МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Секція інформаційно-комунікаційних технологій

ВИПУСКНА РОБОТА

на тему:

«REST API «Розпродаж» з використанням технологій PostgreSQL та Node JS»

Завідувач
випускаючої кафедри **Довбиш А.С.**

Керівник роботи **Шутилєва О.В.**

Студент гр. ІН-71 **Ращупкін Я.В.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедри Довбиш А.С.

"_____" _____2021 р.

ЗАВДАННЯ
до випускної роботи

Студента четвертого курсу, групи ІН-71 спеціальності "122 – Комп'ютерні науки» денної форми навчання Ращупкіна Ярослава Васильовича.

Тема: «REST API 'Розпродаж' з використанням технологій NodeJS та Postgres»

Затверджена наказом по СумДУ

№_____ від _____2021 р.

Зміст пояснювальної записки: 1) огляд та аналіз існуючих аналогів; 2) постановка завдання й формулювання завдань дослідження; 3) огляд технологій для реалізації завдання; 4) розробка REST API; 5) аналіз результату.

Дата видачі завдання "_____"_____2020 р.

Керівник випускної роботи _____ Шутилєва О.В.

Завдання прийняв до виконання _____ Ращупкін Я.В.

РЕФЕРАТ

Записка: 53 стор., 11 рис., 4 табл., 4 додатки, 13 джерел.

Об’єкт дослідження – актуальність онлайн продажів.

Мета роботи – розробка REST API для продажу товарів онлайн. Покриття додатку тестами, автоматизація документації.

Методи дослідження – метод аналітично-статистичний.

Результати – розроблено REST API додаток для продажу/купівлі товарів. Додаток створено на базі мови програмування Typescript з використанням фреймворку Nest JS та інтеграціями AWS S3, Swagger API. Покриття unit та E2E тестами з використанням фреймворків Jest та Supertest.

REST API,ОБРОБКА СТАТИСТИЧНИХ ДАНИХ, МЕТОД АНАЛІТИЧНО-СТАТИСТИЧНИЙ, E2E-ТЕСТУВАННЯ, UNIT-ТЕСТУВАННЯ, ХМАРНЕ ЗБЕРЕЖЕННЯ ФАЙЛІВ

ЗМІСТ

ВСТУП

Інтернет перевернув наше існування з ніг на голову. Він зробив революцію в сфері комунікацій. Інтернет проник майже у всі види нашої діяльності. Ми замовляємо їжу, купуємо техніку, спілкуємося з друзями використовуючи цю технологію. Сьогодні досить одного-двох кліків миші, щоб прочитати місцеву газету або будь-які новини, оновлювані з точністю до хвилини.

Нові технології збільшують швидкість передачі інформації, це відкриває можливість покупок онлайн. Інтернет пропонує величезну кількість можливостей для покупки контенту, новин, товарів для відпочинку. Багато переваг виникає завдяки електронній комерції, яка стала основним каналом розповсюдження товарів і послуг. Клієнт може замовити авіаквитки чи футболку, купити продукти в продуктовому чи придбати автомобіль, і все це - онлайн. Нові додатки створюють комерційні можливості.

Існує безліч інтернет-магазинів, де кожна людина має можливість придбати товар, який її цікавить. Однак значно меншу долю електронної комерції займають онлайн-аукціони. Для когось це можливість продати дорогу річ, потреби в якій більше немає, хтось хоче мати нагоду придбати цінну річ. Розпродажі популярні як у колекціонерів, так і у звичайних покупців.

Метою випускної роботи є створення універсальної платформи для проведення онлайн розпродажу проблеми. Створений додаток буде вирішенням актуальної на сьогодні проблеми - відсутності кросс-платформенного, маштабованого та гнучкого ресурсу для онлайн продажу.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Патерн проектування

REST API − це інтерфейс прикладного програмування (API або веб-інтерфейс), який відповідає обмеженням архітектурного стилю REST і дозволяє взаємодіяти з веб-службам. Розшифровується як Representational State Transfer.

API − це набір значень і протоколів для створення і інтеграції прикладного програмного забезпечення. Іноді його називають контрактом між постачальником інформації і користувачем інформації, що встановлює зміст, необхідний клієнтом, і зміст, необхідний сервером.

API можна розглядати як посередника між користувачами і ресурсами або веб-службами, які вони хочуть отримати. Це також спосіб для організації обміну ресурсами і інформацією, зберігаючи при цьому безпеку, контроль і аутентифікацію − визначення того, хто і до чого отримує доступ [1].

Основні переваги обраного паттерну:

- REST API легко зрозуміти і вивчити, завдяки його простоті;
- за допомогою REST API можна організувати складні додатки і спростити використання ресурсів;
- REST API легко вивчати;
- він спрощує роботу нових клієнтів з іншими додатками, незалежно від того, розроблені вони спеціально для цієї мети чи ні;
- для отримання даних і запитів використовуються стандартні виклики HTTP-процедур.

1.2 Огляд Web-ресурсів

Сьогодні у всесвітній мережі існує багато веб-ресурсів, які виконують роль інтернет-аукціону, і кожного дня їх кількість зростає.

Перед розробкою REST API був проведений огляд вже існуючих сайтів, таких як:

- https://ebay.com;
- http://amazon.com;
- https://ebid.com.

Кожен, із вищевказаних сайтів має свої особливості, які виділяють його.

Основною ідеєю eBay (рис. 1) є надання продавцям інтернет-платформи для продажу будь-яких товарів. Сама фірма eBay виступає лише в ролі посередника при укладанні договору купівлі-продажу між продавцем і покупцем. Оплата товару і його пересилання відбувається без участі eBay. За використання платформи продавці платять внесок, зазвичай складається зі збору за виставлення лота і відсотка від ціни продажу.

Amazon (рис. 2) − другий за величиною аукціон в світі. Компанія була створена в 1994 році американським підприємцем Джеффом Безосом, а сайт був запущений в 1995-му році. Спочатку на сайті продавалися тільки книги.

Це величезне інтернет-підприємство, яке продає книги, музику, фільми, товари для будинку, електроніку, іграшки та багато інших товарів або безпосередньо, або в якості посередника між іншими роздрібними торговцями і мільйонами клієнтів Amazon.com. Бізнес з надання веб-послуг включає в себе оренду сховищ даних і обчислювальних ресурсів, так звані "хмарні обчислення", через Інтернет. Присутність компанії в мережі настільки велика, що в 2012 році 1 відсоток всього інтернет-трафіку в Північній Америці проходив через центри обробки даних Amazon.com.

Компанія також виробляє лідируючі на ринку пристрої для читання електронних книг Kindle. Просування цих пристроїв призвело до різкого зростання обсягів випуску електронних книг і перетворило Amazon.com в одну з головних руйнівних сил на ринку книговидання.

Ebid (рис. 3) − найбільший інтернет аукціон, має широке представництво на всіх континентах світу, мільйони зареєстрованих користувачів в багатьох країнах Азії, Європи, Африки, Америки. За масштабністю поступається тільки

eBay. Сайт аукціону багатомовний, але підтримка російської поки відсутня. Незважаючи на те, що аукціон позиціонує себе як eBid vs eBay, протиставляючи навіть в назві, все ж це більше копія лідера аукціонних торгів, ніж інноваційна схема торгів. Втім, відмінності є, але для розуміння де краще, і що гірше, треба попрацювати на обох ресурсах.

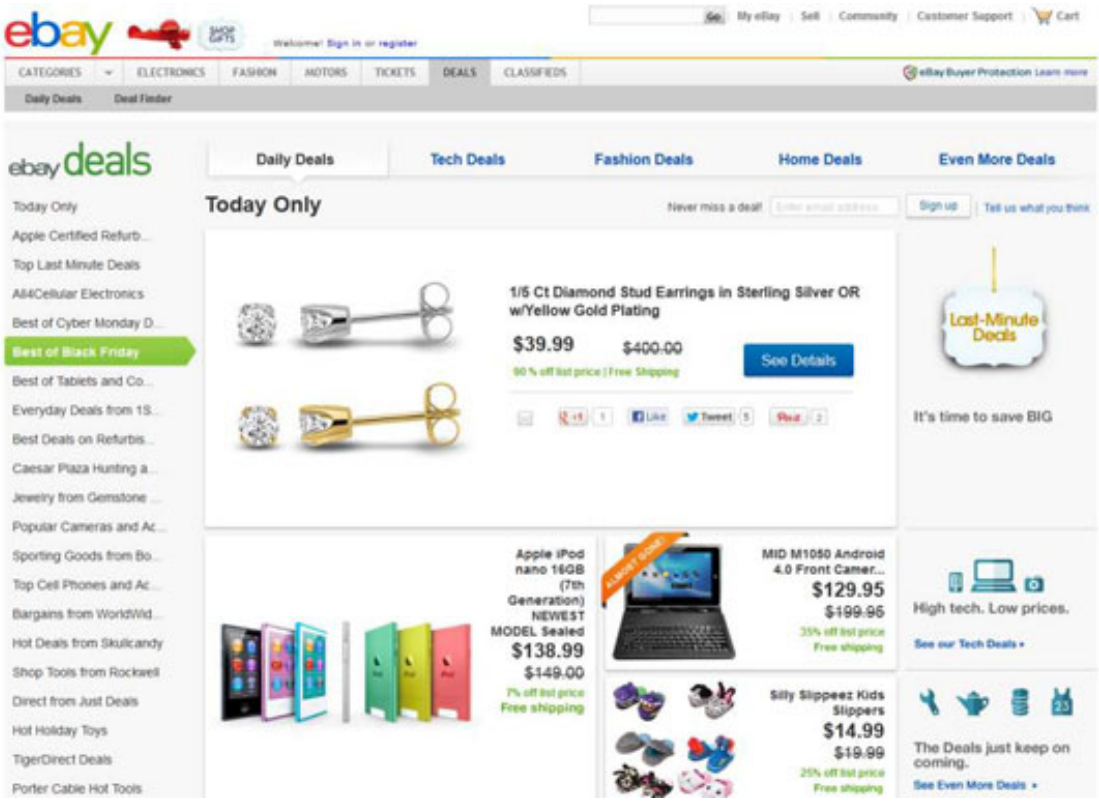

Рисунок 1 − Веб-сайт Ebay

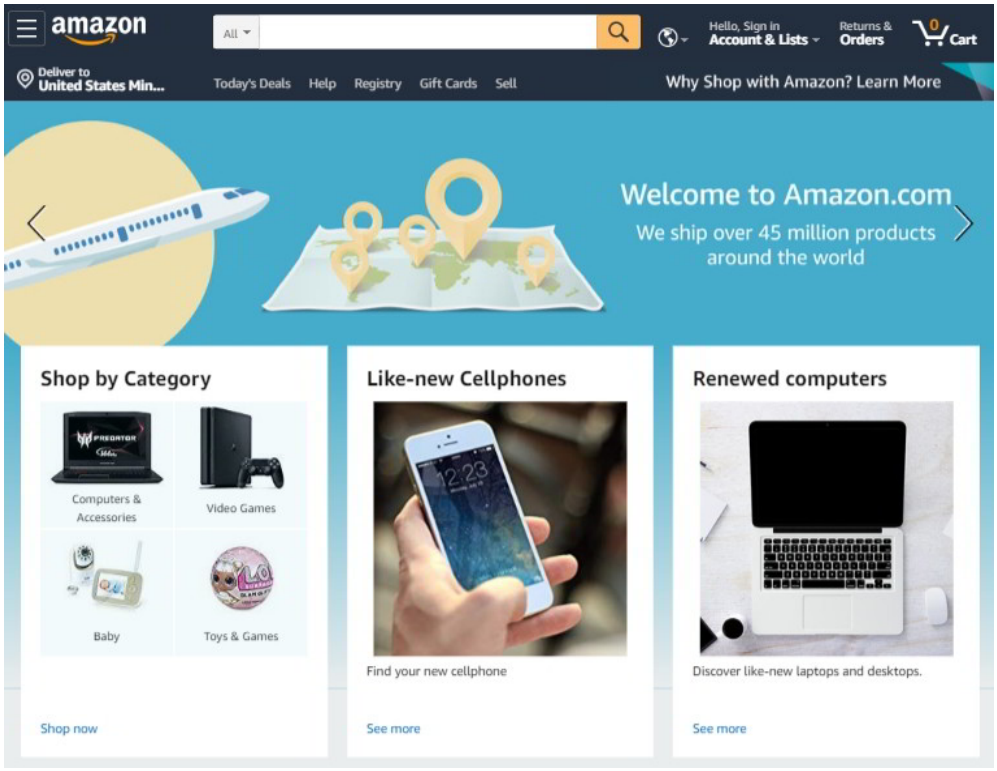

Рисунок 2 − Веб-сайт Amazon

Рисунок 3 − Веб-сайт Ebid

1.3 Порівняльний аналіз сайтів для онлайн продажів

На основні популярних веб-ресурсів для онлайн продажу, проведемо порівняльний аналіз, виділивши основні переваги та недоліки. Стратегія

порівняльного аналізу дозволить нам перейняти сильні сторони конкурентів, а також мінімізувати їх недоліки у власному додатку.

Таблиця 1 − Порівняльний аналіз

	Ebay	**Amazon**	**eBid**
Швидкість роботи	-	+	-
Універсальність	-	+	-
Масштабованість за потреби	-	-	-
Гнучкість	-	+	-
Ціна	-	-	-
Безпека	+	+	+
Зручність	+	+	-
Автоматизація	-	+	-

Провівши аналіз основних платформ для онлайн-продажу можна виділити за згрупувати основні переваги та недоліки.

Переваги:

- кожен з ресурсів є універсальним методом продажу. Дуже легко розмістити будь який товар, та знайти покупця;
- додатки мають високий рівень захищеності, тож особисті дані користувачів надійно захищені;
- додатками зручно користуватися. Інтерфейси логічно зрозумілі, дизайн відповідає останнім стандартам.

Недоліки:

- відносно повільна швидкість роботи через великий обсяг товарів;
- ресурси важко масштабувати, адже кожна зміна впливає на безліч факторів;
- всі ресурси мають платний функціонал;

- немає можливості автоматизувати продаж товарів, створити графік для старту продажу товарів.

Виявлені переваги та недоліки допоможуть у створенні REST API для продажу товарів.

1.4 Постановка задачі

Розробити REST API, який матиме серверну частину з необхідним функціоналом. Необхідний функціонал:

- реєстрація, можливість створити аккаунт с унікальним номером мобільного телефону чи електронною поштою;
- підтримка JWT авторизації;
- відправка листів на електронну пошту, використовуючи SMTP сервіс;
- можливість створювати нові товари та оновлювати їх, налаштування графіку переведення товару у активний статус;
- завантаження медіафайлів;
- можливість створення ставок на необхідний товар;
- автоматизація листування та обрання переможця лоту;
- створення замовлень на виграний лот.

Проект повинен мати UNIT-тести та E2E-тести. Для розробки потрібно використати систему контролю версій. Необхідно розглянути можливості та визначати підхід для побудови масштабованого REST API додатку.

2 ВИБІР МЕТОДУ РІШЕННЯ

2.1 Вибір мови програмування

Серед мов програмування, які відповідають критеріям проекту, можна виділити наступні: PHP, Python, Typescript, Java.

2.1.1 PHP

PHP − це поширена мова програмування загального призначення з відкритим вихідним кодом. PHP спеціально сконструйований для веб-розробок та його код може знаходитись безпосередньо в HTML. Хоча PHP, головним чином, призначений для роботи в середовищі веб-серверів, область його застосування не обмежується тільки цим [2].

Області застосування PHP:

- створення скриптів для виконання на стороні сервера;
- створення скриптів для виконання в командному рядку;
- створення віконних додатків, що виконуються на стороні клієнта.

Переваги PHP:

- орієнтація на Web-розробку − PHP створювався, розвивався і підтримується як мова для створення Web-сайтів;
- кросплатформеність − PHP перенесена на всі основні;
- операційні системи: можна розробляти сайт на Windows, Mac OS X, а експлуатувати на Linux-сервері. Складнощі перенесення будуть мінімальні;
- низький поріг входу − вивчити PHP і почати створювати на ньому готові програми набагато простіше, ніж з використанням конкуруючих технологій.

Недоліки PHP:

- непослідовний синтаксис − при вивченні мови PHP, особливо старої частини;

● PHP − вже застаріла. У міру життєвого циклу мова обросла додатковими ключовими словами, артефактами, застарілими конструкціями, які начебто є, працюють, але якими не рекомендують користуватися;

● безпека. У PHP є засоби безпеки рівня системи і рівня web-додатку. Але, знову ж таки, популярність PHP в деякій мірі стала його недоліком: баги в PHP знаходять швидше, ніж розробники встигають їх закривати. У PHP 7 безліч проблем вирішено, але зловмисник завжди попереду.

2.1.2 Python

Python − це високорівнева мова програмування загального призначення, яка використовується в тому числі і для розробки веб-додатків. Мова орієнтована на підвищення продуктивності розробника і читання коду [3].

Області застосування Python

● розробка веб-сайтів;

● машинне навчання;

● робота з даними.

Переваги Python:

● гнучкість − це, на мою думку, основна перевага мови, так як завдяки своїй гнучкості мова отримала популярність серед багатьох розробників;

● можливість розширення − існує багато бібліотек і фреймворів під будь-який тип завдань і потреб. Також величезним плюсом є те, що ми можемо використовувати C код з Python;

● простота синтаксису. З синтаксису було прибрано все зайве, код чистий і зрозумілий без зайвих дужок і виразів;

● інтерпретатор Python існує для всіх популярних платформ і за замовчуванням входить в більшість дистрибутивів Linux, а значить є на більшості серверів «з коробки»;

● PEP − єдиний стандарт для написання коду, що робить код підтримуваним і читабельним навіть при переході від одного програміста до іншого. Це підтримує популярність Python;

● ком'юніті − навколо Python утворилося досить дружнє і приємне ком'юніті, яке готове прийти на допомогу будь-якому починаючому або вже вмілому розробнику і розібратися в його проблемі.

Недоліки Python:

● продуктивність. Більшість розробників, та й сам творець мови, сходяться на думці, що Python не такий спритний, наскільки хотілося б. Це обумовлено тим, що Python інтерпретована мова;

● синтаксис це і недолік теж, так як якщо користувач переходить з іншої мови програмування, синтаксис буде незвичний і трохи дивним для вас;

● динамічна типізація - через динамічної типізації Python споживає більше ресурсів, ніж міг би, але це часто компенсується внутрішнім кешування;

● Global Interpreter Lock. На даний момент це є основною проблемою продуктивності в Python, а також цим обумовлена погана реалізація багатопоточності.

2.1.3 Java

Java є мовою програмування і платформою обчислень, яка була вперше випущена Sun Microsystems в 1995 р.. Існує безліч додатків і веб-сайтів, які не працюють при відсутності встановленої Java, і з кожним днем число таких веб-сайтів і додатків збільшується. Java відрізняється швидкістю, високим рівнем захисту і надійністю [4].

Області застосування Java:

● мережеве програмування;

● створення мобільних додатків для пристроїв на базі Android;

● розробка back-end-а;

● створення API для БД;

● розробка настільних додатків;

● цифрова обробка графічних файлів.

Переваги Java:

● при розробці Java було приділено велику увагу простоті мови, тому програми на Java, в порівнянні з програмами на інших мовах, простіше написати, компілювати, налагоджувати і вивчати;

● Java − це об'єктно-орієнтована мова. Це дозволяє створювати модульні програми, вихідний код яких може використовуватися багаторазово;

● мова Java не залежить від платформи. Одним з основних переваг мови Java є можливість перенесення програм з однієї системи в іншу. Оскільки програми на Java не залежать від платформи як на рівні вихідного коду, так і на довічним рівні, їх можна запускати в різних системах.

Недоліки Java:

● платне комерційне використання;

● низька продуктивність. У будь-якої мови високого рівня досить низька продуктивність через компіляції та абстракції за допомогою віртуальної машини. Однак це не єдина причина низької швидкості Java. Наприклад, додаток очищення пам'яті: це корисна функція, яка, на жаль, призводить до значних проблем з продуктивністю, якщо вимагає більше 20 відсотків часу процесора;

● складний код. Такий код може здатися перевагою, що допоможе при вивченні мови, однак, довгі, надмірно складні речення ускладнюють читання і перегляд коду;

2.1.4 Typescript

TypeScript − це мова програмування, в якій виправлені багато недоліків JavaScript. Код на TypeScript компілюється в JS і підходить для розробки будь-яких проектів під будь-які браузери − тим більше що можна вибрати версію JS, в яку компілюється код. Це проект з відкритим вихідним кодом, тому він дуже швидко розвивається. Багато того, що з'являється в TS, пізніше переходить і в JavaScript [5].

Області застосування Typescript:

● розробка клієнтських додатків;

- розробка серверних додатків;
- розробка мобільних додатків;
- розробка настільних додатків.

Переваги TypeScript:

- TypeScript − строго типізована і компільована в JavaScript мова. Проста для освоєння Java і C # програмістами;
- TypeScript реалізує багато концепцій ООП, такі як успадкування, поліморфізм, інкапсуляція і модифікатори доступу. У ній є класи, інтерфейси і абстрактні класи;
- потенціал мови дозволяє швидше і простіше писати складні комплексні рішення, які легше розвивати і тестувати надалі, ніж на стандартному JavaScript;
- TypeScript − підмножина JavaScript, тому будь-який код на JavaScript буде виконаний і в TypeScript. Це, і є його головна перевага перед конкурентами.

Недоліки TypeScript:

- у процесі розробки маємо справу з файлами * .ts, * .d.ts, * .map, * .js. Занадто багато додаткових файлів, що буває незручно, якщо проект невеликий;
- не всі браузери підтримують налагодження TypeScript в консолі без зайвих налаштувань;
- безліч нетривіальних класів. Щоб писати код, спираючись на класи, доводиться тримати в голові яка властивість де знаходиться;
- неявна статична типізація. Завжди можна описати тип як any, що за фактом відключить приведення до конкретного типу цієї змінної;

Проаналізувавши мови програмування, які підходять для створення REST API веб додатку, було вирішено зупинитися на Typescript. Середовище NodeJS має безліч фреймворків, які спрощують написання коду, а Typescript дозволяє писати код в ООП стилі, який легко масштабувати та читати.

2.2 Вибір фреймворку

Існує багато фреймворків для написання коду в середовищі NodeJS. Основними з них є: AdonisJS, ExpressJS, NestJS, LoopBack.js.

2.2.1 AdonisJS

AdonisJs − один з найпопулярніших фреймворків Node.js, що працює на всіх основних операційних системах. Він володіє статичної екосистемою для написання серверних веб-додатків. Таким чином, можна вибрати відповідний пакет, орієнтуючись на певні бізнес-потреби [6].

Особливості фреймворку AdonisJS

- підтримка ORM, що складається з баз даних SQL;
- ефективне створення SQL-запиту на основі активної записи;
- легкий в освоєнні конструктор запитів для створення швидких запитів;
- забезпечення підтримки баз даних No-SQL, таких як MongoDB.

2.2.2 ExpressJS

Express.js − це самий простий і швидкий фреймворк Node.js, який використовується в якості проміжного обробника для управління серверами і маршрутами. Express.js підходить для розробки простих додатків, які можуть обробляти декілька запитів одночасно і спираються на можливості технології Express [7].

Особливості фреймворку ExpressJS

- можливість налаштування;
- низька крива навченості;
- орієнтований на браузер.

2.2.3 Nest JS

Nest.js − один з типів фреймворків Node.js, який використовується для розробки професійних і масштабованих серверних додатків Node.js. Він був написаний на TypeScript, але використовує JavaScript. Оскільки він

розроблений з допомогою TypeScript, то поєднує в собі елементи об'єктно-орієнтованого програмування, функціонального програмування і функціонального реактивного програмування [8].

Особливості фреймворку Nest JS

- архітектура додатків "з коробки";
- легкість в створенні кого тестують і масштабованих додатків високої якості;
- генерація додатків Nest.js за допомогою Nest CLI.

2.2.4 LoopBack.js

LoopBack.js − ще один широко поширений фреймворк Node.js зі зручним інтерфейсом командного рядка і динамічним API explorer. З його допомогою можна створювати різні моделі в залежності від необхідної схеми (або навіть при відсутності схеми). Він має гарну сумісністю з різними службами REST і типами баз даних, включаючи MySQL, MongoDB, Oracle, Postgres і багато інших [9].

Особливості фреймворку LoopBack.js

- швидке створення динамічних наскрізних REST API;
- краще з'єднання між різними пристроями і браузерами;
- покращена кореляція між різними типами даних і сервісами;
- використання коштів розробки Android, iOS і Angular для створення клієнтських додатків.

Серед описаних фреймворків було вирішено обрати Nest JS, оскільки його архітектура легко маштабована, є підтримка Typescript, CLI, є підтримка принципів SOLID.

2.3 Вибір СУБД

База даних − це впорядкований набір структурованої інформації або даних, які зазвичай зберігаються в електронному вигляді в комп'ютерній

системі. Дані разом з СУБД, а також додатки, які з ними пов'язані, називаються системою баз даних, або, для стислості, просто базою даних.

СУБД − це комплекс програмно-мовних засобів, що дозволяють створити базу даних і управляти даними.

Реляційні бази даних стали переважати в 1980-х роках. Дані в реляційної базі організовані у вигляді таблиць, що складаються із стовпців і рядків. Реляційна СУБД забезпечує швидкий і ефективний доступ до структурованої інформації.

База даних NoSQL, або нереляційна база даних, дає можливість зберігати і обробляти неструктуровані або слабоструктуровані дані. Популярність баз даних NoSQL зростає в міру поширення і ускладнення веб-додатків.

SQL − простими словами, це мова програмування структурованих запитів, яка використовується в якості ефективного способу збереження даних, пошуку їх частин, поновлення, вилучення з бази і видалення.

Проект буде містити значну кількість даних, реляційна база даних максимізує зручність роботи з ними [10].

Розглянемо 3 основних СУБД: MySQL, Oracle, PostgreSQL.

2.3.1 MySQL

Вважається однією з найпоширеніших СУБД. MySQL − реляційна СУБД з відкритим вихідним кодом, головними плюсами якої є її швидкість і гнучкість, яка забезпечена підтримкою великої кількості різних типів таблиць.

Крім того, це надійна безкоштовна система з простим інтерфейсом і можливістю синхронізації з іншими базами даних. У сукупності ці фактори дозволяють використовувати MySQL як великим корпораціям, так і невеликим компаніям.

2.3.2 Oracle

Перша версія цієї об'єктно-реляційної СУБД з'явилася в кінці 70-х, і з тих пір зарекомендувала себе як надійна, функціональна і практична. СУБД Oracle

постійно розвивається і допрацьовується, спрощуючи установку і первинне налаштування та розширюючи функціонал.

Однак істотним мінусом даної СУБД є висока вартість ліцензії, тому вона використовується в основному великими компаніями і корпораціями, які працюють з величезними обсягами даних.

2.3.3 PostgreSQL

СУБД PostgreSQL − ще одна популярна і безкоштовна система. Найбільше застосування знайшла для управління БД веб-сайтів і різних сервісів. Вона універсальна, тобто підійде для роботи з більшістю популярних платформ.

При цьому PostgreSQL − об'єктно-реляційна СУБД, що дає їй деякі переваги над іншими безкоштовними СУБД, в більшості − реляційними.

Розглянувши основні СУБД, було обрано PostgreSQL, тому що вона об'єктно-реляційна і забезпечує швидкий і ефективний доступ до структурованої інформації.

2.4 Інструмент тестування Postman

Postman − один з найпопулярніших інструментів тестування програмного забезпечення, який використовується для тестування API. За допомогою цього інструменту розробники можуть легко створювати, тестувати, ділитися і документувати API.

Postman − це автономна платформа для тестування API (Application Programming Interface) для створення, тестування, проектування, модифікації і документування API. Це простий графічний користувальницький інтерфейс для відправки і перегляду HTTP-запитів і відповідей. При використанні Postman для цілей тестування не потрібно писати мережевий код HTTP-клієнта. Замість цього ми створюємо набори тестів, звані колекціями, і дозволяємо Postman взаємодіяти з API.

У цей інструмент вбудована практично будь-яка функціональність, яка може знадобитися будь-якому розробнику. Цей інструмент здатний виконувати різні типи HTTP-запитів, такі як GET, POST, PUT, PATCH, і перетворювати API в код для таких мов, як JavaScript та Python.

2.5 Менеджер пакетів Yarn

Yarn − це менеджер пакетів для коду. Він дозволяє вам використовувати код і ділитися ним з іншими розробниками з усього світу. Yarn робить це швидко, безпечно і надійно, щоб користувач ніколи не турбувався.

Yarn дозволяє вам використовувати рішення інших розробників для вирішення різних проблем, що полегшує розробку програмного забезпечення. Якщо у користувача виникають проблеми, він може повідомити про них або внести свій вклад, а коли проблема буде усунена, користувач може використовувати Yarn для підтримки всього цього в актуальному стані.

Спільне використання коду здійснюється через щось, зване пакетом (іноді зване модулем). Пакет містить файл package.json, який описує пакет [11].

2.6 Swagger

Swagger дозволяє описувати структуру API таким чином, щоб машини могли їх читати. Здатність API описувати свою власну структуру є коренем Swagger. Читаючи структуру API, Swagger може автоматично створювати красиву і інтерактивну документацію API. Swagger також може автоматично генерувати клієнтські бібліотеки для API на багатьох мовах і використовувати інші можливості, такі як автоматизоване тестування. Swagger робить це, запитуючи API для повернення YAML або JSON, які містять докладний опис всього API [12].

2.7 Git

Git – це найбільш поширена система контролю версій. Git відстежує зміни, які користувач вносить у файл, щоб у нього був запис про те, що було зроблено, і він міг повернутися до певних версій, якщо вам це знадобиться. Git також полегшує спільну роботу, дозволяючи об'єднувати зміни, внесені декількома людьми, в одне джерело. Тому незалежно від того, пишете користувач код, який будете бачити тільки він, або працює в команді, Git буде йому корисний.

Git - це програмне забезпечення, яке працює локально. Файли і їх історія зберігаються на комп'ютері. Користувач також може використовувати онлайн-хостинги (наприклад, GitHub або Bitbucket) для зберігання копій файлів і історії їх ревізій. Наявність централізованого місця, куди він можете завантажувати свої зміни і завантажувати зміни інших, дозволяє вам легше співпрацювати з іншими розробниками. Git може автоматично об'єднувати зміни, тому дві людини можуть навіть працювати над різними частинами одного і того ж файлу і пізніше об'єднати ці зміни без втрати роботи один одного! [13]

2.8 UNIT-тестування

Модульні тести − це зазвичай автоматизовані тести, які написані і виконуються розробниками програмного забезпечення, щоб переконатися, що частина програми (відома як "модуль") відповідає своєму дизайну і поводиться так, як задумано. У процедурному програмуванні модулем може бути цілий модуль, але найчастіше це окрема функція або процедура. У об'єктно-орієнтованому програмуванні одиницею часто є цілий інтерфейс, наприклад, клас, або окремий метод. Написавши тести спочатку для самих маленьких тестованих одиниць, а потім для складових моделей поведінки між ними, можна створити комплексні тести для складних додатків.

2.9 E2E-тестування

Кінцеве тестування (E2E-тестування) відноситься до методу тестування програмного забезпечення, який включає в себе тестування робочого процесу додатку від початку до кінця. Цей метод в основному спрямований на відтворення реальних користувальницьких сценаріїв, щоб система могла бути перевірена на інтеграцію і цілісність даних.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Визначення основних сутностей

Визначимо основні сутності. Система матиме 4 основних сутності: користувач, товар, ставка, замовлення.

Таблиця 2 − Основні сутності системи

Назва сутності	**Властивості**
Користувач	Пошта, ім’я, прізвище, пароль, телефон, дата народження
Товар	Назва, опис, стартова ціна, кінцева ціна, старт продажу, кінець продажу, фото, статус
Ставка	Запропонована ціна
Замовлення	Вид доставки, поштова компанія, статус

Крім вказаних властивостей сутності матимуть базові властивості: дата створення, дата оновлення, ідентифікатор.

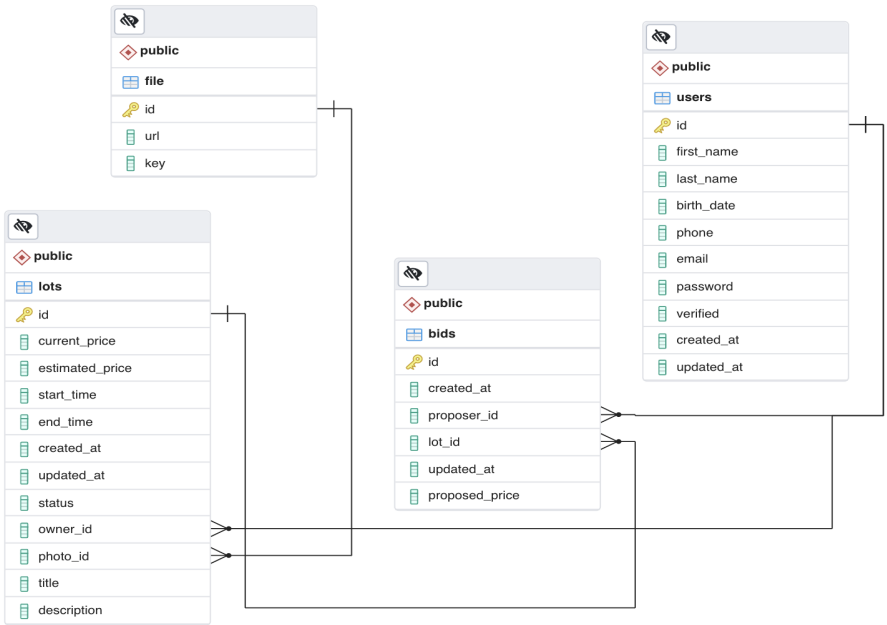

Рисунок 4 − ERD діаграма сутностей

3.2 Файлова система додатку

Ініціюємо проект за допомогою Nest JS CLI. Виконаємо наступну команду:

- nest new project-name

Буде створено каталог проекту, встановлені модулі node і кілька інших шаблонних файлів, а також створено каталог src /, який буде заповнений кількома основними файлами.

На основі визначених сутностей створимо основні модулі:

- nest g module <name>

Крім основних модулів, нам потрібні додаткові модулі для авторизації, листування, токенізації, планування та файлів. Отримаємо наступну структуру:

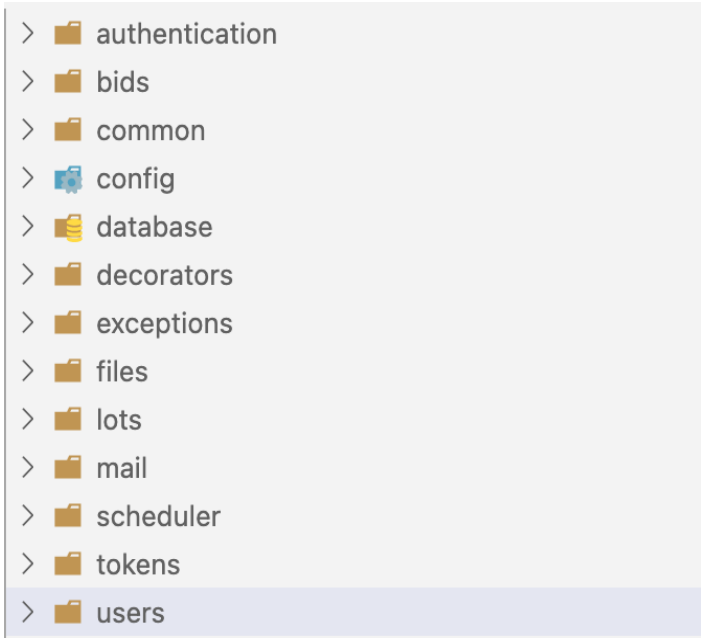

Рисунок 5 − Структура проекту

Ініціюємо сервіси та контролери проекту за допомогою:

- nest g service <name>
- nest g controller <name>

Таблиця 3 − Основні типи файлів проекту

Назва файлу	Роль файлу
<name>.module.ts	Файли модулю
<name>.controller.ts	Файли контроллеру. Відповідний за прийняття запитів та надсилання відповідей клієнту.
<name>.service.ts	Файли сервісу. Відповідають за запити до бази даних, та бізнес логіку.
<name>.controller.spec.ts	Юніт тести для контроллеру. Перевіряють коректність роботи контроллеру
<name>.service.spec.ts	Юніт тести для сервісу. Перевіряють коректність роботи сервісу
<name>.e2e-spec.ts	E2E тести для модулю. Перевіряють кінцеву роботу модулю на рівні з клієнтом.
<name>.exceptions.ts	Файл помилок сутності. Містить всі помилки пов’язані з певною сутністю.
<name>.subscriber.ts	Файл тригерів для операцій з базою даних.
<name>.dto.ts	Файл, який містить дані про сутність, яку передають між модулями, сервісами, чи контролерами.
<name>.validator.ts	Файл, який містить декоратор для перевірки правильності даних.
<name>.entity.ts	Файл сутності бази даних
<name>.config.ts	Файл з налаштуванням певного модулю.
<name>.env	Файл зі змінними оточення.

Крім основних файлів проект також має допоміжні, наприклад файли лінтерів, встановлених пакетів, налаштування проекту та CLI.

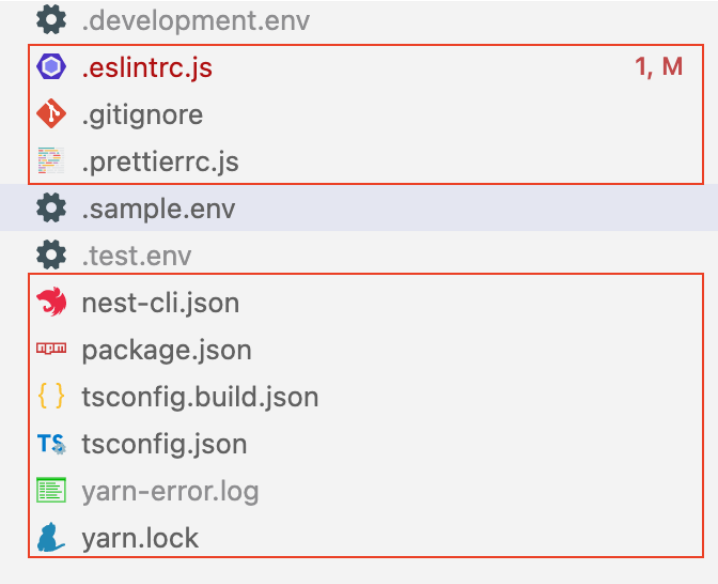

Рисунок 6 – Допоміжні файли проекту

У файлі package.json описані основні скрипти додатку

Таблиця 4 – Основні скрипти додатку

Скрипт	Мета
start	Старт проекту
start:dev	Старт проекту у режимі розробки
start:prod	Старт проекту у режимі продакшену
test	Тестування проекту
test:cov	Перевірка покриття проекту тестами
test:e2e	Запуск E2E тестів

3.3 Тестування додатку

Додаток має 2 типи тестів: E2E тести та юніт-тести. Виконаємо тестування за допомогою скриптів.

```
yarn run v1.22.10
$ jest
 PASS  src/users/users.service.spec.ts (18.673 s)
 PASS  src/bids/bids.service.spec.ts (18.986 s)
 PASS  src/lots/lots.service.spec.ts (19.08 s)
 PASS  src/authentication/authentication.service.spec.ts (19.229 s)
 PASS  src/bids/bids.controller.spec.ts (19.386 s)
 PASS  src/authentication/authentication.controller.spec.ts (19.47 s)
 PASS  src/lots/lots.controller.spec.ts (19.697 s)

Test Suites: 7 passed, 7 total
Tests:       65 passed, 65 total
Snapshots:   0 total
Time:        20.294 s
```

Рисунок 7 – Результат unit-тестування

```
yarn run v1.22.10
$ NODE_ENV=test jest --config ./test/jest-e2e.json --detectOpenHandles
 PASS  test/bids.e2e-spec.ts (9.676 s)
 PASS  test/authentication.e2e-spec.ts
 PASS  test/lots.e2e-spec.ts

Test Suites: 3 passed, 3 total
Tests:       17 passed, 17 total
Snapshots:   0 total
Time:        12.343 s, estimated 13 s
Ran all test suites.
```

Рисунок 8 – Результат E2E-тестування

Перевіримо роботу проекту за допомогою додатку Postman. Відправимо запит на створення користувача.

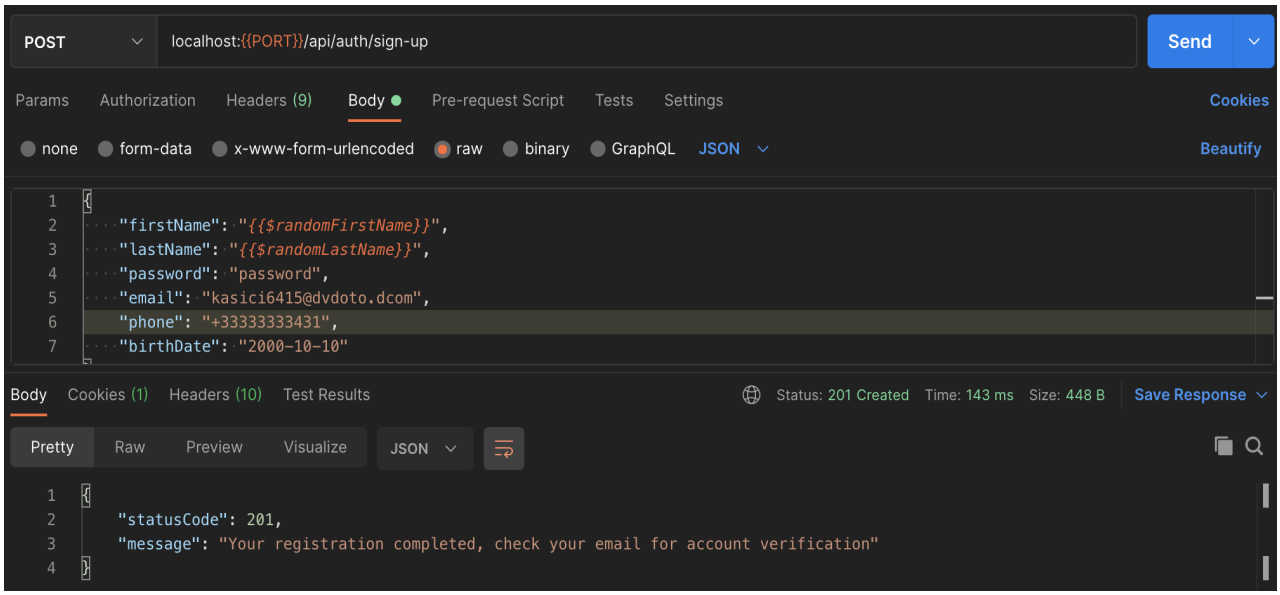

Рисунок 9 – Результат тестування у Postman

Провівши перевірку коректності роботи модулів за допомогою unit-тестування, основного принципу роботи з клієнтом за допомогою E2E-тестування та повноцінної роботи додатку у Postman, можемо зробити висновок, що всі етапи тестування пройдені успішно, тож додаток працює правильно, згідно з поставленою метою.

3.4 Документація проекту

Для документації роутів використаємо Swagger API. В головний файл проекту main.ts внесемо наступні зміни:

```
const swaggerConfig = new DocumentBuilder()
  .setTitle('Auction')
  .setDescription('The auction API description')
  .setVersion('1.0')
  .build();

const document = SwaggerModule.createDocument(app, swaggerConfig);
SwaggerModule.setup('swagger', app, document);
```

Рисунок 10 – Налаштування Swagger API

Після налаштування, перейшовши на url http://localhost:{PORT}/swagger побачимо задокументований список роутів нашого додатку.

За результатом написаних тестів, а також імплементації Swagger API, ми маємо повноцінну документацію проекту, яка спрощує розуміння бізнес-логіки додатку, а також робить розробку клієнтського додатку більш комфортною.

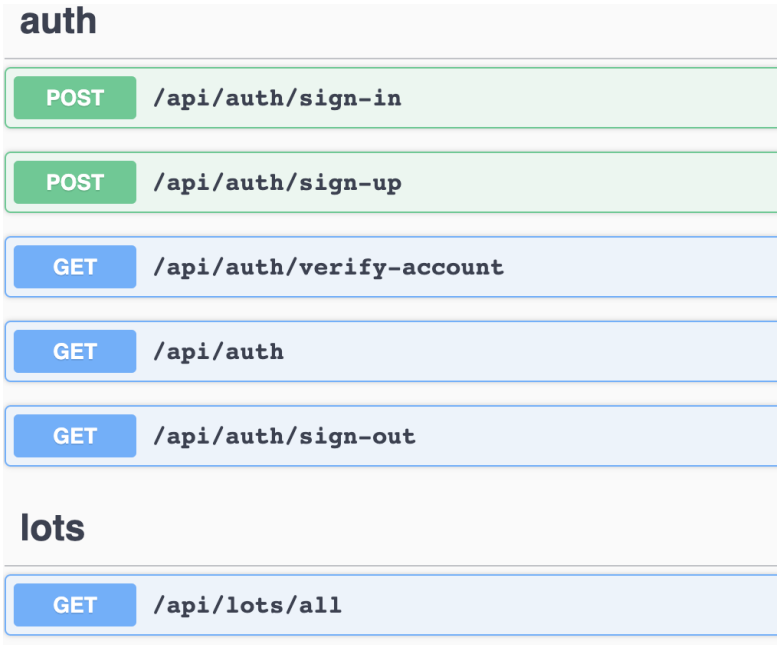

Рисунок 11 – Документація Swagger API

Розроблена документація грає важливу роль у подальшій роботі проекту, спрощуючи його масштабованість та гнучкість.

ВИСНОВКИ

У процесі виконання випускної роботи було розроблено REST API додаток «Розпродаж» з використанням технологій Node JS та PostgreSQL.

Додаток створений за допомогою фреймворку Nest JS та мови програмування Typescript. Додаток покритий E2E та UNIT тестами, також створена документація за допомогою Swagger API. При розробці додатку була використана система контролю версій git, а також система хмарного збереження даних AWS S3.

Реалізований наступний функціонал:

- реєстрація нового користувача;
- підтвердження реєстрації за допомогою пошти;
- авторизація за допомогою JWT-токену;
- можливість створення товару на продаж;
- можливість придбання нового товару;
- завантаження медіафайлів;
- замовлення товару;

Побудовано масштабований REST API проект, який вирішує проблему продажу та купівлі товару. Гнучкий проект дозволить з легкістю використовувати як мобільний додаток, так і повноцінний веб-сайт для купівлі та продажу товару, а переваги REST API та модульної системи Nest JS спрощують форматування додатку для будь-якої форми продажу, будь то спортивні товари, чи високотехнологічні пристрої. Написані тести гарантують коректну роботу, а Swagger API документація спрощує роботу з додатком.

СПИСОК ЛІТЕРАТУРИ

1. REST API [Електронний ресурс] – https://www.redhat.com/en/topics/api/what-is-a-rest-api
2. PHP [Електронний ресурс] – https://www.php.net.
3. Python [Електронний ресурс] – https://www.python.org
4. Java [Електронний ресурс] – https://java.com/en/download/help/whatis_java.html
5. TypeScript [Електронний ресурс] – https://www.altexsoft.com/blog/typescript-pros-and-cons/
6. Adonis.JS [Електронний ресурс] – https://adonisjs.com/
7. ExpressJS [Електронний ресурс] – https://expressjs.com/ru
8. Nest JS [Електронний ресурс] – https://nestjs.com/
9. LoopBack.js [Електронний ресурс] – https://loopback.io/
10. SQL [Електронний ресурс] – https://www.oracle.com/ru/database/what-is-a-relational-database/
11. Yarn [Електронний ресурс] – https://yarnpkg.com/
12. Swagger [Електронний ресурс] – https://swagger.io/
13. Git [Електронний ресурс] – https://git-scm.com/book/ru/v2

ДОДАТОК 1

users.module.ts

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user.entity';
import { UsersService } from './users.service';

@Module({
 imports: [TypeOrmModule.forFeature([User])],
 providers: [UsersService],
 exports: [UsersService],
})
export class UsersModule {}
```

users.service.ts

```
import { Injectable, InternalServerErrorException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { UserExceptions } from '../exceptions';
import { Repository } from 'typeorm';
import { SignUpDto } from '../authentication/dto';
import { User } from './user.entity';

@Injectable()
export class UsersService {
 constructor(
  @InjectRepository(User)
  private userRepository: Repository<User>,
 ) {}

 async createUser(signUpDto: SignUpDto): Promise<User> {
  try {
   return await this.userRepository.save(signUpDto);
```

```
  } catch (error) {
    if (error.code === '23505') {
      throw UserExceptions.emailOrPhoneAlreadyTaken();
    } else {
      throw new InternalServerErrorException(error);
    }
  }
}

async verifyUser(id: number): Promise<User> {
  const user = await this.findUser(id);

  if (user.verified) {
    throw UserExceptions.userAlreadyVerified();
  }

  return await this.userRepository.save({ ...user, verified: true });
}

async findUserByEmail(email: string): Promise<User> {
  try {
    return await this.userRepository.findOneOrFail({ email });
  } catch (err) {
    throw UserExceptions.userNotFoundByEmail();
  }
}

async findUser(id: number): Promise<User> {
  try {
    return await this.userRepository.findOneOrFail(id);
  } catch (err) {
    throw UserExceptions.userNotFound();
  }
 }
}
```

user.entity.ts

```
import { Exclude, Expose } from 'class-transformer';
import { Bid } from '../bids/bid.entity';
import {
 Entity,
 PrimaryGeneratedColumn,
 Column,
 BaseEntity,
 CreateDateColumn,
 UpdateDateColumn,
 OneToMany,
} from 'typeorm';
import { Lot } from '../lots/lot.entity';

@Entity('users')
@Exclude()
export class User extends BaseEntity {
 @PrimaryGeneratedColumn()
 id: number;

 @Expose()
 @Column('varchar', { length: 32 })
 firstName: string;

 @Expose()
 @Column('varchar', { length: 32 })
 lastName: string;

 @Expose()
 @Column('date')
 birthDate: Date;

 @Expose()
 @Column('varchar', { length: 20, unique: true })
 phone: string;
```

```
@Expose()
@Column('varchar', { length: 120, unique: true })
email: string;

@Column('varchar', { length: 120 })
password: string;

@Column({ default: false })
verified: boolean;

@CreateDateColumn()
createdAt: Date;

@UpdateDateColumn()
updatedAt: Date;

@OneToMany(() => Lot, (lot) => lot.owner, { eager: false })
lots: Lot[];

@OneToMany(() => Bid, (bid) => bid.proposer, { eager: false, nullable: true })
bids?: Bid[];

// exposed properties
@Expose()
get fullName(): string {
  return `${this.firstName} ${this.lastName}`;
}

public static of(data: Partial<User>): User {
  const user = new User();
  Object.assign(user, data);

  return user;
}
```

```
}
```

user.subscriber.ts

```
import { EntitySubscriberInterface, EventSubscriber, InsertEvent } from 'typeorm';
import { User } from './user.entity';
import * as bcrypt from 'bcrypt';

@EventSubscriber()
export class UserSubscriber implements EntitySubscriberInterface<User> {
 listenTo() {
  return User;
 }

 async beforeInsert({ entity }: InsertEvent<User>) {
  entity.password = await bcrypt.hash(entity.password, 10);
 }
}
```

ДОДАТОК 2

lots.module.ts

```
import { Module } from '@nestjs/common';
import { LotsService } from './lots.service';
import { LotsController } from './lots.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Lot } from './lot.entity';
import { SchedulerModule } from '../scheduler/scheduler.module';
import { FilesModule } from '../files/files.module';

@Module({
 imports: [TypeOrmModule.forFeature([Lot]), SchedulerModule, FilesModule],
 providers: [LotsService],
 controllers: [LotsController],
 exports: [LotsService],
})
export class LotsModule {}
```

lots.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { CreateLotDto, UpdateLotDto } from './dto';
import { Lot } from './lot.entity';
import { Repository } from 'typeorm';
import { User } from '../users/user.entity';
import { LotExceptions } from '../exceptions';
import { LotStatus } from './lot-status.enum';
import { SchedulerService } from '../scheduler/scheduler.service';
import { DateTime } from 'luxon';
import { PaginationDto } from '../common/dto';
import { FilesService } from '../files/files.service';
import { DataWithTotal } from '../common/types/response.types';

@Injectable()
export class LotsService {
 constructor(
  @InjectRepository(Lot) private readonly lotsRepository: Repository<Lot>,
  private readonly schedulerService: SchedulerService,
  private readonly filesService: FilesService,
 ) {}

 async getProcessingLots({ perPage = 10, page = 0 }: PaginationDto):
Promise<DataWithTotal<Lot>> {
  const [data, total] = await this.lotsRepository.findAndCount({
   where: {
    status: LotStatus.IN_PROCESS,
   },
   skip: page * perPage,
   take: perPage,
   order: { createdAt: 'DESC' },
  });
```

```
    return { data, total };
  }

  async createLot(createLotDto: CreateLotDto, creator: User): Promise<Lot> {
    const createdLot = await this.lotsRepository.save({ ...createLotDto, owner: creator }, {
reload: true });
    this.scheduleLot(createdLot);

    // TODO: next line is hotfix because of save method partial return
    return await this.find(createdLot.id);
  }

  async updateLot(updateLotDto: UpdateLotDto, id: number): Promise<Lot> {
    const lot = await this.find(id);
    const updatedLot = await this.lotsRepository.save({ ...lot, ...updateLotDto });
    this.scheduleLot(updatedLot);

    return updatedLot;
  }

  async updateLotPhoto(id: number, photo: Express.Multer.File): Promise<Lot> {
    const lot = await this.find(id);
    const uploadedPhoto = await this.filesService.uploadFile(photo);

    return await this.lotsRepository.save({ ...lot, photo: uploadedPhoto });
  }

  async deleteLot(id: number): Promise<Lot> {
    const lot = await this.find(id);

    return await this.lotsRepository.remove(lot);
  }

  async find(id: number): Promise<Lot> {
    try {
```

```
      return await this.lotsRepository.findOneOrFail(id);
     } catch (err) {
      throw LotExceptions.lotNotFound();
     }
    }

    async openLot(id: number): Promise<Lot> {
     return await this.lotsRepository.save({ id, status: LotStatus.IN_PROCESS }, { listeners:
false });
    }

    async closeLot(id: number): Promise<Lot> {
     return await this.lotsRepository.save({ id, status: LotStatus.CLOSED }, { listeners: false
});
    }

    async updateLotCurrentPrice(id: number, increasedPrice: number): Promise<Lot> {
     return this.lotsRepository.save({ id, currentPrice: increasedPrice }, { listeners: false });
    }

    scheduleLot(lot: Lot): void {
     const msToStart = DateTime.fromJSDate(lot.startTime).diff(DateTime.now(),
'milliseconds').milliseconds;
     const msToEnd = DateTime.fromJSDate(lot.endTime).diff(DateTime.now(),
'milliseconds').milliseconds;
     this.schedulerService.addTimeout(`lot-${lot.id}-start`, () => this.openLot(lot.id),
msToStart);
     this.schedulerService.addTimeout(`lot-${lot.id}-end`, () => this.closeLot(lot.id),
msToEnd);
    }
   }
```

lots.controller.ts

```
   import {
    Body,
```

```
 Controller,
 Delete,
 Get,
 Param,
 Post,
 Put,
 Query,
 UploadedFile,
 UseGuards,
 UseInterceptors,
} from '@nestjs/common';
import { ApiTags } from '@nestjs/swagger';
import { LotsService } from './lots.service';
import { CreateLotDto, UpdateLotDto } from './dto';
import { AuthGuard } from '@nestjs/passport';
import { GetUser } from '../decorators/get-user.decorator';
import { PaginationDto } from '../common/dto';
import { FileInterceptor } from '@nestjs/platform-express';

@Controller('lots')
@ApiTags('lots')
@UseGuards(AuthGuard('jwt'))
export class LotsController {
 constructor(private readonly lotsService: LotsService) {}

 @Get('/all')
 getAllLots(@Query() paginationDto: PaginationDto) {
  return this.lotsService.getProcessingLots(paginationDto);
 }

 @Post()
 createLot(@Body() createLotDto: CreateLotDto, @GetUser() user) {
  return this.lotsService.createLot(createLotDto, user);
 }
```

```
 @Put(':id')
 updateLot(@Body() updateLotDto: UpdateLotDto, @Param() { id }) {
  return this.lotsService.updateLot(updateLotDto, id);
 }

 @Post(':id/photo')
 @UseInterceptors(FileInterceptor('photo'))
 updateLotPhoto(@Param() { id }, @UploadedFile() photo: Express.Multer.File) {
  return this.lotsService.updateLotPhoto(id, photo);
 }

 @Delete(':id')
 deleteLot(@Param() { id }) {
  return this.lotsService.deleteLot(id);
 }

 @Get(':id')
 getLot(@Param() { id }) {
  return this.lotsService.find(id);
 }
}
```

lot.entity.ts

```
import {
 BaseEntity,
 Column,
 CreateDateColumn,
 Entity,
 JoinColumn,
 ManyToOne,
 OneToMany,
 OneToOne,
 PrimaryGeneratedColumn,
 UpdateDateColumn,
} from 'typeorm';
```

```
import { LotStatus } from './lot-status.enum';
import { User } from '../users/user.entity';
import { Exclude, Expose } from 'class-transformer';
import File from '../files/file.entity';
import { Bid } from '../bids/bid.entity';

@Entity('lots')
@Exclude()
export class Lot extends BaseEntity {
 @Expose()
 @PrimaryGeneratedColumn()
 id: number;

 @Expose()
 @Column('varchar')
 title: string;

 @Expose()
 @Column('varchar')
 description: string;

 @Expose()
 @Column('float')
 currentPrice: number;

 @Expose()
 @Column('float')
 estimatedPrice: number;

 @Column('timestamptz')
 startTime: Date;

 @Column('timestamptz')
 endTime: Date;
```

```
@CreateDateColumn()
createdAt: Date;

@UpdateDateColumn()
updatedAt: Date;

@Column({ type: 'enum', enum: LotStatus, default: LotStatus.PENDING })
status: LotStatus;

@ManyToOne(() => User, (user) => user.lots, { eager: false, nullable: false })
@JoinColumn({ name: 'owner_id' })
owner: User;

@Expose()
@OneToOne(() => File, {
  eager: true,
  nullable: true,
})
@JoinColumn()
public photo?: File;

@Expose()
@OneToMany(() => Bid, (bid) => bid.lot, { eager: false, nullable: true })
bids?: Bid[];

public static of(data: Partial<Lot>): Lot {
  const lot = new Lot();
  Object.assign(lot, data);

  return lot;
 }
}
```

lot.subscriber.ts

```
import { LotExceptions } from '../exceptions';
```

```
import { EntitySubscriberInterface, EventSubscriber, UpdateEvent } from 'typeorm';
import { LotStatus } from './lot-status.enum';
import { Lot } from './lot.entity';

export enum LotAction {
 UPDATE = 'update',
 REMOVE = 'remove',
}

const LotStatusPolicy = {
 [LotAction.UPDATE]: LotStatus.PENDING,
 [LotAction.REMOVE]: LotStatus.PENDING,
};

@EventSubscriber()
export class LotSubscriber implements EntitySubscriberInterface<Lot> {
 listenTo() {
  return Lot;
 }

 beforeUpdate({
  entity,
  queryRunner: {
   data: { action = LotAction.UPDATE },
  },
 }: UpdateEvent<Lot>) {
  if (entity.status !== LotStatusPolicy[action]) {
   throw LotExceptions.lotInvalidStatus(entity.status);
  }
 }

 beforeRemove({
  entity,
  queryRunner: {
   data: { action = LotAction.REMOVE },
```

```
  },
 }: UpdateEvent<Lot>) {
  if (entity.status !== LotStatusPolicy[action]) {
   throw LotExceptions.lotInvalidStatus(entity.status);
  }
 }
}
```

ДОДАТОК 3

bids.module.ts

```
import { Module } from '@nestjs/common';
import { BidsService } from './bids.service';
import { BidsController } from './bids.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Bid } from './bid.entity';
import { LotsModule } from '../lots/lots.module';

@Module({
 imports: [TypeOrmModule.forFeature([Bid]), LotsModule],
 providers: [BidsService],
 controllers: [BidsController],
})
export class BidsModule {}
```

bids.controller.ts

```
import { Body, Controller, Get, Post, Query, UseGuards } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';
import { GetUser } from '../decorators/get-user.decorator';
import { User } from '../users/user.entity';
import { BidsService } from './bids.service';
import { CreateBidDto } from './dto/create-bid.dto';

@Controller('bids')
@UseGuards(AuthGuard('jwt'))
```

```
export class BidsController {
 constructor(private readonly bidsService: BidsService) {}

 @Post()
 createBid(@Body() createBidDto: CreateBidDto, @GetUser() proposer: User) {
  return this.bidsService.createBid(createBidDto, proposer);
 }

 @Get()
 getLotBids(@Query() { lotId }: { lotId: number }) {
  return this.bidsService.getLotBids(lotId);
 }
}
```

bids.service.ts

```
import { DataWithTotal } from './../common/types/response.types';
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from '../users/user.entity';
import { Repository } from 'typeorm';
import { Bid } from './bid.entity';
import { CreateBidDto } from './dto/create-bid.dto';
import { LotsService } from '../lots/lots.service';

@Injectable()
export class BidsService {
 constructor(
  @InjectRepository(Bid) private readonly bidsRepository: Repository<Bid>,
  private readonly lotsService: LotsService,
 ) {}

 async createBid({ lotId, proposedPrice }: CreateBidDto, proposer: User): Promise<Bid> {
  const lot = await this.lotsService.find(lotId);
  const createdBid = await this.bidsRepository.save({ proposedPrice, proposer, lot });
  await this.lotsService.updateLotCurrentPrice(lot.id, proposedPrice);
```

```
    return await this.bidsRepository.findOne(createdBid.id);
  }

  async getLotBids(lotId: number): Promise<DataWithTotal<Bid>> {
    const lot = await this.lotsService.find(lotId);
    const [data, total] = await this.bidsRepository.findAndCount({ where: { lot } });

    return {
      data,
      total,
    };
  }
}
```

bid.entity.ts

```
import { User } from '../users/user.entity';
import { BaseEntity, Column, CreateDateColumn, Entity, JoinColumn, ManyToOne,
PrimaryGeneratedColumn } from 'typeorm';
import { Lot } from '../lots/lot.entity';
import { Exclude, Expose } from 'class-transformer';

@Entity('bids')
@Exclude()
export class Bid extends BaseEntity {
 @Expose()
 @PrimaryGeneratedColumn()
 id: number;

 @Expose()
 @Column({ type: 'float' })
 proposedPrice: number;

 @ManyToOne(() => Lot, (lot) => lot.bids, { eager: false })
 @JoinColumn({ name: 'lot_id' })
 lot: Lot;
```

```
  @ManyToOne(() => User, (user) => user.bids, { eager: false })
  @JoinColumn({ name: 'proposer_id' })
  proposer: User;

  @CreateDateColumn()
  @Expose()
  createdAt: Date;

  public static of(data: Partial<Bid>): Bid {
    const bid = new Bid();
    Object.assign(bid, data);

    return bid;
  }
}
```

bid.subscriber.ts

```
import { LotExceptions } from '../exceptions';
import { EntitySubscriberInterface, EventSubscriber, UpdateEvent } from 'typeorm';
import { Bid } from './bid.entity';
import { BidExceptions } from '../exceptions/bid.exceptions';
import { LotStatus } from '../lots/lot-status.enum';

@EventSubscriber()
export class BidSubscriber implements EntitySubscriberInterface<Bid> {
 listenTo() {
  return Bid;
 }

 beforeInsert({ entity: { lot, proposedPrice } }: UpdateEvent<Bid>) {
  if (lot.status !== LotStatus.IN_PROCESS) {
   throw LotExceptions.lotInvalidStatus(lot.status);
  }

  if (lot && lot.currentPrice >= proposedPrice) {
   throw BidExceptions.lowProposedPrice();
  }
 }
}
```

ДОДАТОК 4

authentication.module.ts

```
import { Module } from '@nestjs/common';
import { UsersModule } from '../users/users.module';
import { AuthenticationController } from './authentication.controller';
import { AuthenticationService } from './authentication.service';
import { MailModule } from '../mail/mail.module';
import { TokensModule } from '../tokens/tokens.module';
```

```
import { PassportModule } from '@nestjs/passport';
import { JwtStrategy } from './jwt.strategy';
import { ConfigModule } from '@nestjs/config';

@Module({
 imports: [UsersModule, MailModule, TokensModule, PassportModule, ConfigModule],
 providers: [AuthenticationService, JwtStrategy],
 controllers: [AuthenticationController],
})
export class AuthenticationModule {}
```

authentication.controller.ts

```
import { Body, Controller, Get, Post, Query, Res, UseGuards } from '@nestjs/common';
import { AuthenticationService } from './authentication.service';
import { ApiTags } from '@nestjs/swagger';
import { SignUpDto, SignInCredentialsDto, AccountVerificationDto } from './dto';
import { Response } from 'express';
import { AuthGuard } from '@nestjs/passport';
import { GetUser } from '../decorators/get-user.decorator';

@Controller('auth')
@ApiTags('auth')
export class AuthenticationController {
 constructor(private readonly authService: AuthenticationService) {}

 @Post('sign-in')
 async signIn(@Body() signInCredentialsDto: SignInCredentialsDto, @Res({ passthrough:
true }) response: Response) {
   const { user, accessToken } = await this.authService.signIn(signInCredentialsDto);
   response.cookie('accessToken', accessToken);

   return user;
 }

 @Post('sign-up')
```

```
 signUp(@Body() signUpDto: SignUpDto) {
  return this.authService.signUp(signUpDto);
 }

 @Get('verify-account')
 async verifyAccount(
  @Query() { verificationToken }: AccountVerificationDto,
  @Res({ passthrough: true }) response: Response,
 ) {
  const { user, accessToken } = await this.authService.verifyAccount(verificationToken);
  response.cookie('accessToken', accessToken);

  return user;
 }

 @Get()
 @UseGuards(AuthGuard('jwt'))
 authenticateUser(@GetUser() user) {
  return user;
 }

 @Get('/sign-out')
 @UseGuards(AuthGuard('jwt'))
 signOut(@Res({ passthrough: true }) response: Response) {
  response.clearCookie('accessToken');

  return;
 }
}
```

authentication.service.ts

```
import { Injectable } from '@nestjs/common';
import { SignUpDto, SignInCredentialsDto } from './dto';
import { UsersService } from '../users/users.service';
import * as bcrypt from 'bcrypt';
```

```
import { MailService } from '../mail/mail.service';
import { TokensService } from '../tokens/tokens.service';
import { UserExceptions } from '../exceptions';
import { User } from '../users/user.entity';

@Injectable()
export class AuthenticationService {
 constructor(
  private readonly usersService: UsersService,
  private readonly mailService: MailService,
  private readonly tokensService: TokensService,
 ) {}

 async signIn({ email, password }: SignInCredentialsDto): Promise<{ user: User;
accessToken: string }> {
  const user = await this.usersService.findUserByEmail(email);
  const isPasswordMatch = await bcrypt.compare(password, user.password);

  if (!isPasswordMatch) {
   throw UserExceptions.emailOrPasswordIncorrect();
  }

  if (!user.verified) {
   throw UserExceptions.userIsNotVerified();
  }

  return {
   user,
   accessToken: this.tokensService.generateUserAccessToken(user),
  };
 }

 async signUp(signUpDto: SignUpDto) {
  const createdUser = await this.usersService.createUser(signUpDto);
```

```
        const verificationToken =
this.tokensService.generateUserVerificationToken(createdUser);
        this.mailService.sendUserVerificationLetter(createdUser, verificationToken);

        return { statusCode: 201, message: 'Your registration completed, check your email for
account verification' };
      }

      async verifyAccount(verificationToken: string) {
        const { id } = this.tokensService.verifyUserVerificationToken(verificationToken);
        const verifiedUser = await this.usersService.verifyUser(id);

        return {
          user: verifiedUser,
          accessToken: this.tokensService.generateUserAccessToken(verifiedUser),
        };
      }
    }
```