACCEPTABLE })) userId: number, @Body() dto: ChangeRoleDto) {
            return this.userService.changeUserRole(userId, dto);
        }
    }
```

auth.service.ts

```
    config({ path: '.development.env' });


    @Injectable()
    export class AuthService {
        constructor(
            private jwtService: JwtService,
            private userService: UsersService,
            private mailService: MailService,
            @InjectRepository(User)    private    readonly    usersRepository:
Repository<User>,
        ) {}

        async  signUp(email:  string,  username:  string,  password:  string):
Promise<User | null> {
            const existingEmail = await this.usersRepository.findOneBy({
                email: email,
            });
            const existingUserName = await this.usersRepository.findOneBy({
                username: username,
            });
            if(existingEmail || existingUserName) {
                throw new BadRequestException('email or login already exist', {
cause: new Error(), description: 'signUp error' });
            }
            const passwordHash = bcrypt.hashSync(password, 10);
            const doc = {
                email: email,
                username: username,
                password: passwordHash,
                role: 2, // User role
            };
            const user = this.usersRepository.create(doc);
            await this.usersRepository.save(user);
            delete user.password;
            return user;
        }

        async  signIn(username:  string,  password:  string,  res:  Response):
Promise<Object | null> {
            const  user  =  await  (await this.usersRepository.findOne({  where:  {
username: username }, relations: ['role'] }));
```

```
            if (!user) {
                throw new NotFoundException('user not found', { cause: new
Error(), description: 'signIn error' });
            }

            const valid = await bcrypt.compare(password, user.password);
            if (!valid) {
                throw new UnauthorizedException('incorrect login or password', {
cause: new Error(), description: 'signIn error' });
            }
            return this.getTokens(user, res);
        }

        private async getTokens(user: User, res: Response): Promise<Object |
null> {
            try {
                const payload = {
                    id: user.id,
                    username: user.username,
                    role: user.role,
                }
                const access_token = await this.jwtService.signAsync(payload,
                    {
                        expiresIn: jwtConstants.expiresInAccess,
                    });
                const refresh_token = await this.jwtService.signAsync(payload,
                    {
                        expiresIn: jwtConstants.expiresInRefresh,
                    })
                res.cookie('jwt', access_token, { httpOnly: true, expires: new
Date(new Date().getTime() + 15 * 60000) });
                res.cookie('refresh_token', refresh_token, { httpOnly: true,
expires: new Date(new Date().getTime() + 60 * 60000) })
                return {
                    access_token,
                    refresh_token,
                    expires_in: jwtConstants.expiresInAccess
                };

            } catch (err) {
                throw new BadRequestException();
            }
```

```
        }

        async refreshToken(refreshToken: string, res: Response) {
            const result = await this.jwtService.verifyAsync(refreshToken);
            if(!result) {
                throw new UnauthorizedException('invalid refresh-token');
            }
            const user = await this.userService.getUser(result.id);
            return this.getTokens(user, res);
        }

        async resetRequest(dto: ResetPasswordDto, res: Response) {
                const existingEmail = await this.usersRepository.findOneBy({
                    email: dto.email,
                });
                const existingUserName = await this.usersRepository.findOneBy({
                    username: dto.username,
                });
                if(!existingEmail || !existingUserName) {
                    throw new BadRequestException('email or login not found');
                }

                const resetLink = mailConstants.reset_url + uuidv4();
                await this.mailService.sendResetMail(dto.email, resetLink);

                const payload = {
                    email: dto.email,
                    username: dto.username,
                    resetLink: resetLink,
                }
                const reset_token = await this.jwtService.signAsync(payload,
                    {
                        expiresIn: jwtConstants.expiresInReset,
                });
                res.cookie('reset_token', reset_token, { httpOnly: true, expires:
new Date(new Date().getTime() + 60 * 60000) });
                return [];
        }

        async redirect(res: Response) {
            res.redirect(`${process.env.FRONT_RESET_URL}/login?recover=true`);
            return [];
```

```
}

async changePassword(dto: ChangePasswordDto, req: Req, res: Response) {
    const resetToken = req.cookies?.reset_token;
    if (!resetToken) {
        throw new NotFoundException('reset token not found')
    }
    const payload = this.jwtService.decode(resetToken);

    const user = await this.usersRepository.findOneBy({
        email: payload['email']
    });
    if(!user) {
        throw new BadRequestException('user not found');
    }

    const passwordHash = bcrypt.hashSync(dto.password, 10);
    user.password = passwordHash;

    await this.usersRepository.save(user);
    res.clearCookie('reset_token');
    if (user) {
        res.json({ success: true });
    }
}

async getMyInfo(req: Req) {
    const resetToken = req.cookies?.jwt;
    const payload = this.jwtService.decode(resetToken);

    if (!payload['username']) {
        throw new BadRequestException('Incorrect token');
    }

    const user = await this.usersRepository.findOneBy({
        username: payload['username']
    });

    if(!user) {
        throw new BadRequestException('user not found');
    }
    delete user.password;
```

```
            return user;
        }
    }
```

base.controller.ts

```
    import { Get, Param, Post, Patch, Delete, Body, ParseIntPipe, HttpStatus, BadRequestException } from "@nestjs/common";
    import { IBaseEntity } from "../interfaces/database.interface";
    import { ApiResponse } from "@nestjs/swagger";
    import { IBaseCrudService } from "../interfaces/crud.service.interface";
    import { validate } from "class-validator";

    export class BaseController<T extends IBaseEntity> {
        constructor(private readonly IBaseService: IBaseCrudService<T>) {}

     @Get()
     @ApiResponse({ status: 200, description: 'Ok'})
     async findAll(): Promise<T[]> {
       return this.IBaseService.getAll();
     }

     @Get(':id')
     @ApiResponse({ status: 200, description: 'Record retrieved successfully.'})
     @ApiResponse({ status: 404, description: 'Record does not exist'})
     async findById(@Param('id', new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE })) id: number): Promise<T> {
       return this.IBaseService.getOneById(id);
     }

     @Post()
     @ApiResponse({ status: 201, description: 'The record has been successfully created.'})
     @ApiResponse({ status: 403, description: 'Forbidden.'})
     @ApiResponse({ status: 400, description: 'Bad Request.'})
     async create(@Body() entity: T): Promise<T> {
           return this.IBaseService.create(entity);
     }

     @Post('update/:id')
     @ApiResponse({ status: 400, description: 'Bad Request.'})
     @ApiResponse({ status: 200, description: 'Record deleted successfully.'})
```

```
    async update(@Param('id', new ParseIntPipe({ errorHttpStatusCode:
HttpStatus.NOT_ACCEPTABLE })) id: number, @Body() entity: T): Promise<T> {
      return this.IBaseService.update(id, entity);
    }

    @Post('delete/:id')
    @ApiResponse({ status: 200, description: 'Record deleted successfully.'})
    @ApiResponse({ status: 400, description: 'Bad Request.'})
    async delete(@Param('id', new ParseIntPipe({ errorHttpStatusCode:
HttpStatus.NOT_ACCEPTABLE })) id: number) {
      this.IBaseService.delete(id);
    }
  }
```

base.service.ts

```
  import { NotFoundException, BadRequestException } from '@nestjs/common';
  import { FindManyOptions, FindOneOptions, FindOptionsWhere,
FindOptionsWhereProperty, Repository } from 'typeorm';
  import { IBaseEntity } from '../interfaces/database.interface';
  import { IBaseCrudService } from '../interfaces/crud.service.interface';


  export class BaseService<T extends IBaseEntity> implements
IBaseCrudService<T> {
    private readonly options: FindManyOptions<T>;
      private readonly searchField: keyof T;

      constructor (
          private readonly genericRepository: Repository<T>,
          private readonly relationOptions?: FindManyOptions<T>,
          private readonly relatedRepositories?: {[key: string]:
Repository<any>},
          searchField?: keyof T,
      ) {
          this.options = relationOptions || {};
          this.relatedRepositories = relatedRepositories || null;
          this.searchField = searchField || 'id';
      }

      protected async checkRelated(entity: T) {
          const relationArray = Object.keys(this.relatedRepositories);
```

```
            for (const relation of relationArray) {
                if (entity.hasOwnProperty(relation)) {
                    const relatedEntity = entity[relation];
                    const                    relatedRepository                    =
this.relatedRepositories[relation];

                    if (Array.isArray(entity[relation])) {
                        //if relation have several items (relation - Array)
                        for (const item of entity[relation]) {
                            if (!item.id) {
                                throw new BadRequestException(`Invalid data in
${relation} id not specified`);
                            }

                            const findOptions = {
                                where: {
                                    id:                item.id                as
FindOptionsWhereProperty<NonNullable<T['id']>, any>
                                },
                            } as FindOneOptions<T>;
                            const                  foundEntity                  =
relatedRepository.findOne(findOptions);
                            if (!foundEntity) {
                                throw new BadRequestException(`${relation} for
${relatedEntity.id} id not found`);
                            }
                        }

                    } else {
                        if (!relatedEntity.id) {
                            throw   new   BadRequestException(`Invalid   data   in
${relation} id not specified`);
                        }
                        const findOptions = {
                            where: {
                                id:              relatedEntity.id              as
FindOptionsWhereProperty<NonNullable<T['id']>, any>
                            },
                        } as FindOneOptions<T>;
                        const                    foundEntity                    =
relatedRepository.findOne(findOptions);
                        if (!foundEntity) {
```

```
                            throw new BadRequestException(`${relation} for
${relatedEntity.id} id not found`);
                        }
                    }
                }
            }

        async create(entity: T): Promise<T> {
            try {
                const findOptions: FindOneOptions<T> = {
                    where: {
                        [this.searchField]: entity[this.searchField],
                    } as FindOptionsWhere<T>,
                };
                const record = await
this.genericRepository.findOne(findOptions);

                if(record) {
                    throw new BadRequestException('record is already exist');
                }

                if (this.relatedRepositories) {
                    this.checkRelated(entity);
                }

                const createRecord = await
this.genericRepository.create(entity);
                const saveRecord = await
this.genericRepository.save(createRecord);

                return saveRecord;
            } catch (e) {
                console.log(e)
            }

        }

        async getAll(): Promise<T[]> {
            const records =  await this.genericRepository.find(this.options);
            if(records.length === 0) {
                throw new NotFoundException('table is empty');
```

```
        }
        return records;
    }

    async getOneById(id: number): Promise<T> {
        const findOptions = {
            where: {
              id: id as FindOptionsWhereProperty<NonNullable<T['id']>, any>
            },
            ...this.options
        } as FindOneOptions<T>;
        const record = await this.genericRepository.findOne(findOptions);
        if (!record) {
            throw new NotFoundException('record not found')
        }

        return record;
    }

    async update(id: number, entity: T): Promise<T> {
        const record = await this.getOneById(id);
        Object.assign(record, entity);
        return await this.genericRepository.save(record);
    }

    async delete(id: number): Promise<T> {
        const record = await this.getOneById(id);
        await this.genericRepository.delete(id);
        return record;
    }
}
```

api-responce.interface.ts

```
export interface ApiResponse<Data = any> {
    data?: Data;
    success: boolean;
    faultString?: string;
    faultCode?: string;
    validationResult?: string[];
}
```