МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

ВИПУСКНА РОБОТА

на тему:

«Веб-додаток для магазину музичних інструментів "Waves"»

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

Керівник роботи — **Тиркусова Н.В.**

Студента групи ІН – 64-8 — **Голуб Н.О.**

СУМИ 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2020 р.

ЗАВДАННЯ
до випускної роботи

Студента четвертого курсу, групи ІН-64/8 спеціальності "Інформатика" денної форми навчання Голуба Нікіти Олександровича.

Тема: «**Веб-додаток для магазину музичних інструментів "Waves"**»

Затверджена наказом по СумДУ

№_____ від _____2020 р.

Зміст пояснювальної записки: 1) Інофмаційний огляд; 2) вибір методу рішення; 3) програмна реалізація

Дата видачі завдання "_____"_____2020 р.

Керівник випускної роботи _____Тиркусова Н.В.

Завдання приняв до виконання _____ Голуб Н.О.

РЕФЕРАТ

Записка: 70 стор., 21 рис., 1 додаток, 6 джерел.

Об’єкт дослідження — веб-додаток для магазину музичних інструментів “Waves”.

Мета роботи — розробка веб-додатку для магазину музичних інструментів «Waves».

Методи дослідження — в процесі програмної реалізації проекту було застосовано стек технологій для побудови односторінкових веб-додатків MERN (MongoDB, Express, Node.js, React).

Результати — створено веб-додаток для магазину музичних інструментів “Waves”. Реалізовано повний функціонал для продажу товарів, виставлення їх на аукціон, а також можливість онлайн-оплати. Після завершення тестів веб-додаток був завантажений на хостинг.

ВЕБ-ДОДАТОК ІНТЕРНЕТ-МАГАЗИНУ, WEB, MERN, MONGODB, EXPRESS, REACT, NODE.JS.

ЗМІСТ

ВСТУП

У сьогоденні Інтернет виступає в ролі потужного інструменту з пошуку та надання інформації. За статистикою, більше половини жителів планети мають доступ до мережі Інтернет. Як наслідок, розробка веб-додатку дозволить використовувати сучасні технології для розвитку інформаційної підтримки та реклами. У зв'язку з цим Web-програмування виділяється як самостійна галузь програмування.

Спочатку дана галузь не могла порівнюватись з іншими областями програмування. В результаті розвитку Web-технологій комп'ютер став інструментом інформаційної підтримки. Кількість людей, що використовують Інтернет та знаходять з його допомогою потрібну інформацію, весь час розширюється.

Розробка веб-додатку забезпечить нових користувачів, оскільки добре створений програмний продукт буде досить легко знайти за допомогою пошукових систем. За допомогою сайту можна домогтися швидкого реагування на побажання користувачів, а також вносити відповідні зміни в роботу. Крім цього, веб-додаток дозволить знизити витрати на рекламу. Розроблений із застосуванням сучасних технологій, він являє собою основний інформаційний ресурс, за допомогою якого можна здійснювати:

- передачу всієї необхідної інформації про компанії;
- безпосередній контакт з користувачем, а також інформаційну підтримку клієнта;
- рекламу компанію.

Поява технології динамічних веб-додатів змусила переосмислити роботу людини не тільки з інформацією, але і з комп'ютером в цілому, з огляду на те, що такі властивості обчислювальної техніки, як пропускна здатність, продуктивність процесора, обсяг накопичувальної пам'яті не включали в себе інтерфейс користувача для зручності роботи людини з системою. Тому застосування нових веб-технологій було важким, але в результаті розвитку інтерфейсу взаємодії людини з комп'ютером проявився великий інтерес до можливостей обчислювальної техніки.

Актуальність проекту полягає в тому, що кожного року кількість веб-додатків для магазинів збільшується, так як даний метод торгівлі дійсно вигідний як для продавця, так і для клієнтів. Це дозволяє зекономити час і багато витрат.

Практична значимість. Розробка веб-додатку допоможе зміцнити позиції на традиційних ринках і виходити на нові. За допомогою інтернет-магазину з'явиться можливість збільшити свої продажі, а відповідно грошовий обіг і прибуток. Це допоможе з мінімальними витратами організувати свій власний бізнес і відкрити нові ринки збуту товарів.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Актуальність веб-додатків для магазинів

Багато людей по всьому світу вважають краще за все купувати в Інтернеті та купувати товари декількох марок та компаній, які вони не можуть знайти або які недоступні для придбання у своїх країнах. У наш час за допомогою нових технологій та підтримки Інтернету, люди з усього світу почали купувати товари в Інтернеті, просто сидячи в своїх будинках. Купівля предметів та продуктів через Інтернет - це дуже легке завдання. Зараз він відіграє дуже важливу роль у житті кожного, особливо людей похилого віку, а також людей з дуже напруженим графіком життя. Він забезпечує дуже комфортний сервіс для своїх клієнтів, маючи змогу зберегти товар у особистій корзині, для можливості придбання його згодом. Покупки через Інтернет просто працюють для людей, які мають дійсну кредитну карту, дебетову карту або рахунок в Інтернет-банку.

Прямий продаж в Інтернеті збільшує прибуток. Завдяки інтернет-магазину він більше не обмежується кількістю клієнтів, які можуть фізично відвідувати приміщення магазину. Це надає змогу зняти всі географічні обмеження. Інтернет-магазин також дозволяє обслуговувати покупців, яким зручніше переглядати та купувати в ті часи, коли торгові місця традиційно не відкриті. Інтернет-покупки можуть заощадити час як для покупця, так і для продавця, зменшивши телефонні дзвінки щодо доступності, технічних характеристик, графіку роботи та іншої інформації, яку легко знайти на сторінках компанії та товарів.

Також система електронної комерції забезпечує дані в реальному часі та аналітику щодо продуктів та клієнтів. Можна бачити, як люди взаємодіють із сайтом, які продукти їх цікавлять, що вони залишили у кошику та яка була середня покупка. Це цінні показники, які дозволяють вносити корективи для задоволення потреб клієнтів.

Ще однією перевагою є те, що для утримання інтернет-магазину потрібно всього кілька людей: адміністратор сайту, який буде приймати і обробляти замовлення, що

надійшли, а також вести бухгалтерію і контролювати доставку товару. Надалі штат можна збільшити, додавши посаду бухгалтера та контент-менеджера, який займеться змістом сайту і його рекламою.

1.2 Огляд сучасних рішень для проектуваня та створення бази даних

Окрім частини, що працює з даними, потрібне також і сховище цих даних, а саме – бази даних. Реляційні бази даних, що колись високо цінувалися за можливості запиту та управління транзакціями, тепер витісняються новою парадигмою NoSQL (не тільки SQL), яка орієнтується на їх недоліки масштабованості та великі проблеми з ефективністю даних. Найпопулярнішим представником NoSQL є MongoDB, проект з відкритим кодом та потужною підтримкою.

MongoDB часто порівнюють з MySQL, однією з найпопулярніших реляційних баз даних. Результати показують, що використання денормалізованої моделі для mongoDB дало більшу гнучкість у запитах [1].

На відміну від реляційних баз, MongoDB має модель даних, що документно-орієнтовані, завдяки якій MongoDB працює швидше, володіє найкращою масштабністю, її легше використовувати.

Якщо говорити про MySQL - це перевірена технологія. Зрозуміло, що MySQL використовується великими компаніями більш ніж 15 років. Так як ця СКБД використовує стандарт SQL, є можливість досить простої міграції на інші SQL-бази даних, якщо в цьому є потреба. Існує можливість транзакцій, підтримуються складні запити, включаючи аналітику, і т. д.

З точки зору MongoDB, тут перевагою є те, що у наявний гнучкий JSON-формат документів. Для деяких завдань розробникам це зручніше, ніж витрачати час на додавання колонок в SQL-базах даних. Не потрібно вчити SQL, для деяких це складно. Прості запити рідше створюють проблеми. Якщо подивитися на проблеми продуктивності, в основному вони виникають, коли люди пишуть складні запити з JOIN в купу таблиць і GROUP BY. Якщо такої функціональності в системі немає, то

створити складний запит виходить складніше. Також у MongoDB вбудована досить проста масштабованість з використанням технології шардінга.

Якщо говорити про додатки, де використовується MongoDB, і на чому вони фокусуються - це дуже швидка розробка. Тому що все можна постійно змінювати, не потрібно постійно піклуватися про суворі формати документа.

Другий момент - це схема даних. Тут потрібно розуміти, що у даних завжди є схема, питання лише в тому, де вона реалізується. Можна реалізовувати схему даних у себе в додатку, або ця схема реалізується на рівні бази даних.

Якщо говорити про розподіл переваг і недоліків MySQL і MongoDB з точки зору циклу розробки програми, то їх можна представити наступним чином.

MySQL:

- реляційна структруа потребує більшого планування та контролю;
- дані легко використувувати з різних додатків;
- гнучкість.

MongoDB:

- швидкість розробки;
- немає потреби синхронізувати схему в базі даних та в додатку;
- зрозумілий шлях до масштабування;
- легкі рішення.

Взагалі, моделі даних дуже сильно залежать від додатків та команд розробників. Неможливо сказати, що є кращим вибором завжди.

MySQL - реляційна база даних. Можна підтримувати реляційну базу даних, що легко підтримує зв'язки між таблицями [2]. Нормалізуючи інформацію, ми можемо створити зміну даних, що відбувається атомарно в одному місці. Коли дані денормалізовано, нам не потрібно робити зміни, а також модифікувати кучу документів. Результат - завжди таблиця. З однієї сторони, це просто, з іншої - деякі структури даних не завжди добре входять у рамки таблиці. Існує вірогідність, що з цим може бути незручно працювати.

Але це всё в теорії. Якщо ми говоримо про практичне використання MySQL, ми знаємо, що часто денормалізуємо дані, і інколи для деяких додатків використовуємо щось подібне: зберігаємо JSON, XML або іншу структуру в колонках.

У MongoDB структура даних заснована на документах [1]. Дані багатьох веб-додатків дуже просто відобразити. Результати, як список документів, у яких може бути досконало розроблена структура - більш гнучке рішення.

Наприклад, якщо потрібно зберегти контактний лист із телефона. Зрозуміло, що є у дані, які добре записуються в одну реляційну табличку: прізвище, ім'я та ін. Але що стосується номерів телефонів, електронні адреси, то у однієї людини їх може бути декілька. Якщо зберігати це в доброму реляційному вигляді, то краще за все зберігати дані у окремих таблицях, а потім отримувати їх за допомогою JOIN, що менш зручно, ніж зберігати це в колекції, де знаходяться ієрархічні документи. Варто сказати, що це все в реляційній теорії. Деякі бази даних підтримують масиви. Наприклад, в MySQL підтримується формат JSON.

Продуктивність дуже складно порівняти, тому що при розробці додатків використовуються різні схеми баз даних. Але, MongoDB спочатку було зроблено, щоб добре масштабувати багато вузлів через шардинг, тому ефективності було приділено менше уваги.

Масштабованість в даному контексті - те, наскільки легко нам узяти маленький додаток і масштабувати його на багато мільйонів, можливо, навіть на мільярди користувачів. Масштабованість буває різна. Вона буває середня, у рамках однієї машини, коли потрібно підтримувати додатки середнього розміру, або масштабованість на кластері, коли нас додатки вже дуже великі, коли зрозуміло, що навіть одна найпотужніша машина не впорається.

Також має сенс говорити про те, чи масштабуємо ми читання, запис або об'єм даних. У різних додатках їх пріоритети можуть розрізнятися, але в цілому, якщо додаток дуже великий, зазвичай доводиться працювати з усіма з цих речей.

У MySQL в нових версіях дуже хороша масштабованість у рамках одного вузла для LTP- навантажень. Аналітика або складні запити масштабуються погано, тому що MySQL може використати для одного запиту тільки один потік, що погано [2].

Традиційне читання в MySQL масштабується з реплікацією, запис і розмір даних - через шардинг. Якщо дивитися на великі компанії - Facebook, Twitter - вони усі використовують шардинг. Традиційно шардинг в MySQL використовується вручну. Є деякі фреймворки для цього. Наприклад, Vitess - це фреймворк, який Google використовує для сервісу YouTube. Стандартного рішення для шардинга MySQL не пропонує, часто перехід на шардинг вимагає уваги від розробників.

У MongoDB фокус спочатку був в масштабованості на багатьох вузлах. Навіть у випадках з маленьким застосуванням рекомендується використати шардинг із самого початку.

Адміністрування - це усі ті речі, про які не думають розробники. Принаймні в першу чергу. MySQL досить гнучкий, у нього є багато різних підходів. Але ця безліч варіантів породжує складність.

У MongoDB все більше орієнтовано на те, що все працює одним стандартним чином - є мінімізація адміністрування. Але зрозуміло, що це відбувається при втраті гнучкості.

Підсумовуючи розглянуту вище інформацію, для реалізації було обрано СКБД MongoDB, опираючись на наступні фактори:

- відсутність схеми, ця база даних грунтована на колекціях різних документів;
- кількість полів, зміст і розмір цих документів може відрізнятись, тобто різні сутності не мають бути ідентичні по структурі;
- зрозуміла структура кожного об'єкту;
- легке масштабування;
- зберігання використовуваних в даний момент у внутрішній пам'яті, що дозволяє отримувати швидший доступ;
- дані зберігаються у вигляді JSON документів;
- підтримка динамічних запитів документів;

- відсутність складних JOIN запитів;
- відсутність необхідності маппинга об'єктів додатка в об'єкти БД.

Для зручного представлення та конструювання бази даних було використано графічний клієнт Robo3T, інтерфейс якого зображений на рисунку 1.1.

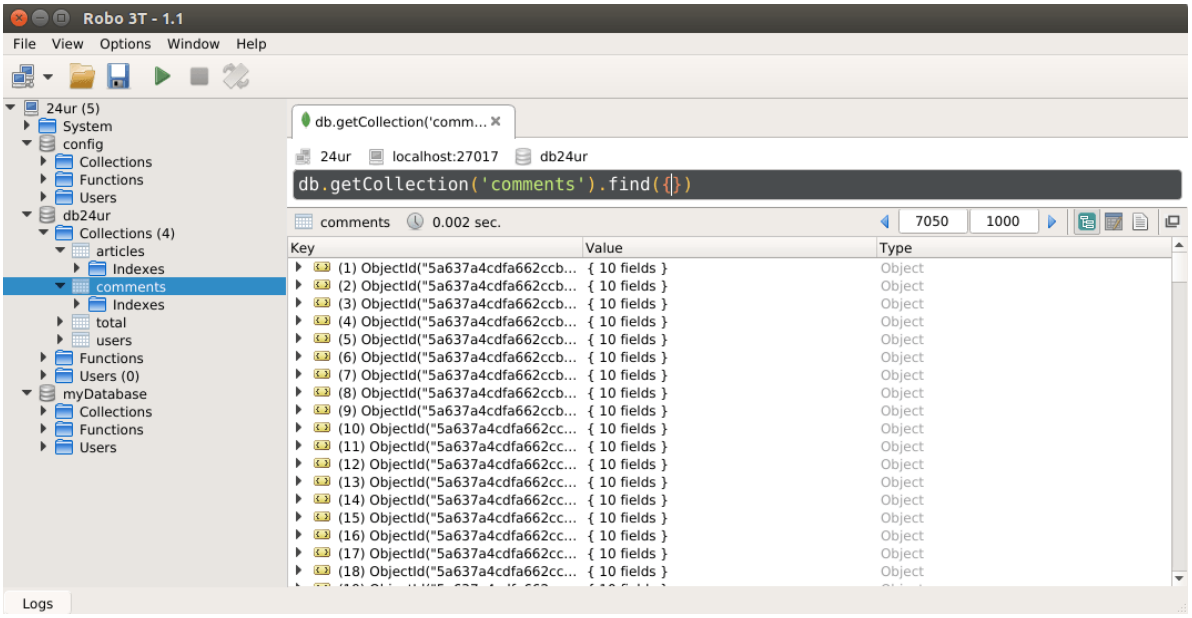

Рисунок 1.1 – Інтерфейс графічного клієнта Robo3T

Він надає багато можливостей, що поліпшують роботу з базою даних, серед яких:

- автозаповнення запитів за допомогою IntelliShell;
- використання SQL для запиту MongoDB;
- створення запитів агрегування поетапно;
- кілька способів імпорту та експорту, включаючи міграцію SQL.

1.3 Постановка задачі

«Waves» являє собою магазин в сфері продажу музичних інструментів для індивідуальних споживачів. В ході розширення, керівництво визнало потрібним створення інтернет-магазину для залучення більшої кількості споживачів (враховуючи споживачів з інших міст країни).

При розробці концепту сайту, було видвинуто наступні вимоги:

- надання інформації щодо товару;
- наявність горизонтального меню;
- наявність модуля здійснення пошукових запитів;
- навігація по сайту забезпечується за допомогою посилань на його розділи;
- графічний інтерфейс користувача повинен забезпечувати інтуїтивно зрозуміле уявлення структури розміщеної на ньому інформації, забезпечувати швидкий перехід до вибраного розділу сайту;
- меню навігації має бути зрозумілим для користувача, а саме: гіперпосилання на сторінки повинні мати зрозумілі заголовки, щоб користувач міг перейти до потрібної йому сторінки;
- додаток повинен підтримувати можливості адміністрування (додавання, видалення та редагування товарів, категорій і т.д.);
- графічні елементи сайту повинні бути розроблені з урахуванням специфіки подання інформації;
- основні елементи меню та інші елементи навігації повинні бути доступні з головної сторінки сайту;
- дизайн сайту повинен бути сучасним і лаконічним;
- відсутність елементів, які відволікали б увагу користувачів від перегляду інформації;
- відсутність складно структурованої інформації, важкої для сприйняття;
- відсутність надлишку інформації на сторінках сайту;
- користувачі додатку мають можливість реєстрації для отримання додаткових можливостей його використання;
- товари сайту можна додавати в кошик, замовляти та оплачувати їх;
- можливість створення аукціонів для товарів.

2 ВИБІР МЕТОДУ РІШЕННЯ

2.1 Стек технологій для розробки

Будь-який веб-додаток побудований за допомогою декількох технологій. Комбінації цих технологій називаються терміном "стек", популяризованим стеком LAMP, який є абревіатурою від Linux, Apache, MySQL, PHP, які всі являють собою програмне забезпечення з відкритим початковим кодом. По мірі розвитку веб-додатків і їх інтерактивності на перший план вийшов односторінковий застосунок (SPA).

SPA - парадигма веб-додатків, що дозволяє уникнути отримання даних всієї веб-сторінки з сервера для відображення нового вмісту. [2] Натомість він використовує легкі запити до сервера, щоб отримати деякі дані або фрагменти та змінити веб-сторінку. Результат виглядає досить чудово порівняно зі старим способом, що повністю перезавантажував сторінку. Це призвело до зростання фреймворків для створення клієнтської частини веб-додатків, оскільки на ній почало виконуватися багато коду.

Приблизно в той же час NoSQL бази даних також почали набирати популярність. Стек MEAN (MongoDB, Express, AngularJS, Node.js) був одним із ранніх стеків з відкритим кодом, що уособлював перехід до SPA та прийняття NoSQL. AngularJS, фреймворк на основі MVC, MongoDB, дуже популярна NoSQL база даних, що була використана для постійного зберігання інформації, Node.js, середовище виконання мови JavaScript на сервері та Express, веб-сервер, побудований на Node.js, утворили один з найпопулярніших стеків цього періоду [3].

Але згодом не зовсім конкуруюча на той час альтернативна технологія React для розробки front-end, створена Facebook, почала набувати популярності і запропонувала альтернативу AngularJS. Таким чином утворився стек MERN (MongoDB, Express, React, Node.js).

2.2 Опис обраних технологій для розробки frontend

Для розробки клієнтської частини додатку було обрано React. Це JavaScript технологія з відкритим кодом, що є компонентно-орієнтованою і підтримується компанією Facebook [4]. На відміну від AngularJS, React не є фреймворком. Це бібліотека. Таким чином, вона сама по собі не створює шаблонних рамок, таких як MVC. React використовується для відображення виду, але як зв'язати решту програми разом, залежить тільки від розробника.

Є і багато інших компаній, окрім самого Facebook, які використовують React у виробництві, наприклад Airbnb, Atlassian, Bitbucket, Disqus, Walmart, VK тощо. Розробники з Facebook створили React для власного використання, а згодом зробили його проектом з відкритим кодом. Однією з її характерних особливостей бібліотеки є можливість використовувати JSX, мову з близьким до HTML синтаксисом, яка компілюється в JavaScript.

Фундаментальний будівельний блок в React - це компонент, який підтримує свій власний стан. Можна сказати, що у React все, робиться розробником, - це складання компонентів. Компоненти складаються, щоб зробити інший компонент, які в подальшому формують повний вигляд або сторінку. Компонент інкапсулює стан даних, що полегшує написання додатку шляхом його поділу на маленькі частини, що надають можливість зосереджуватись на чомусь одному.

Створені компоненти можуть бути з легкістю змінені і використані заново в нових проектах. Високий відсоток перевикористання коду підвищує ефективність тестів, що, в свою чергу, призводить до більш високого рівня контролю якості.

Використовуючи React розробники можуть досягти більшої продуктивності додатків за допомогою Virtual DOM, що надає змогу створювати ізоморфні додатки, які допоможуть позбавитися від неприємної ситуації, коли користувач з нетерпінням чекає, коли завершиться завантаження даних.

Document Object Model, або DOM, - це спосіб представлення та взаємодії з об'єктами в HTML, XHTML і XML документах [5]. Відповідно до цієї моделі, кожен такий документ являє собою ієрархічне дерево елементів, зване DOM-деревом.

Використовуючи спеціальні методи, можна отримати доступ до певних елементів документа і змінювати їх так, як потрібно. Але коли ми створюється динамічна інтерактивна веб-сторінка, потрібно, щоб DOM оновлювався так швидко, як це можливо.

Для даної задачі деякі фреймворки використовують прийом, який називається «dirty checking» і полягає в регулярній перевці стану документа і перевірці змін в структурі даних. Подібна задача складно релізується у високонавантажених додатках.

Virtual DOM, в свою чергу, зберігається в пам'яті. Саме тому в момент, коли «справжній» DOM змінюється, React може змінювати Virtual DOM в одну мить. React «збирає» такі зміни, порівнює їх зі станом DOM, а потім змінює компоненти. При цьому підході не робиться регулярне оновлення DOM. Саме тому може бути досягнута більш висока продуктивність таких додатків.

Також разом з React буде використано Redux - менеджер станів. Хоча в React є власний метод управління станами, він погано масштабується. Переміщення стану вгору по дереву працює для простих додатків, але в більш складних архітектурах зміна стану проводиться через властивості (props). Ще краще робити це через зовнішнє глобальне сховище.

Redux - це спосіб управління станом додатку. Він заснований на декількох концепціях, вивчивши які, можна з легкістю вирішувати проблеми зі станом. Слід зазначити, що Redux ідеально підходить для середніх і великих додатків. Їм варто користуватися тільки у випадках, коли неможливо управляти станом додатку за допомогою стандартного менеджера станів. Простим додаткам Redux не потрібен.

2.3 Опис обраних технологій для розробки backend

На сьогоднішній день існує багато рішень для розробки серверної частини веб-додатків. Найпопулярнішими з них є Django, Laravel, Yii2, Ruby On Rails, ASP.Net та Express для Node.js. Для розробки проекту було обрано Node.js Та фреймворк Express.

Node.js — платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript. Простіше кажучи, Node.js - це

JavaScript поза браузером. Розробники Node.js просто взяли движок ChromeV8 і змусили його працювати незалежно як середовище виконання для JavaScript.

У браузері можливо завантажити кілька файлів JavaScript, але для цього потрібна HTML-сторінка. Неможливо зв'язати файл JavaScript з іншим таким файлом. Але для Node.js не існує сторінки HTML, яка б запускала все це.

За відсутності вкладеної HTML-сторінки, Node.js використовує власну модульну систему на основі CommonJS, зв'язуючи разом кілька файлів JavaScript. Вони називаються модулями.

Модулі схожі на бібліотеки, тому ви можете розділити свій код на файли або модулі заради кращої організації та завантаження. Слід зазначити, що порівняно з JavaScript у браузері, існує більш чистий спосіб модуляції коду за допомогою Node.js.

Node.js постачається з купою основних модулів, зібраних у двійковий файл. Ці модулі забезпечують доступ до елементів операційної системи, таких як файлова система, мережа, введення та виведення тощо. Вони також надають деякі функціональні утиліти, які зазвичай потрібні більшості програм. Окрім власних файлів та основних модулів, можна також знайти велику кількість сторонніх бібліотек за допомогою npm.

Npm - менеджер пакетів за замовчуванням для Node.js. Він надає змогу встановлення та використовувати сторонніх бібліотек (пакетів, модулів) і управляти залежностями між ними.

Хоча npm створювався як сховище для модулів Node.js, він швидко трансформувався в пакетний менеджер для доставки інших модулів на базі JavaScript, зокрема, тих, які можна використовувати в браузері.

Навіть незважаючи на те, що React може бути включений безпосередньо у ваш HTML як файл сценарію, замість цього рекомендується встановити React через npm. Але як тільки він буде встановлений як пакет, потрібно з’єднати весь код, щоб включити його до HTML для відкриття доступу для браузеру. Для цього існують такі інструменти, як Browrify або Webpack, які можуть зібрати модулі.

Node.js має асинхронну модель, керовану подіями, що не блокує введення і виведення, на відміну від використання потоків для досягнення багатозадачності. Більшість інших мов залежать від потоків, щоб підтримувати декілька проесів одночасно. Але насправді немає такого поняття, як одночасність, коли мова йде про єдиний процесор, на якому виконується код. Потоки створюють відчуття одночасності, дозволяючи фрагментам коду виконуватись, поки інші чекають завершення події. Зазвичай це події вводу та виводу, такі як читання з файлу або спілкування по мережі.

Node.js, не має потоків. Він заснований на тому, що називається «callback» - виклик функції іншою функцією.

Програмування, кероване подіями, є природним для Node.js через основні мовні конструкції JavaScript. Node.js досягає багатозадачності, використовуючи цикл подій.

Node.js - це просто середовище виконання, яке може запускати JavaScript. Написати повноцінний веб-сервер вручну на Node.js безпосередньо не просто, і це не потрібно. Express - це фреймворк, який спрощує завдання написання коду сервера. Структура Express дозволяє визначати маршрути, специфікації, час надходження запиту HTTP, що відповідає певній схемі.

Express аналізує URL-адресу запити, заголовки та параметри запиту. З боку відповіді, як очікується, він має всі функції, необхідні веб-додаткам. Сюди входить визначення кодів відповідей, налаштування файлів cookie, надсилання користувацьких заголовків тощо. Крім того, можна писати проміжне програмне забезпечення Express, спеціальні фрагменти коду, які можна вставити в будь-який шлях обробки запиту та відповіді, щоб досягти загальних функціональних можливостей, таких як реєстрація, аутентифікація і т.д.

Фреймворк підтримує будь-який механізм шаблону на вибір. Але для SPA використовувати механізм шаблону на стороні сервера не потрібно. Це відбувається тому, що все динамічне генерування контенту здійснюється на клієнтській частині, а веб-сервер обслуговує лише статичні файли та дані через API. Підводячи підсумок, Express - це система веб-сервера, призначена для Node.js [3]. Він не сильно

відрізняється від багатьох інших веб-серверів з точки зору того, що можливо реалізувати з його допомогою.

2.4 Інші бібліотеки та інструменти

Важко створити будь-яку веб-програму, не використовуючи додаткових інструментів, що полегшать розробку та дозволять зекономити час.

React дає лише можливість відображення і допомагає керувати взаємодіями в одному компоненті. Але для синхронізації URL-адреси браузера з поточним станом представлення даних, нам щось більше.

Ця можливість управління URL-адресами та історією називається маршрутизацією. Це схоже на маршрутизацію на стороні сервера, що робить Express: URL-адреса аналізується і на основі її компонентів викноується фрагмент коду пов'язаний з URL-адресою.

React-Router не тільки робить це, але й управляє функцією кнопки «Назад» у браузері, щоб користувач мав змогу використання цієї функції, не завантажуючи всю сторінку з сервера. Можливо реалізувати це самостійно, але React-Router - дуже проста у користуванні бібліотека, яка зекономить час.

Bootstrap, найпопулярніший фреймворк CSS, був адаптований для React, і проект отримав назву ReactBootstrap. Ця бібліотека надає не тільки більшість функцій Bootstrap, такі як компоненти та віджети, а й інформацію про те, як все це власноруч.

Існують інші бібліотеки компонентів, створені для React (Material-UI, MUI, Elemental UI тощо), а також окремі компоненти (react-select, react-treeview, and react-date-picker). Все це теж хороший вибір, залежно від того, які стоять вимоги. Але React-Bootstrap - це найповніша бібліотека, що існує на сьогодні.

Webpack – незамінний інструмент, коли мова йде про модульний код. Є й інші конкуруючі інструменти, такі як Bower і Browrify, які також служать для групування всього коду, але Webpack простіший у використанні і не потребує інших інструментів (наприклад, gulp або grunt) для управління процесом роботи.

Visual Studio Code — текстовий редактор, розроблений компанією Microsoft. Він безкоштовно розповсюджується і доступний у версіях для платформ Windows, Linux і MacOS. [6]

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Реалізація back-end

Програмну реалізацію серверної частини сайту можна розділити на дві частини: розробка та створення структури бази даних сайту і розробка API для клієнта.

При розробці структури бази даних було виділено п’ять сутностей, які фігурують при роботі даного інтернет-магазину: продукти, категорії, замовлення, аукціони та користувачі. При створенні самої бази даних, дані сутності було перенесено до наступних колекцій:

- products – колекція, яка зберігає у полях вкладених в неї об’єктів всі можливі дані про товари, а саме: ідентифікатор товару (_id), зображення, що відповідає товару (image), назву товару (name), опис товару (description), категорію, до якої він відноситься (category), наявна кількість товару в магазині (quantity), ціну товару (price), дату створення товару (created), дату оновлення товару (updated);

- orders – колекція, яка зберігає у полях вкладених в неї об’єктів всі можливі дані про замовлення, а саме: ідентифікатор замовлення (_id), ідентифікатор замовленого товару (product_id), електронну адресу покупця (customer_email), адресу доставки (delivery_address), ідентифікатор чеку про оплату товару (payment_id), ідентифікатор користувача, що зробив замовлення (user);

- shops – колекція, яка зберігає у полях вкладених в неї об’єктів всі можливі дані про категорії, а саме: ідентифікатор категорії (_id), зображення, що відповідає категорії (image), назву категорії (name), опис категорії (description), дату створення категорії (created), дату оновлення категорії (updated);

- users - колекція, яка зберігає у полях вкладених в неї об’єктів дані про користувачів, а саме: ідентифікатор користувача (_id), чи є користувач продавцем (seller), ім'я користувача (name), електронна пошта користувача (email), зашифрований пароль користувача (hashed password), дата створення аккаунта (created), дата оновлення аккаунта (updated).

- auctions - колекція, яка зберігає у полях вкладених в неї об’єктів дані про користувачів, а саме: ідентифікатор аукціону (_id), назву аукціону (itemName), зображення, що відповідає аукціону (image), дата оновлення аукціону (updated), дата початку аукціону (bidStart), дата закінчення аукціону (bidEnd), початкова ціна (startingBid).

При розробці самого сервера, вихідні файли було розсортовано у різні директорії, відповідно до їх функціоналу. Структура проекту зображена на рисунку 3.1.

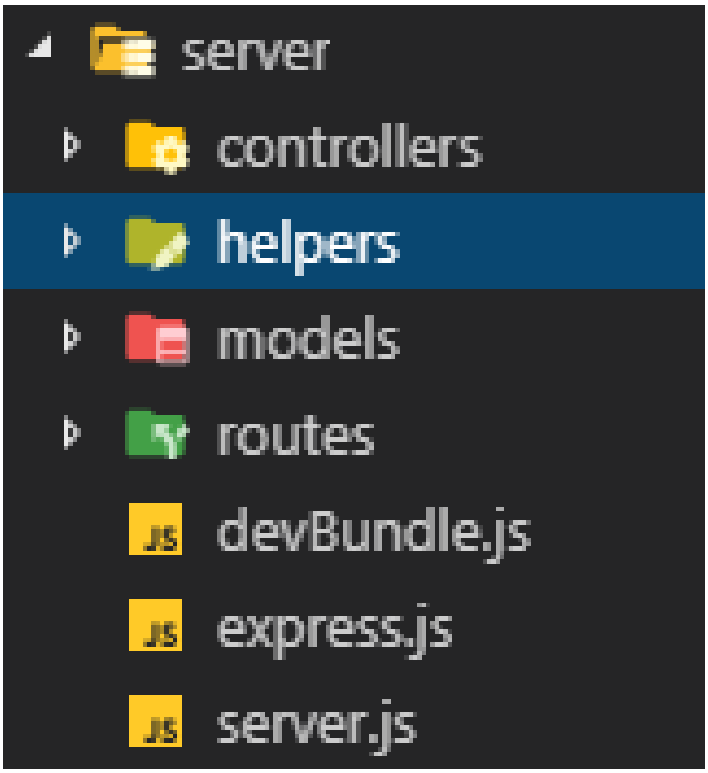

Рисунок 3.1 – Структура back-end

В кореневому каталозі знаходяться 3 модулі:

- devBundle.js - відповідає за налаштування модулю зборки проектів Webpack;

- express.js - відповідає за конфігурацію сервера та використані залежності;

- server.js – файл, у якому описане підключення до бази даних та який об’єднує всі вихідні файли.

Також у back-end частині проекту наявно 4 каталоги.

Controllers - зберігає модулі, функціонал яких відповідає за обробку запитів від клієнта. Його вміст:

- auction.controller.js – відповідає за обробку запитів, зв’язаних з аукціонами;
- auth.controller.js – відповідає за аутентифікацію користувачів;
- bidding.controller.js - відповідає за обробку запитів, зв’язаних з цінами товарів на аукціонах;
- order.controller.js - відповідає за обробку запитів, зв’язаних з замовленнями товарів;
- product.controller.js – відповідає за обробку запитів, зв’язаних з товарами;
- shop.controller.js - відповідає за обробку запитів, зв’язаних з категоріями;
- user.controller.js - відповідає за обробку запитів, зв’язаних з користувачами.

Helpers - зберігає файли, функціонал яких відповідає за обробку всіх можливих помилок при роботі сайту. В даному випадку в директорії наявний один файл:

- dbErrorHandler.js – модуль обробки помилок при роботі з БД.

Models - зберігає модулі, які описують моделі сутностей, які використані в базі даних:

- auction.model.js – опис сутності аукціону;
- order.model.js – опис сутності замовлення;
- product.model.js – опис сутності товару;
- shop.model.js – опис сутності категорії;
- user.model.js – опис сутності користувача.

Routes - зберігає модулі, які описують увесь API, яким можна користуватися:

- auction.routes.js – опис API для роботи з аукціонами;
- auth.routes.js - опис API для аутентифікації користувачів;
- order.routes.js – опис API для роботи з замовленнями;
- product.routes.js – опис API для роботи з товарами;
- shop.routes.js – опис API для роботи з категоріями;

- user.routes.js – опис API для роботи з користувачами.

Повний програмний код back-end чаастини надено в Додатку.

3.2 Реалізація front-end

Розробку front-end сайта можна розділити на два етапи. В рамках першого етапу релізовано основну логіку – запити до сервера для отримання потрібних даних для подальшого відображення користувачу.

Другий етап розглядається як створення інтерфейсу сайта та функціоналу, що дозволить користувачу працювати з ним.

Для збільшення продуктивності та полегшення розробки при створенні сайту було використано додаткові модулі Node.js. Список модулів зображено на рисунку 3.2.

```
"dependencies": {
  "@hot-loader/react-dom": "16.13.0"
  "@material-ui/core": "4.9.8",
  "@material-ui/icons": "4.9.1",
  "body-parser": "1.19.0",
  "compression": "1.7.4",
  "cookie-parser": "1.4.5",
  "cors": "2.8.5",
  "express": "4.17.1",
  "express-jwt": "5.3.1",
  "formidable": "1.2.2",
  "helmet": "3.22.0",
  "jsonwebtoken": "8.5.1",
  "lodash": "4.17.15",
  "mongoose": "5.9.7",
  "query-string": "6.11.1",
  "react": "16.13.1",
  "react-dom": "16.13.1",
  "react-hot-loader": "4.12.20",
  "react-router": "5.1.2",
  "react-router-dom": "5.1.2",
  "react-stripe-elements": "6.1.1",
  "request": "2.88.2",
  "socket.io": "2.3.0",
  "socket.io-client": "2.3.0",
  "stripe": "8.38.0"
}
```

Рисунок 3.2 – Використні модулі Node.js

Для структурованості проекту, частини коду були виокремлені в папки з файлами, що відповідають за різні блоки сайту. Структура проекту зображена на рисунку 3.3.

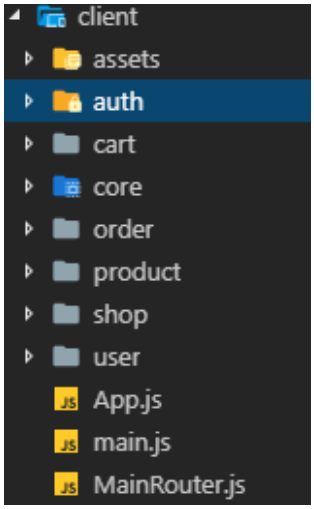

Рисунок 3.3 – Структура front-end

В кореневому каталозі знаходиться три модулі:

- App.js - відповідає за об’єднання всіх вихідних файлів;
- main.js – структура сайту за замовчуванням;
- MainRouter.js - відповідає за маршрутизацію в проекті.

Також у структурі наявні наступні папки:

- assets – у ній знаходяться додаткові іконки для відображення;
- auction – у ній знаходяться модулі, що відповідають за блок аукціонів;
- cart – у ній знаходяться модулі, що відповідають за блок кошику;
- core – у ній знаходяться модулі, що відповідають за домашню сторінку та меню сайту;
- order – у ній знаходяться модулі, що відповідають за блок оформлення замофлення на сайті;
- product – у ній знаходяться модулі, що відповідають за блок товарів на сайті;
- shop – у ній знаходяться модулі, що відповідають за блок категорій сайту;
- user – у ній знаходяться модулі, що відповідають за блок користувачів.

Повний програмний код front-end чаастини надено в Додатку.

3.3 Опис функцій проекту

Однією з найважливіших функцій є авторизація користувачів. Всю роботу по створенню сесії бере на себе модуль connect. Для цього потрібно додати два правила:

```
app.use(connect.cookieParser());
app.use(connect.session({ secret: 'your secret here'} ));
```

Перше правило забезпечує роботу з cookie в цілому. Друге додає до звичайного request поле session, через яке буде доступ до сесії.

Connect.session отримує такі параметри:

- secret - фраза, яка використовується для шифрування інформації в cookies;
- store - об'єкт, який буде використовуватися для зберігання даних сесії, за замовчуванням connect зберігає всі дані в пам'яті;
- cookie - набір параметрів cookie, найважливіший - maxAge, час життя в мілісекундах.

Як вже було зазначено, connect буде додавати поле session до кожного запиту, але за замовчуванням там немає важливої інформації. Якщо користувач введе правильний пароль, треба додати власну інформацію про нього до сесії:

```
if
((request.body.login===user.byId().login)&&(hash(request.body.password)===
user.byId().hashedPassword)) {
    request.session.authorized = true;
    request.session.username = request.body.login;
}
```

Коли користувач захоче разлогінитися, досить буде просто видалити додані поля:

```
delete req.session.authorized;
delete req.session.username ;
```

Для контроля доступу найбільш очевидне рішення - перевіряти request.session.authorized щоразу, коли потрібно згенерувати захищену сторінку. Але

краще створити спеціальне правило, яке перевірятиме права користувача і, перенаправляти його на сторінку помилки в разі її виникнення:

```
// адреси, що підтримує додаток;
Const siteUrls = [
  {pattern:'^/login/?$', restricted: false}
, {pattern:'^/logout/?$', restricted: true}
, {pattern:'^/$', restricted: false}
, {pattern:'^/single/\\w+/?$', restricted: true}
];

function authorizeUrls(urls) {
  function authorize(req, res, next) {
    var requestedUrl = url.parse(req.url).pathname;
    for (var ui in urls) {
      var pattern = urls[ui].pattern;
      var restricted = urls[ui].restricted;
      if (requestedUrl.match(pattern)) {
        if (restricted) {
          if (req.session.authorized) {
            // якщо все добре, переходимо до наступних правил
            next();
            return;
          }
          else{
            // якщо користува не авторизований, то переправляємо його на
сторінку логіну
            res.writeHead(303, {'Location': '/login'});
            res.end();
            return;
          }
        }
        else {
          next();
          return;
        }
      }
    }
    // посилання, якщо в циклі не знайшлося збігів
    console.log('common 404 for ', req.url);
    res.end('404: there is no ' + req.url + ' here');
  }
  return authorize ;
}
app.use('/', authorizeUrls(siteUrls));
```

Іншні розроблені функції наведено в Додатку.

3.4 Інструкція по роботі з програмним продуктом

Відкривши веб-додаток користувач побачить перед собою головну сторінку, що зображена на рисунку 3.4.

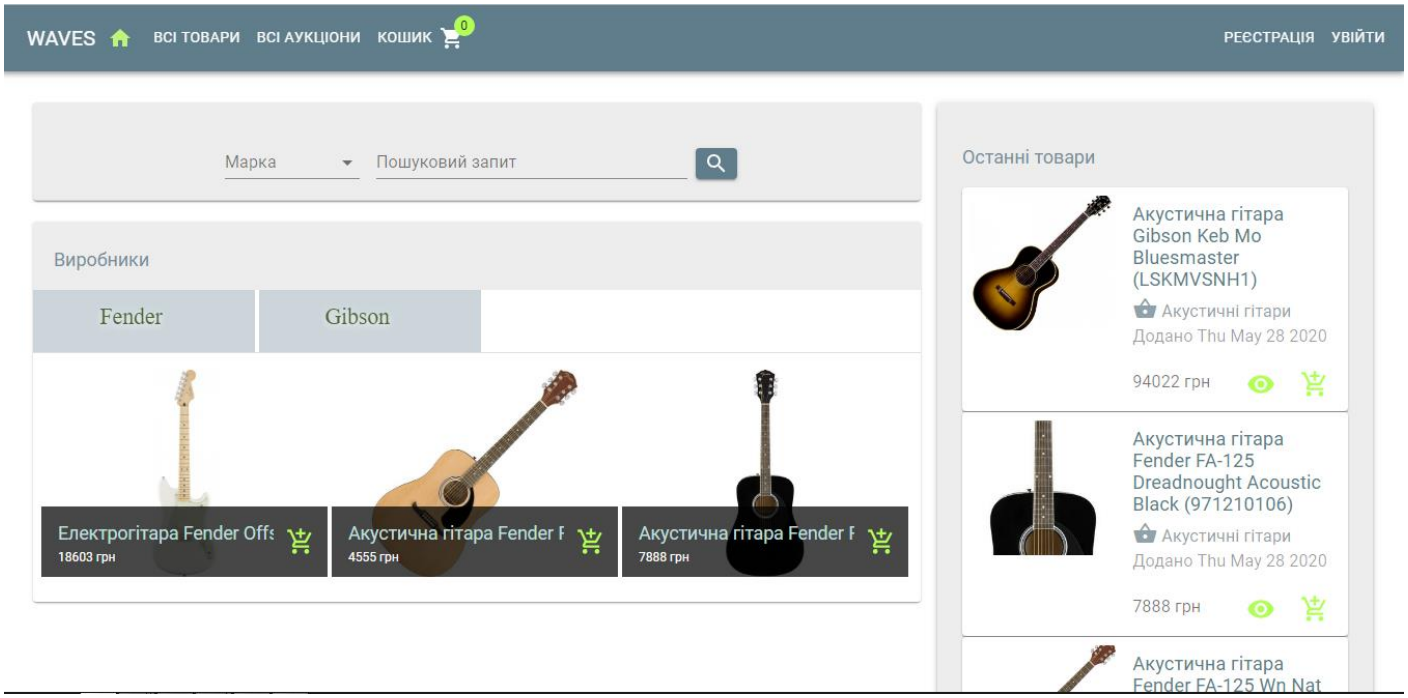

Рисунок 3.4 – Головна сторінка додатку

У верхньому меню наявні функції переходу на сторігки всіх товарів, всіх аукціонів, перехід до кошику, перехід на форми входу та реєстрації. Нижче меню знаходиться панель для пошуку інструментів, панель сортування інструментів по виробника та панель, що відображає останні додані на сайт товари. При натисканні на назву виробника на панелі «Виробники» відбудеться сортування, результат якого проілюстровано на рисунку 3.5

У полі пошуку можна обрати марку товару та здійснити пошук, викоорисовуючи поле вводу. Після натиску на кнопку пошуку вілбудеится пошук, результат якого зображено на рисунку 3.6.

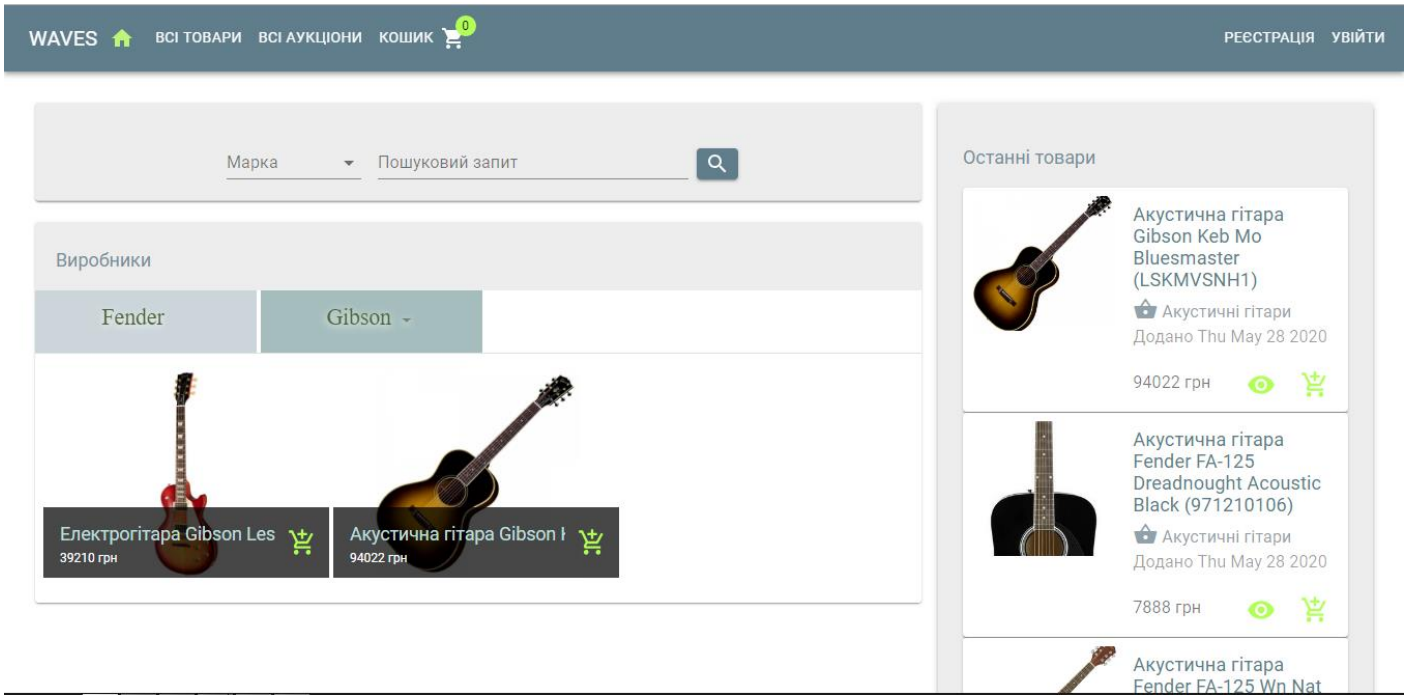

Рисунок 3.5 – Сортування інтрументів за виробником

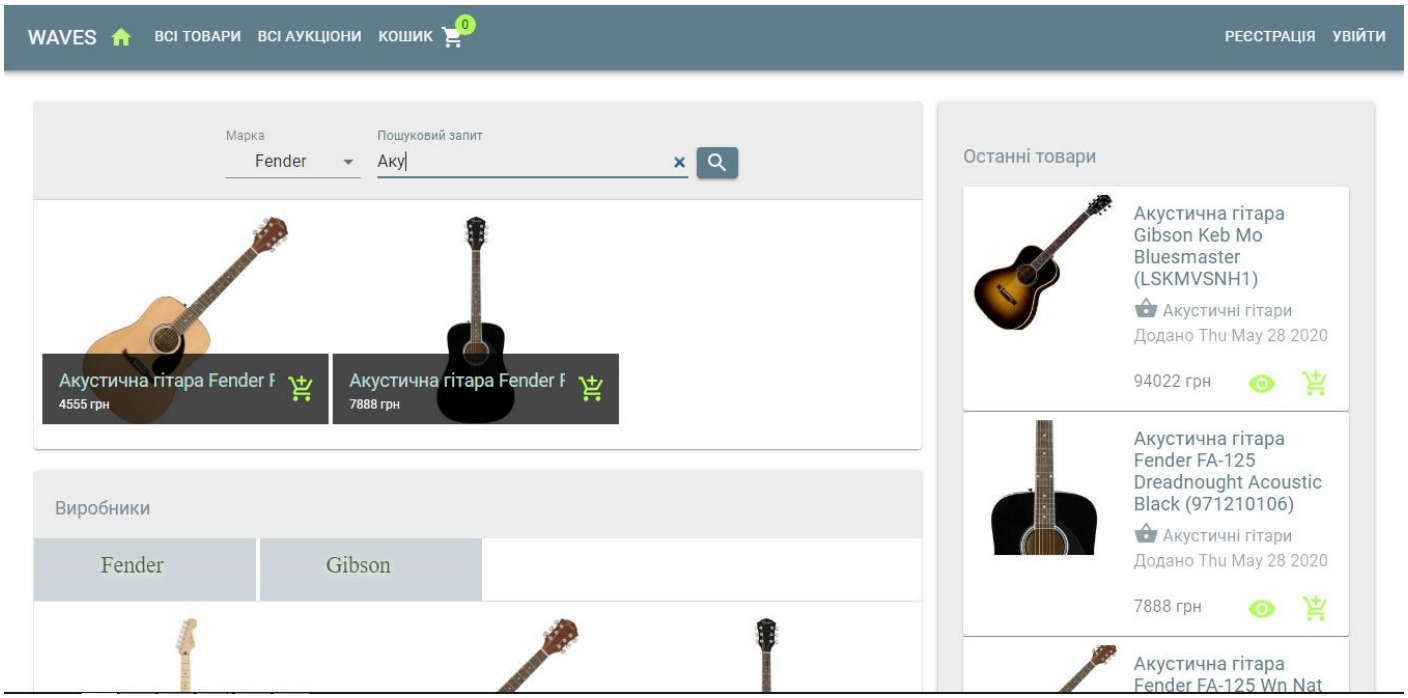

Рисунок 3.6 – Рещультат виконання пощуку товарів

Після натискання у верхньомк меню «Всі товари» користувачу буде відкрито список категорій, що зображено на рисунку 3.7

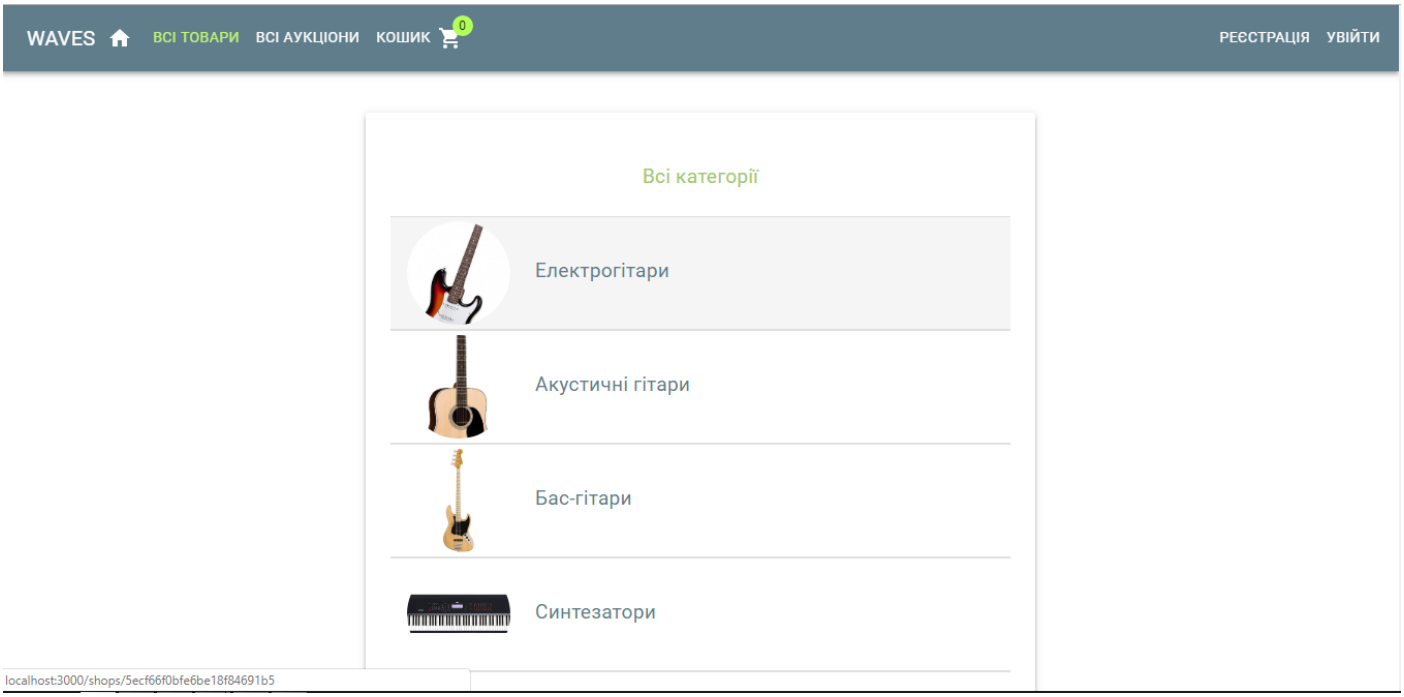

Рисунок 3.7 – Сторінка категорій товарів

Після обрання потрібної категорії буде відкрито сторінку з інформацією про категорію та список доступних товарів в ній (демонстрація на рисунку 3.8).

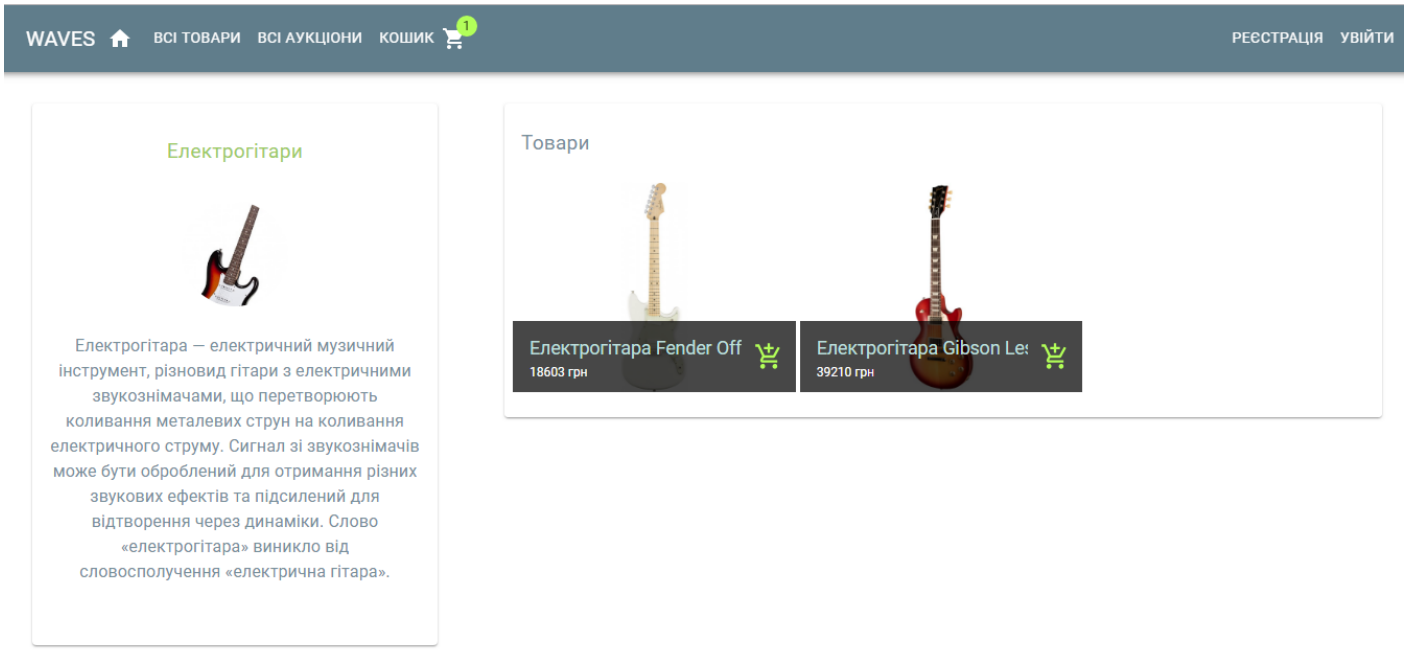

Рисунок 3.8 – Список доступних товарів категорії

Натіснувши на товар користувач отримає змогу прочитати детальніше про нього та побачити зправа список товарів, шо переглядав нещодавно (приклад на рисунку 3.9).

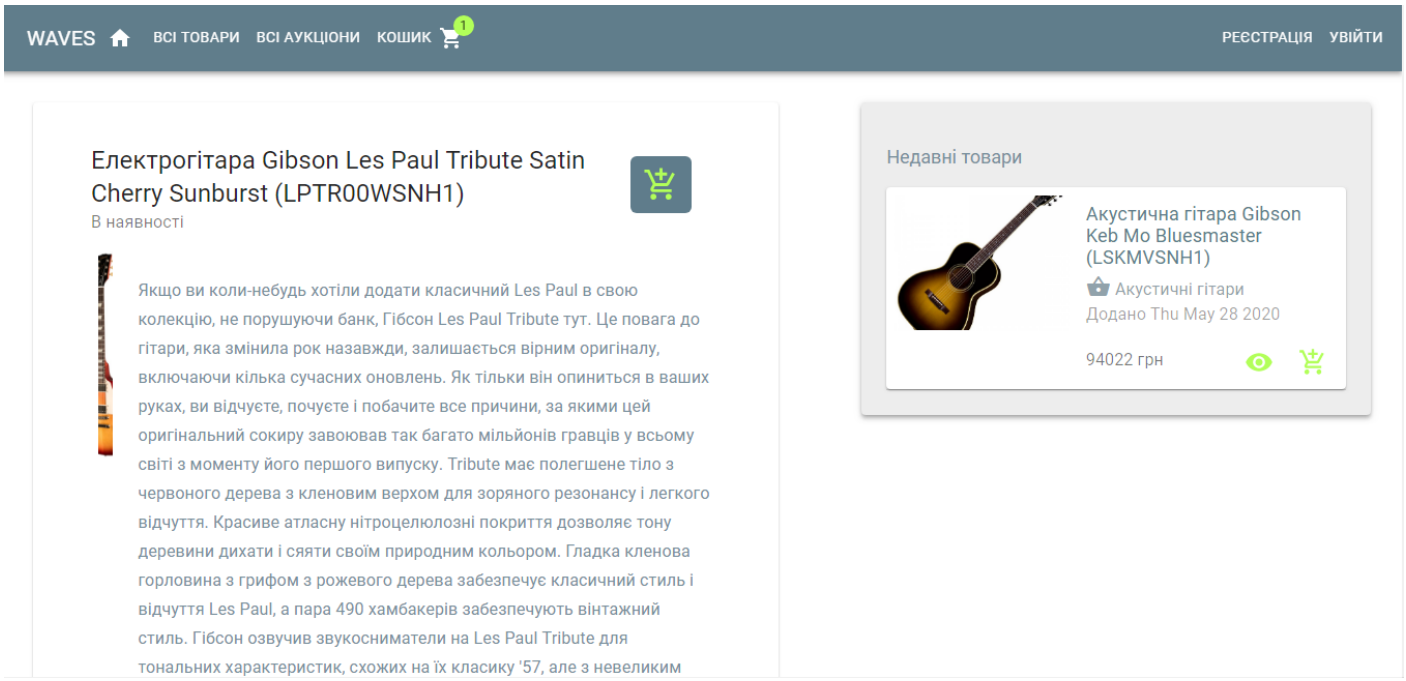

Рисунок 3.9 – Інформація про товар

Натиснувши у верхньому меню «Всі аукціони» буде завантаженр список аукціонів, що проводятсся на даний момент (рисунок 3.10).

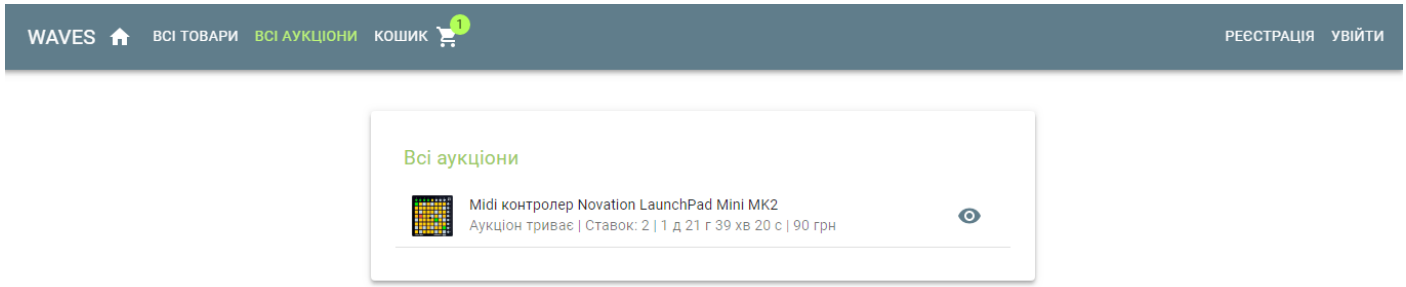

Рисунок 3.10 – Список товарів на аукціоні

Натиснувши на товар користувач відкриє сторінку з його детальним описом, що зобрадено на рисунку 3.11.

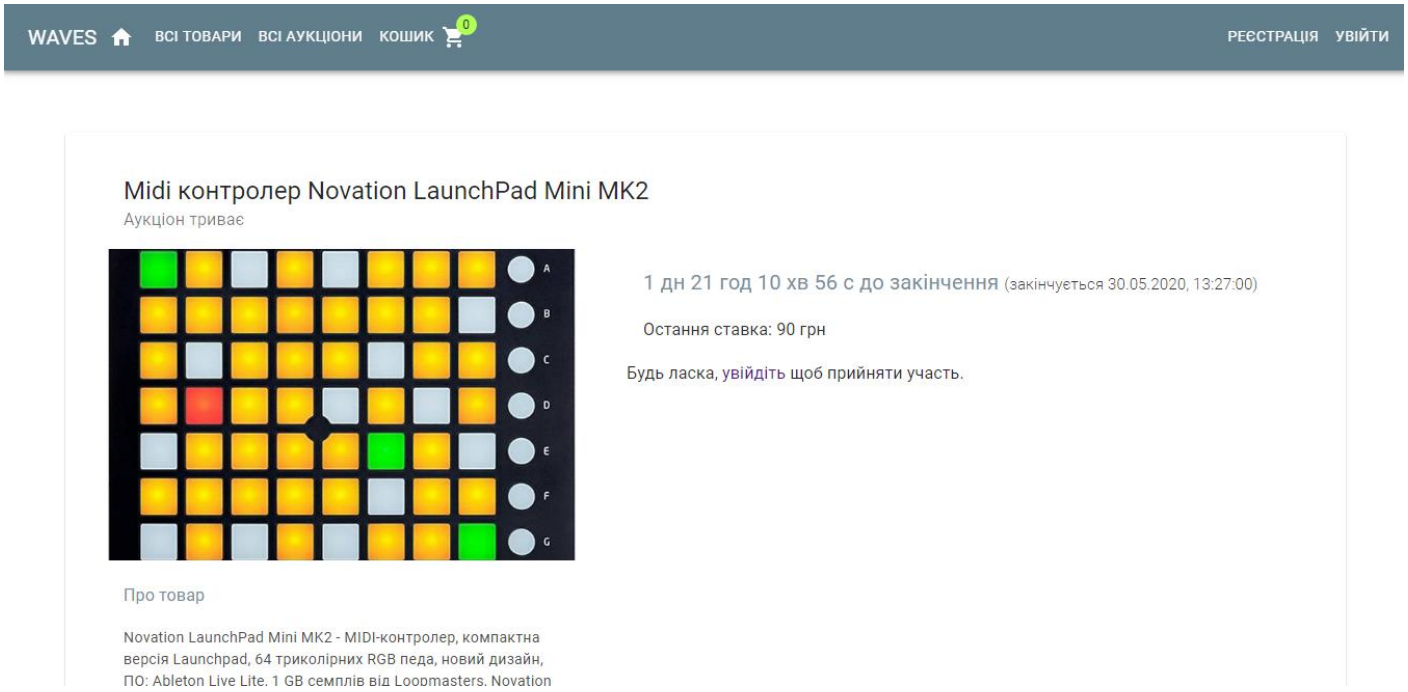

Рисунок 3.11 – Інформація про товар на аукціоні

Для розширення функціоналу, яким можна користувать, потрібно зареєструватися, натиснувши «Реєстраія» у верхньому меню. Відкриється сторінка з формою реєстрації, що зображена на рисунку 3.12.

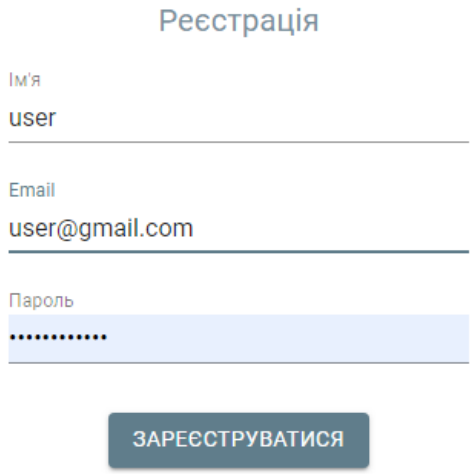

Рисунок 3.12 – Форма реєстрації

Після того як користувач введе всі необхідні дані та натисне кнопку «зареєструватися», буде виведено повідомлення про успішну реєстрацію (рисунок 3.13) та буде надана можливість увійти в особистий кабінет.

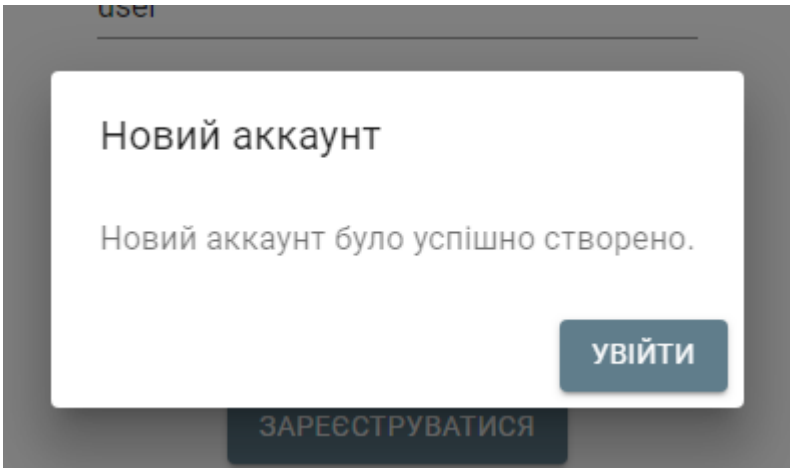

Рисунок 3.13 – Повідомлення про реєстрацію

Після натискання кнопки «Увійти» користувача буде переправлено на сторінку з формою авторизації, яка зображена на рисунку 3.14. Коли користувач знаходиться у власному кабінеті, йому надається розширений функціонал. По-перше – це можливість замовити товар на сайті. Натиснувши на іконку кошика на товарі та перейшовши до самого кошику буде відображено картку кошику (рисунок 3.15).

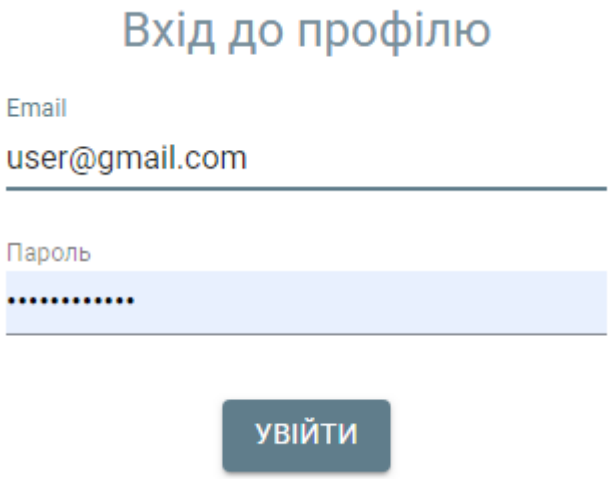

Рисунок 3.14 – Форма авторизації

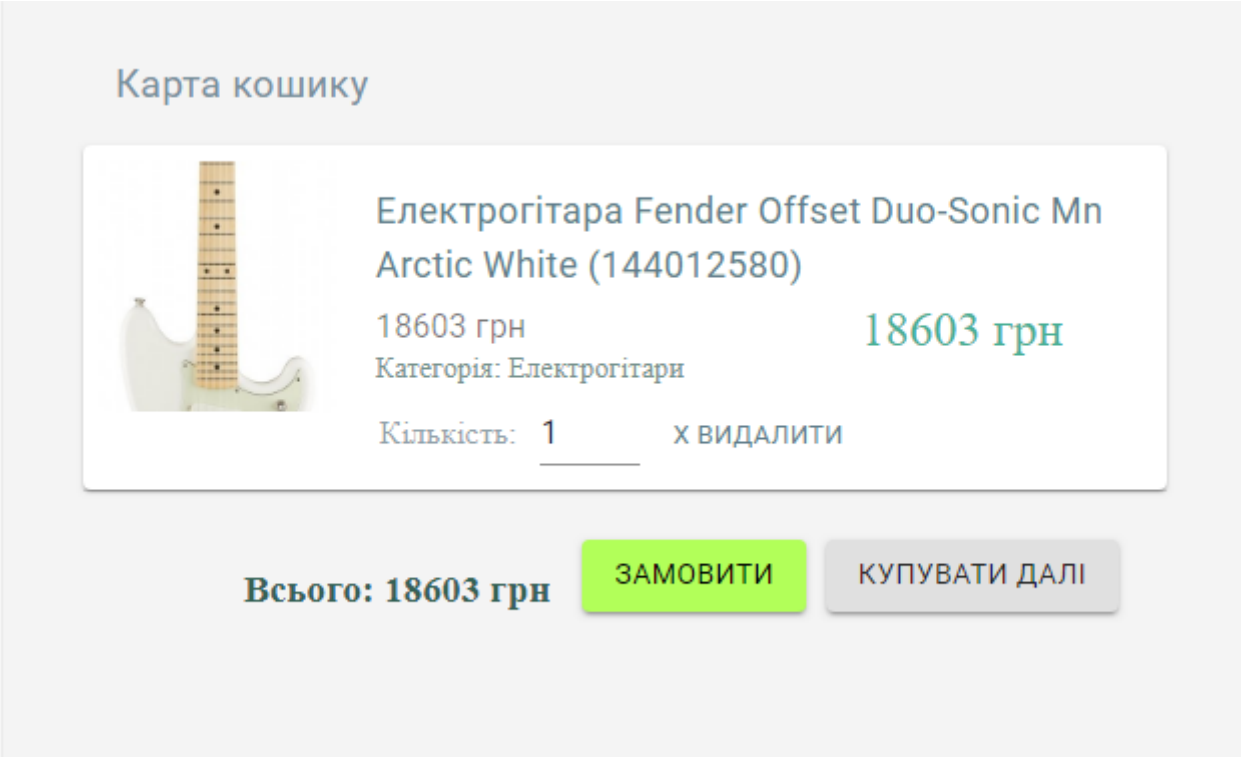

Рисунок 3.15 – Картка кошику.

Тут користувач може обрати кількість одиниць товару, продовжити купувати (кнопка «Купувати далі перенаправляє на домашню сторінку) або замовити товар. При натисканні на кнопку «Замовити» буде відображено форму для вводу персональних даних, яка зображена на рисунку 3.16.

Користувач повиинен ввести свої дані, після чого натиснути «Замовити». На телефон, що прив’язаний до введеної картки прийде сповіщення, щодо підтвердження покупки. Після підтвердження покупки користувачу залишається тільки чекати швидку доставку на вказаний адрес.

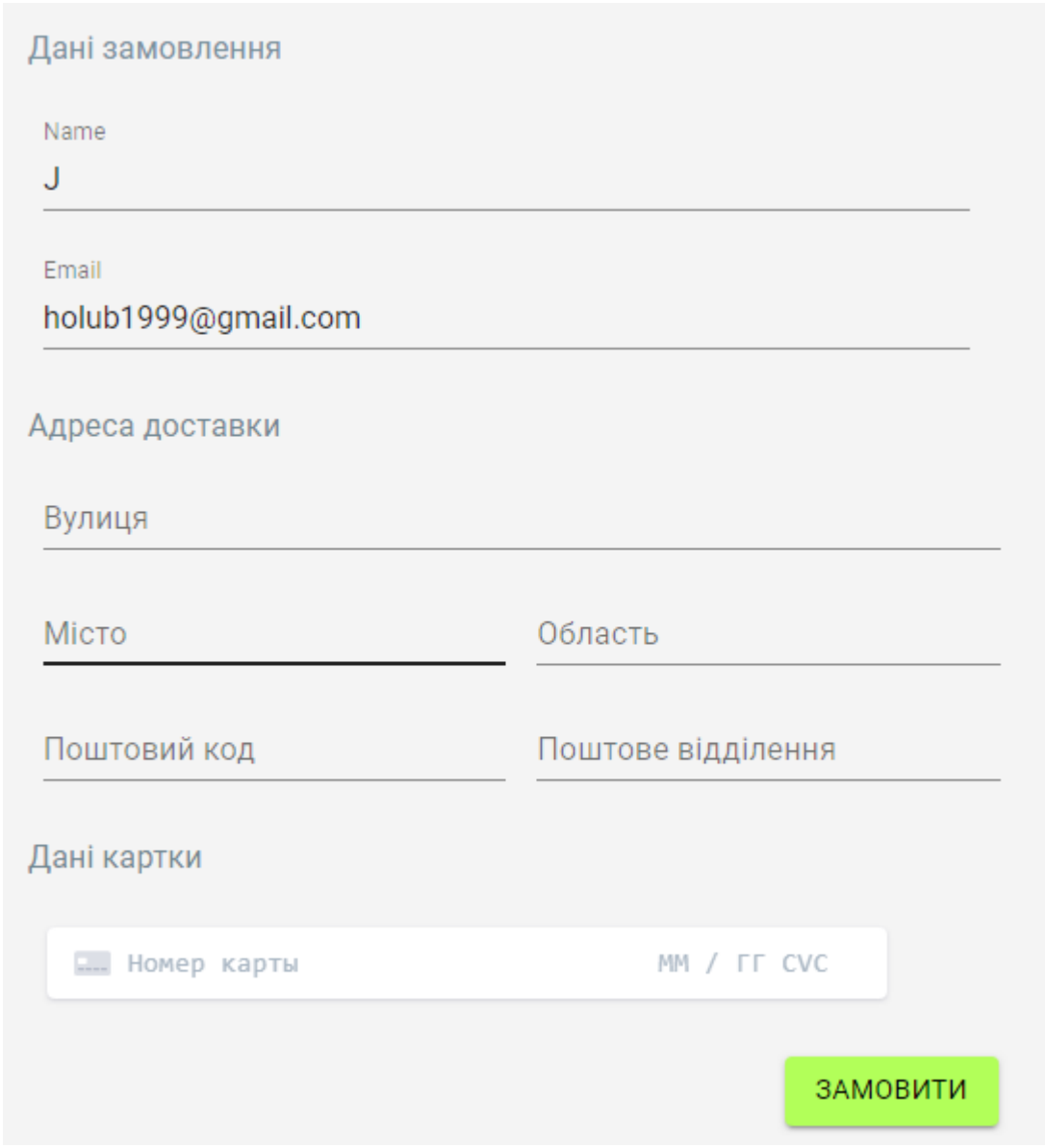

Рисунок 3.16 – Форма для замовлення товару

Ще однією функцією, що відкривається користувачеві після реєстрації є участь у аукціонах. На строінці товару, що виставлено на аукціон з’явиться список людей, що приймають участь, їх ставки та можливість створити свою ставку (рисунок 3.17).

У вкалдинці головного меню «Мій профіль» є можливість переглядати свої замовлення, аукіони, редагувати свої дані або видалити профіль, що зображено на рисунку 3.18.

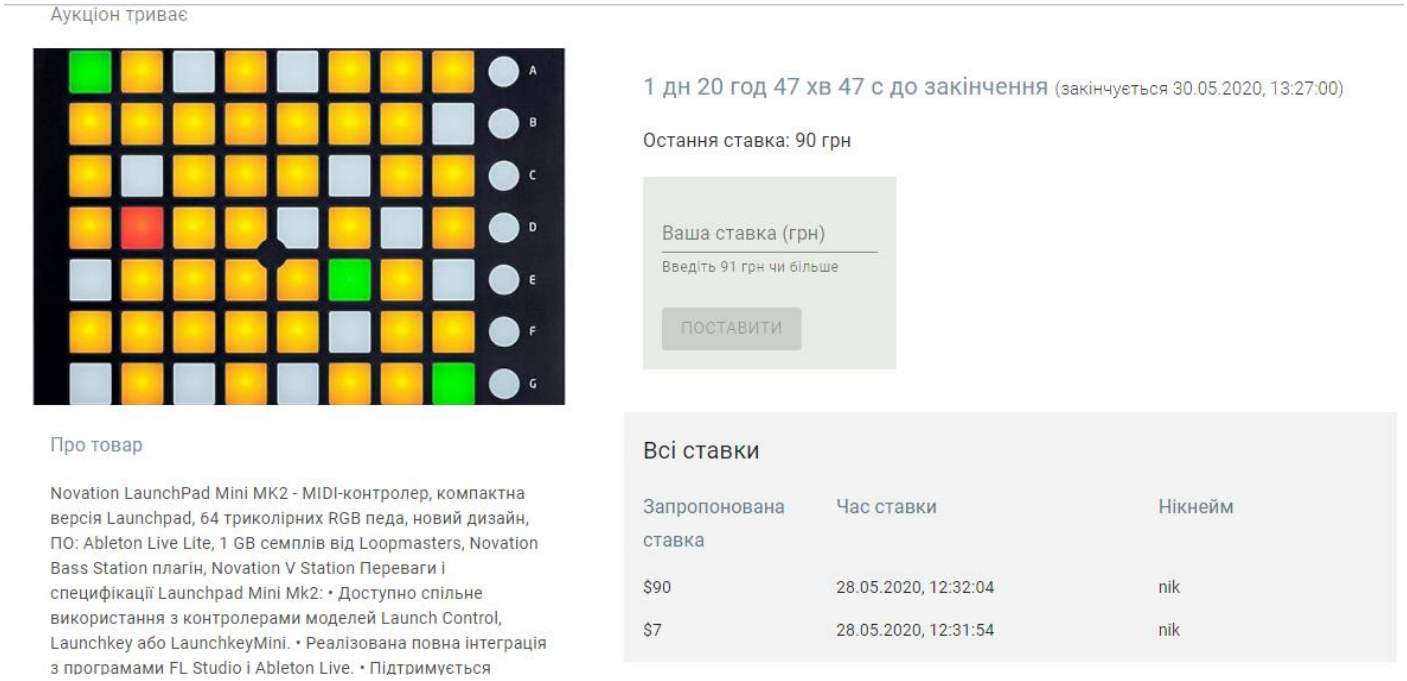

Рисунок 3.17 – Повний функціонал аукціону

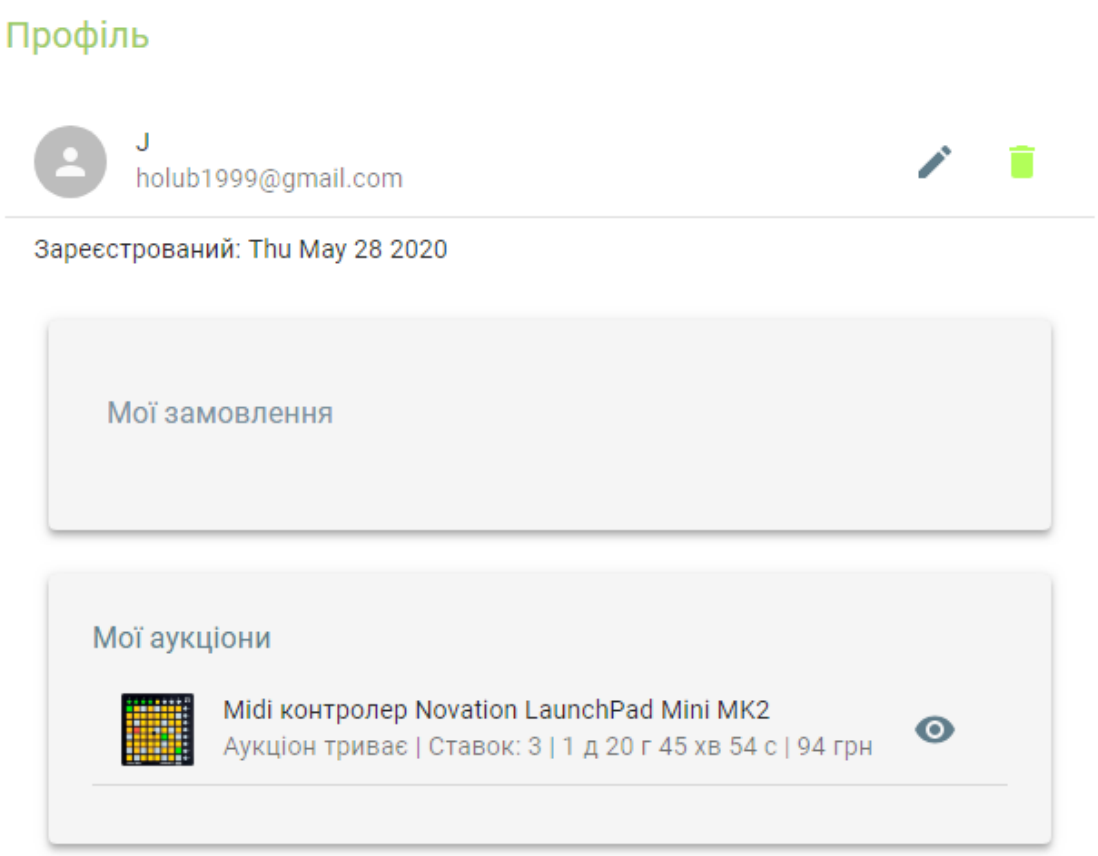

Рисунок 3.18 – Профіль користувача

При вході на сайт адміністратора у меню відкриється функціонал для управління веб-додатком. По-перше – це вкладинка «Категорії адміністратора», що зображена на рисунку 3.19. В ній можна редагувати категорії, створювати нові видаляти їх. Перейшовши до редагування категорій з’явиться функціонал для редагування товарів, які до неї належать. \

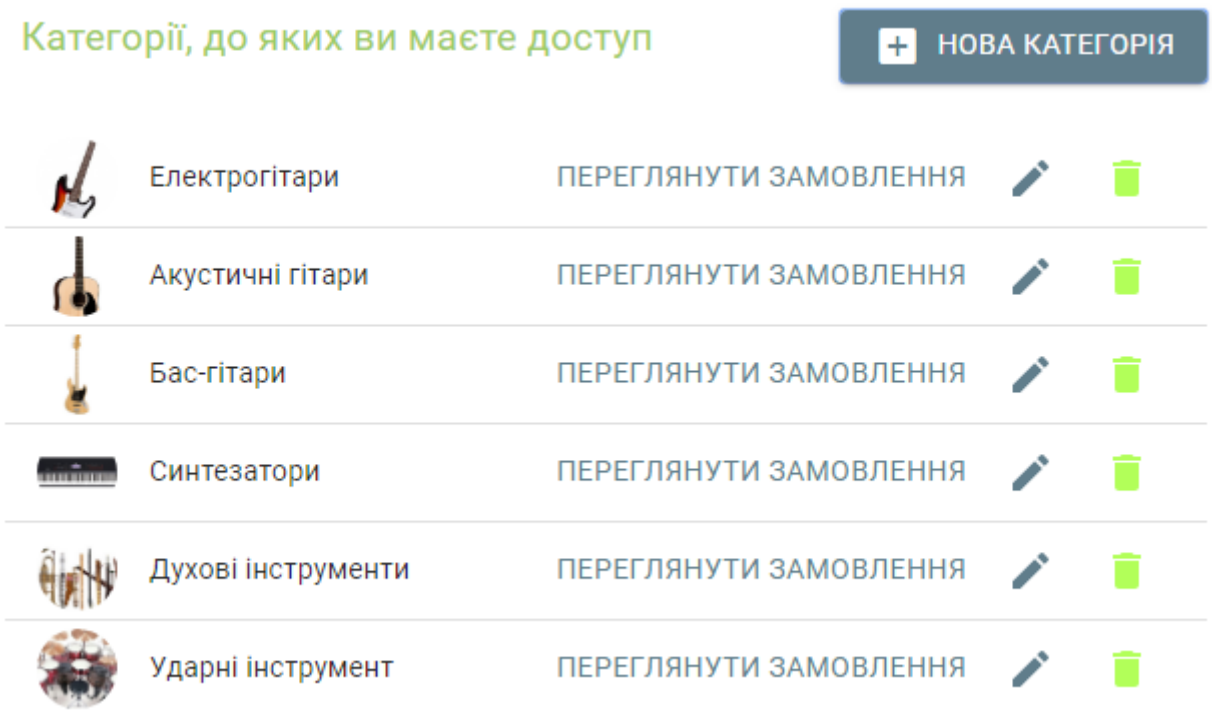

Рисунок 3.19 – Панель керування категоріями та товарами

Також стає доступним можливість редагування, видалення створення аукціонів при переході на вкладинку «Мої аукціони», що зображено на рисунку 3.20.

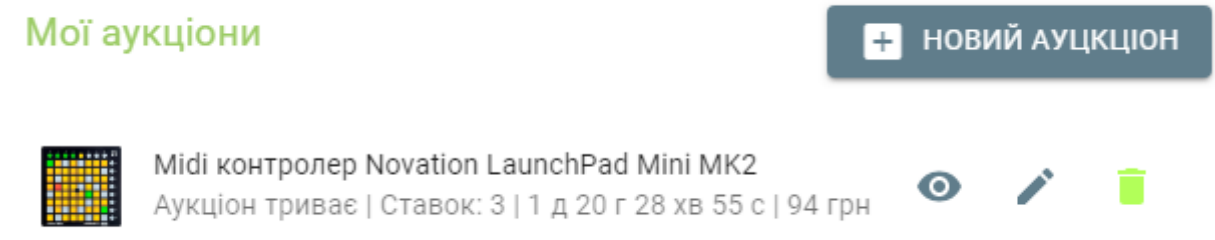

Рисунок 3.20 – Панель керування аукціонами

ВИСНОВКИ

При виконанні дипломного проекту була проаналізована література та Інтернет джерела з питання електронної комерції, проаналізовано літературу і визначено поняття, пов'язані з веб-додатками, їх роботою, організацією, розробкою і просуванням.

В ході роботи були вивчені і проаналізовані програмні засоби для розробки сайтів. Так само розглянуті схожі сайти, виявлено їх переваги та недоліки, що лягло в основу створення програмного продукту.

В якості веб-сервера було обрано платформу Node.js. Важливими критеріями для відбору стали мультиплатформеність і безкоштовне розповсюдження. Крім того, Node.js в даний момент один з найбільш популярних веб-серверів в усьому світі.

Створення розподіленої архітектури додатку підвищило надійність і продуктивність програмного продукту. Чітко визначена роль кожної з підсистем на всіх рівнях продукту в значній мірі спростила завдання проектування і реалізації системи, дозволила забезпечити модифікацію продукту і його підтримку.

Розраблений веб-додаток відпвідає вимогам користувачів, володіє помірними потребами в апаратних ресурсах, заснований на переносних незалежних технологіях.

Програмний продукт пройшов ряд тестових випробувань на предмет виявлення помилок в проектуванні і реалізації програми. Програмний продукт відповідає вимогам і має функціональність, надійністю, мобільністю та зручністю використання.

В результаті, все це дозволяє забезпечити більш тісний контакт з клієнтами через Інтернет, надає можливість пошуку нових контактів. За допомогою даного програмного продукту клієнти мають можливість отримати потрібну інформацію про товари і магазині, не виходячи з дому.

СПИСОК ЛІТЕРАТУРИ

1. https://ru.wikipedia.org/wiki/Mongo. – [Електронний ресурс]. – MongoDB.

2. Остроух А. В. Веб-разработка/ Остроух А. В. – Красноярск: Научно-инновационный центр, 2015. – 110 с.

3. https://ua.wikipedia.org/wiki/NodeJS. – [Електронний ресурс]. – Програмна платформа Node.js.

4. https://metanit.com/js/tutorial/1.1.php. – [Електронний ресурс]. – Мова програмування JavaScript.

5. Ріхтер Д. HTML, CSS, ECMAScript 5. 5-те видання/ Ріхтер Д. – Львів: Центр ЛНУ ім. І. Франка, 2011.- 567 с.

6. https://metanit.com/js/tutorial/1.2.php. – [Електронний ресурс]. – Робота з Visual Studio Code.

ДОДАТОК

// модуль auction.controller.js

```
import Auction from '../models/auction.model'
import extend from 'lodash/extend'
import errorHandler from '../helpers/dbErrorHandler'
import formidable from 'formidable'
import fs from 'fs'
import defaultImage from './../../client/assets/images/default.png'

const create = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Image could not be uploaded"
      })
    }
    let auction = new Auction(fields)
    auction.seller= req.profile
    if(files.image){
      auction.image.data = fs.readFileSync(files.image.path)
      auction.image.contentType = files.image.type
    }
    try {
      let result = await auction.save()
      res.status(200).json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}

const auctionByID = async (req, res, next, id) => {
  try {
    let auction = await Auction.findById(id).populate('seller', '_id name').populate('bids.bidder', '_id name').exec()
    if (!auction)
      return res.status('400').json({
        error: "Auction not found"
      })
    req.auction = auction
    next()
  } catch (err) {
    return res.status('400').json({
```

```
      error: "Could not retrieve auction"
    })
  }
}

const photo = (req, res, next) => {
  if(req.auction.image.data){
    res.set("Content-Type", req.auction.image.contentType)
    return res.send(req.auction.image.data)
  }
  next()
}
const defaultPhoto = (req, res) => {
  return res.sendFile(process.cwd()+defaultImage)
}

const read = (req, res) => {
  req.auction.image = undefined
  return res.json(req.auction)
}

const update = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Photo could not be uploaded"
      })
    }
    let auction = req.auction
    auction = extend(auction, fields)
    auction.updated = Date.now()
    if(files.image){
      auction.image.data = fs.readFileSync(files.image.path)
      auction.image.contentType = files.image.type
    }
    try {
      let result = await auction.save()
      res.json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}

const remove = async (req, res) => {
  try {
```

```
    let auction = req.auction
    let deletedAuction = auction.remove()
    res.json(deletedAuction)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listOpen = async (req, res) => {
  try {
    let auctions = await Auction.find({ 'bidEnd': { $gt: new Date() }}).sort('bidStart').populate('seller', '_id
name').populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}


const listBySeller = async (req, res) => {
  try {
    let auctions = await Auction.find({seller: req.profile._id}).populate('seller', '_id
name').populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}
const listByBidder = async (req, res) => {
  try {
    let auctions = await Auction.find({'bids.bidder': req.profile._id}).populate('seller', '_id
name').populate('bids.bidder', '_id name')
    res.json(auctions)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const isSeller = (req, res, next) => {
  const isSeller = req.auction && req.auth && req.auction.seller._id == req.auth._id
  if(!isSeller){
    return res.status('403').json({
```

```
      error: "User is not authorized"
    })
  }
  next()
}

export default {
  create,
  auctionByID,
  photo,
  defaultPhoto,
  listOpen,
  listBySeller,
  listByBidder,
  read,
  update,
  isSeller,
  remove
}
```

// модуль auth.controller.js

```
import User from '../models/user.model'
import jwt from 'jsonwebtoken'
import expressJwt from 'express-jwt'
import config from './../../config/config'

const signin = async (req, res) => {
  try {
    let user = await User.findOne({
      "email": req.body.email
    })

    if (!user)
      return res.status('401').json({
        error: "User not found"
      })

    if (!user.authenticate(req.body.password)) {
      return res.status('401').send({
        error: "Email and password don't match."
      })
    }

    const token = jwt.sign({
      _id: user._id
    }, config.jwtSecret)

    res.cookie("t", token, {
      expire: new Date() + 9999
    })
```

```
      return res.json({
        token,
        user: {_id: user._id, name: user.name, email: user.email, seller: user.seller}
      })
  } catch (err) {
    return res.status('401').json({
      error: "Could not sign in"
    })
  }
}

const signout = (req, res) => {
  res.clearCookie("t")
  return res.status('200').json({
    message: "signed out"
  })
}

const requireSignin = expressJwt({
  secret: config.jwtSecret,
  userProperty: 'auth'
})

const hasAuthorization = (req, res, next) => {
  const authorized = req.profile && req.auth && req.profile._id == req.auth._id
  if (!(authorized)) {
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}

export default {
  signin,
  signout,
  requireSignin,
  hasAuthorization
}
```

//модуль bidding.controller.js

```
import Auction from '../models/auction.model'

export default (server) => {
    const io = require('socket.io').listen(server)
    io.on('connection', function(socket){
        socket.on('join auction room', data => {
            socket.join(data.room)
```

```
        })
        socket.on('leave auction room', data => {
           socket.leave(data.room)
        })
        socket.on('new bid', data => {
           bid(data.bidInfo, data.room)
        })
    })
    const bid = async (bid, auction) => {
        try {
          let result = await Auction.findOneAndUpdate({_id:auction, $or:
[{'bids.0.bid':{$lt:bid.bid}},{bids:{$eq:[]}} ]}, {$push: {bids: {$each:[bid], $position: 0}}}, {new: true})
                            .populate('bids.bidder', '_id name')
                            .populate('seller', '_id name')
                            .exec()
           io
           .to(auction)
           .emit('new bid', result)
        } catch(err) {
          console.log(err)
        }
    }
}
```

// модуль order.controller.js

```
import {Order, CartItem} from '../models/order.model'
import errorHandler from './../helpers/dbErrorHandler'

const create = async (req, res) => {
  try {
    req.body.order.user = req.profile
    let order = new Order(req.body.order)
    let result = await order.save()
    res.status(200).json(result)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listByShop = async (req, res) => {
  try {
    let orders = await Order.find({"products.shop": req.shop._id})
      .populate({path: 'products.product', select: '_id name price'})
      .sort('-created')
      .exec()
    res.json(orders)
  } catch (err){
    return res.status(400).json({
```

```
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const update = async (req, res) => {
  try {
    let order = await Order.updateOne({'products._id':req.body.cartItemId}, {
        'products.$.status': req.body.status
    })
      res.json(order)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const getStatusValues = (req, res) => {
  res.json(CartItem.schema.path('status').enumValues)
}

const orderByID = async (req, res, next, id) => {
  try {
    let order = await Order.findById(id).populate('products.product', 'name price').populate('products.shop',
'name').exec()
    if (!order)
      return res.status('400').json({
        error: "Order not found"
      })
    req.order = order
    next()
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listByUser = async (req, res) => {
  try{
    let orders = await Order.find({ "user": req.profile._id })
        .sort('-created')
        .exec()
    res.json(orders)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
```

```
}

const read = (req, res) => {
  return res.json(req.order)
}

export default {
  create,
  listByShop,
  update,
  getStatusValues,
  orderByID,
  listByUser,
  read
}
```

//модуль product.controller.js

```
import Product from '../models/product.model'
import extend from 'lodash/extend'
import errorHandler from './../helpers/dbErrorHandler'
import formidable from 'formidable'
import fs from 'fs'
import defaultImage from './../../client/assets/images/default.png'

const create = (req, res, next) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        message: "Image could not be uploaded"
      })
    }
    let product = new Product(fields)
    product.shop= req.shop
    if(files.image){
      product.image.data = fs.readFileSync(files.image.path)
      product.image.contentType = files.image.type
    }
    try {
      let result = await product.save()
      res.json(result)
    } catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}
```

```
const productByID = async (req, res, next, id) => {
  try {
    let product = await Product.findById(id).populate('shop', '_id name').exec()
    if (!product)
      return res.status('400').json({
        error: "Product not found"
      })
    req.product = product
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve product"
    })
  }
}

const photo = (req, res, next) => {
  if(req.product.image.data){
    res.set("Content-Type", req.product.image.contentType)
    return res.send(req.product.image.data)
  }
  next()
}
const defaultPhoto = (req, res) => {
  return res.sendFile(process.cwd()+defaultImage)
}

const read = (req, res) => {
  req.product.image = undefined
  return res.json(req.product)
}

const update = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      return res.status(400).json({
        message: "Photo could not be uploaded"
      })
    }
    let product = req.product
    product = extend(product, fields)
    product.updated = Date.now()
    if(files.image){
      product.image.data = fs.readFileSync(files.image.path)
      product.image.contentType = files.image.type
    }
    try {
```

```
      let result = await product.save()
      res.json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}

const remove = async (req, res) => {
  try{
    let product = req.product
    let deletedProduct = await product.remove()
    res.json(deletedProduct)

  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listByShop = async (req, res) => {
  try {
    let products = await Product.find({shop: req.shop._id}).populate('shop', '_id name').select('-image')
    res.json(products)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listLatest = async (req, res) => {
  try {
    let products = await Product.find({}).sort('-created').limit(5).populate('shop', '_id name').exec()
    res.json(products)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listRelated = async (req, res) => {
  try{
    let products = await Product.find({ "_id": { "$ne": req.product }, "category": req.product.category}).limit(5).populate('shop', '_id name').exec()
    res.json(products)
```

```
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listCategories = async (req, res) => {
  try {
    let products = await Product.distinct('category',{})
    res.json(products)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const list = async (req, res) => {
  const query = {}
  if(req.query.search)
    query.name = {'$regex': req.query.search, '$options': "i"}
  if(req.query.category && req.query.category != 'All')
    query.category =  req.query.category
  try {
    let products = await Product.find(query).populate('shop', '_id name').select('-image').exec()
    res.json(products)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const decreaseQuantity = async (req, res, next) => {
  let bulkOps = req.body.order.products.map((item) => {
    return {
        "updateOne": {
            "filter": { "_id": item.product._id } ,
            "update": { "$inc": {"quantity": -item.quantity} }
        }
    }
  })
   try {
     await Product.bulkWrite(bulkOps, {})
     next()
   } catch (err){
      return res.status(400).json({
        error: "Could not update product"
      })
```

```
  }
}

const increaseQuantity = async (req, res, next) => {
  try {
    await Product.findByIdAndUpdate(req.product._id, {$inc: {"quantity": req.body.quantity}}, {new: true})
    .exec()
      next()
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

export default {
  create,
  productByID,
  photo,
  defaultPhoto,
  read,
  update,
  remove,
  listByShop,
  listLatest,
  listRelated,
  listCategories,
  list,
  decreaseQuantity,
  increaseQuantity
}
```

//модуль shop.controller.js

```
import Shop from '../models/shop.model'
import extend from 'lodash/extend'
import errorHandler from './../helpers/dbErrorHandler'
import formidable from 'formidable'
import fs from 'fs'
import defaultImage from './../../client/assets/images/default.png'

const create = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
  form.parse(req, async (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Image could not be uploaded"
      })
    }
    let shop = new Shop(fields)
    shop.owner= req.profile
```

```
    if(files.image){
      shop.image.data = fs.readFileSync(files.image.path)
      shop.image.contentType = files.image.type
    }
    try {
      let result = await shop.save()
      res.status(200).json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}

const shopByID = async (req, res, next, id) => {
  try {
    let shop = await Shop.findById(id).populate('owner', '_id name').exec()
    if (!shop)
      return res.status('400').json({
        error: "Shop not found"
      })
    req.shop = shop
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve shop"
    })
  }
}

const photo = (req, res, next) => {
  if(req.shop.image.data){
    res.set("Content-Type", req.shop.image.contentType)
    return res.send(req.shop.image.data)
  }
  next()
}
const defaultPhoto = (req, res) => {
  return res.sendFile(process.cwd()+defaultImage)
}

const read = (req, res) => {
  req.shop.image = undefined
  return res.json(req.shop)
}

const update = (req, res) => {
  let form = new formidable.IncomingForm()
  form.keepExtensions = true
```

```
  form.parse(req, async (err, fields, files) => {
    if (err) {
      res.status(400).json({
        message: "Photo could not be uploaded"
      })
    }
    let shop = req.shop
    shop = extend(shop, fields)
    shop.updated = Date.now()
    if(files.image){
      shop.image.data = fs.readFileSync(files.image.path)
      shop.image.contentType = files.image.type
    }
    try {
      let result = await shop.save()
      res.json(result)
    }catch (err){
      return res.status(400).json({
        error: errorHandler.getErrorMessage(err)
      })
    }
  })
}

const remove = async (req, res) => {
  try {
    let shop = req.shop
    let deletedShop = shop.remove()
    res.json(deletedShop)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const list = async (req, res) => {
  try {
    let shops = await Shop.find()
    res.json(shops)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const listByOwner = async (req, res) => {
  try {
    let shops = await Shop.find({owner: req.profile._id}).populate('owner', '_id name')
```

```
    res.json(shops)
  } catch (err){
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const isOwner = (req, res, next) => {
  const isOwner = req.shop && req.auth && req.shop.owner._id == req.auth._id
  if(!isOwner){
    return res.status('403').json({
      error: "User is not authorized"
    })
  }
  next()
}

export default {
  create,
  shopByID,
  photo,
  defaultPhoto,
  list,
  listByOwner,
  read,
  update,
  isOwner,
  remove
}
```

//модуль user.controller.js

```
import User from '../models/user.model'
import extend from 'lodash/extend'
import errorHandler from './../helpers/dbErrorHandler'
import request from 'request'
import config from './../../config/config'
import stripe from 'stripe'

const myStripe = stripe(config.stripe_test_secret_key)

const create = async (req, res) => {
  const user = new User(req.body)
  try {
    await user.save()
    return res.status(200).json({
      message: "Successfully signed up!"
    })
  } catch (err) {
    return res.status(400).json({
```

```
      error: errorHandler.getErrorMessage(err)
    })
  }
}

/**
 * Load user and append to req.
 */
const userByID = async (req, res, next, id) => {
  try {
    let user = await User.findById(id)
    if (!user)
      return res.status('400').json({
        error: "User not found"
      })
    req.profile = user
    next()
  } catch (err) {
    return res.status('400').json({
      error: "Could not retrieve user"
    })
  }
}

const read = (req, res) => {
  req.profile.hashed_password = undefined
  req.profile.salt = undefined
  return res.json(req.profile)
}

const list = async (req, res) => {
  try {
    let users = await User.find().select('name email updated created')
    res.json(users)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const update = async (req, res) => {
  try {
    let user = req.profile
    user = extend(user, req.body)
    user.updated = Date.now()
    await user.save()
    user.hashed_password = undefined
    user.salt = undefined
    res.json(user)
```

```
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const remove = async (req, res) => {
  try {
    let user = req.profile
    let deletedUser = await user.remove()
    deletedUser.hashed_password = undefined
    deletedUser.salt = undefined
    res.json(deletedUser)
  } catch (err) {
    return res.status(400).json({
      error: errorHandler.getErrorMessage(err)
    })
  }
}

const isSeller = (req, res, next) => {
  const isSeller = req.profile && req.profile.seller
  if (!isSeller) {
    return res.status('403').json({
      error: "User is not a seller"
    })
  }
  next()
}

const stripe_auth = (req, res, next) => {
  request({
    url: "https://connect.stripe.com/oauth/token",
    method: "POST",
    json: true,
    body: {client_secret:config.stripe_test_secret_key,code:req.body.stripe, grant_type:'authorization_code'}
  }, (error, response, body) => {
    //update user
    if(body.error){
      return res.status('400').json({
        error: body.error_description
      })
    }
    req.body.stripe_seller = body
    next()
  })
}

const stripeCustomer = (req, res, next) => {
```

```
  if(req.profile.stripe_customer){
      //update stripe customer
      myStripe.customers.update(req.profile.stripe_customer, {
          source: req.body.token
      }, (err, customer) => {
        if(err){
          return res.status(400).send({
            error: "Could not update charge details"
          })
        }
        req.body.order.payment_id = customer.id
        next()
      })
  }else{
      myStripe.customers.create({
            email: req.profile.email,
            source: req.body.token
      }).then((customer) => {
          User.update({'_id':req.profile._id},
            {'$set': { 'stripe_customer': customer.id }},
            (err, order) => {
              if (err) {
                return res.status(400).send({
                  error: errorHandler.getErrorMessage(err)
                })
              }
              req.body.order.payment_id = customer.id
              next()
            })
      })
  }
}

const createCharge = (req, res, next) => {
  if(!req.profile.stripe_seller){
    return res.status('400').json({
      error: "Please connect your Stripe account"
    })
  }
  myStripe.tokens.create({
    customer: req.order.payment_id,
  }, {
    stripeAccount: req.profile.stripe_seller.stripe_user_id,
  }).then((token) => {
      myStripe.charges.create({
        amount: req.body.amount * 100, //amount in cents
        currency: "usd",
        source: token.id,
      }, {
        stripeAccount: req.profile.stripe_seller.stripe_user_id,
```

```
      }).then((charge) => {
        next()
      })
  })
}

export default {
  create,
  userByID,
  read,
  list,
  remove,
  update,
  isSeller,
  stripe_auth,
  stripeCustomer,
  createCharge
}
```

//Модуль dbErrorHandler.js

```
'use strict'

/**
 * Get unique error field name
 */
const getUniqueErrorMessage = (err) => {
    let output
    try {
        let fieldName = err.message.substring(err.message.lastIndexOf('.$') + 2, err.message.lastIndexOf('_1'))
        output = fieldName.charAt(0).toUpperCase() + fieldName.slice(1) + ' already exists'
    } catch (ex) {
        output = 'Unique field already exists'
    }

    return output
}

/**
 * Get the error message from error object
 */
const getErrorMessage = (err) => {
    let message = ''

    if (err.code) {
        switch (err.code) {
            case 11000:
            case 11001:
                message = getUniqueErrorMessage(err)
                break
            default:
                message = 'Something went wrong'
```

```
        }
    } else {
        for (let errName in err.errors) {
            if (err.errors[errName].message) message = err.errors[errName].message
        }
    }

    return message
}

export default {getErrorMessage}
```

// модуль auction.model.js

```
import mongoose from 'mongoose'
const AuctionSchema = new mongoose.Schema({
  itemName: {
    type: String,
    trim: true,
    required: 'Item name is required'
  },
  description: {
    type: String,
    trim: true
  },
  image: {
    data: Buffer,
    contentType: String
  },
  updated: Date,
  created: {
    type: Date,
    default: Date.now
  },
  bidStart: {
    type: Date,
    default: Date.now
  },
  bidEnd: {
    type: Date,
    required: "Auction end time is required"
  },
  seller: {
    type: mongoose.Schema.ObjectId,
    ref: 'User'
  },
  startingBid: { type: Number, default: 0 },
  bids: [{
    bidder: {type: mongoose.Schema.ObjectId, ref: 'User'},
    bid: Number,
    time: Date
  }]
```

```
})

export default mongoose.model('Auction', AuctionSchema)
```

//модуль order.model.js

```
import mongoose from 'mongoose'
const CartItemSchema = new mongoose.Schema({
  product: {type: mongoose.Schema.ObjectId, ref: 'Product'},
  quantity: Number,
  shop: {type: mongoose.Schema.ObjectId, ref: 'Shop'},
  status: {type: String,
    default: 'Not processed',
    enum: ['Not processed' , 'Processing', 'Shipped', 'Delivered', 'Cancelled']}
})
const CartItem = mongoose.model('CartItem', CartItemSchema)
const OrderSchema = new mongoose.Schema({
  products: [CartItemSchema],
  customer_name: {
    type: String,
    trim: true,
    required: 'Name is required'
  },
  customer_email: {
    type: String,
    trim: true,
    match: [/.+\@.+\..+/, 'Please fill a valid email address'],
    required: 'Email is required'
  },
  delivery_address: {
    street: {type: String, required: 'Street is required'},
    city: {type: String, required: 'City is required'},
    state: {type: String},
    zipcode: {type: String, required: 'Zip Code is required'},
    country: {type: String, required: 'Country is required'}
  },
  payment_id: {},
  updated: Date,
  created: {
    type: Date,
    default: Date.now
  },
  user: {type: mongoose.Schema.ObjectId, ref: 'User'}
})

const Order = mongoose.model('Order', OrderSchema)

export {Order, CartItem}
```

//модуль product.model.js

```
import mongoose from 'mongoose'
const ProductSchema = new mongoose.Schema({
  name: {
```

```
    type: String,
    trim: true,
    required: 'Name is required'
  },
  image: {
    data: Buffer,
    contentType: String
  },
  description: {
    type: String,
    trim: true
  },
  category: {
    type: String
  },
  quantity: {
    type: Number,
    required: "Quantity is required"
  },
  price: {
    type: Number,
    required: "Price is required"
  },
  updated: Date,
  created: {
    type: Date,
    default: Date.now
  },
  shop: {type: mongoose.Schema.ObjectId, ref: 'Shop'}
})

export default mongoose.model('Product', ProductSchema)
```

//модуль shop.model.js

```
import mongoose from 'mongoose'
const ShopSchema = new mongoose.Schema({
  name: {
    type: String,
    trim: true,
    required: 'Name is required'
  },
  image: {
    data: Buffer,
    contentType: String
  },
  description: {
    type: String,
    trim: true
  },
  updated: Date,
  created: {
```

```
    type: Date,
    default: Date.now
  },
  owner: {type: mongoose.Schema.ObjectId, ref: 'User'}
})

export default mongoose.model('Shop', ShopSchema)
```

// модуль user.model

```
import mongoose from 'mongoose'
import crypto from 'crypto'
const UserSchema = new mongoose.Schema({
  name: {
    type: String,
    trim: true,
    required: 'Name is required'
  },
  email: {
    type: String,
    trim: true,
    unique: 'Email already exists',
    match: [/.+\@.+\..+/, 'Please fill a valid email address'],
    required: 'Email is required'
  },
  hashed_password: {
    type: String,
    required: "Password is required"
  },
  salt: String,
  updated: Date,
  created: {
    type: Date,
    default: Date.now
  },
  seller: {
    type: Boolean,
    default: false
  },
  stripe_seller: {},
  stripe_customer: {}
})

UserSchema
  .virtual('password')
  .set(function(password) {
    this._password = password
    this.salt = this.makeSalt()
    this.hashed_password = this.encryptPassword(password)
  })
  .get(function() {
    return this._password
```

```
  })

UserSchema.path('hashed_password').validate(function(v) {
  if (this._password && this._password.length < 6) {
    this.invalidate('password', 'Password must be at least 6 characters.')
  }
  if (this.isNew && !this._password) {
    this.invalidate('password', 'Password is required')
  }
}, null)

UserSchema.methods = {
  authenticate: function(plainText) {
    return this.encryptPassword(plainText) === this.hashed_password
  },
  encryptPassword: function(password) {
    if (!password) return ''
    try {
      return crypto
        .createHmac('sha1', this.salt)
        .update(password)
        .digest('hex')
    } catch (err) {
      return ''
    }
  },
  makeSalt: function() {
    return Math.round((new Date().valueOf() * Math.random())) + ''
  }
}

export default mongoose.model('User', UserSchema)
```

// модуль auction.routes.js

```
import express from 'express'
import userCtrl from '../controllers/user.controller'
import authCtrl from '../controllers/auth.controller'
import auctionCtrl from '../controllers/auction.controller'

const router = express.Router()

router.route('/api/auctions')
  .get(auctionCtrl.listOpen)

router.route('/api/auctions/bid/:userId')
  .get(auctionCtrl.listByBidder)

router.route('/api/auction/:auctionId')
  .get(auctionCtrl.read)

router.route('/api/auctions/by/:userId')
```

```
  .post(authCtrl.requireSignin, authCtrl.hasAuthorization, userCtrl.isSeller, auctionCtrl.create)
  .get(authCtrl.requireSignin, authCtrl.hasAuthorization, auctionCtrl.listBySeller)

router.route('/api/auctions/:auctionId')
  .put(authCtrl.requireSignin, auctionCtrl.isSeller, auctionCtrl.update)
  .delete(authCtrl.requireSignin, auctionCtrl.isSeller, auctionCtrl.remove)

router.route('/api/auctions/image/:auctionId')
  .get(auctionCtrl.photo, auctionCtrl.defaultPhoto)

router.route('/api/auctions/defaultphoto')
  .get(auctionCtrl.defaultPhoto)

router.param('auctionId', auctionCtrl.auctionByID)
router.param('userId', userCtrl.userByID)

export default router
```

//модуль auth.routes.js

```
import express from 'express'
import authCtrl from '../controllers/auth.controller'

const router = express.Router()

router.route('/auth/signin')
  .post(authCtrl.signin)
router.route('/auth/signout')
  .get(authCtrl.signout)

export default router
```

//модуль routes

```
import express from 'express'
import orderCtrl from '../controllers/order.controller'
import productCtrl from '../controllers/product.controller'
import authCtrl from '../controllers/auth.controller'
import shopCtrl from '../controllers/shop.controller'
import userCtrl from '../controllers/user.controller'

const router = express.Router()

router.route('/api/orders/:userId')
  .post(authCtrl.requireSignin, userCtrl.stripeCustomer, productCtrl.decreaseQuantity, orderCtrl.create)

router.route('/api/orders/shop/:shopId')
  .get(authCtrl.requireSignin, shopCtrl.isOwner, orderCtrl.listByShop)

router.route('/api/orders/user/:userId')
  .get(authCtrl.requireSignin, orderCtrl.listByUser)

router.route('/api/order/status_values')
  .get(orderCtrl.getStatusValues)
```

```
router.route('/api/order/:shopId/cancel/:productId')
  .put(authCtrl.requireSignin, shopCtrl.isOwner, productCtrl.increaseQuantity, orderCtrl.update)

router.route('/api/order/:orderId/charge/:userId/:shopId')
  .put(authCtrl.requireSignin, shopCtrl.isOwner, userCtrl.createCharge, orderCtrl.update)

router.route('/api/order/status/:shopId')
  .put(authCtrl.requireSignin, shopCtrl.isOwner, orderCtrl.update)

router.route('/api/order/:orderId')
  .get(orderCtrl.read)

router.param('userId', userCtrl.userByID)
router.param('shopId', shopCtrl.shopByID)
router.param('productId', productCtrl.productByID)
router.param('orderId', orderCtrl.orderByID)

export default router
```

//модуль product.routes

```
import express from 'express'
import productCtrl from '../controllers/product.controller'
import authCtrl from '../controllers/auth.controller'
import shopCtrl from '../controllers/shop.controller'

const router = express.Router()

router.route('/api/products/by/:shopId')
  .post(authCtrl.requireSignin, shopCtrl.isOwner, productCtrl.create)
  .get(productCtrl.listByShop)

router.route('/api/products/latest')
  .get(productCtrl.listLatest)

router.route('/api/products/related/:productId')
  .get(productCtrl.listRelated)

router.route('/api/products/categories')
  .get(productCtrl.listCategories)

router.route('/api/products')
  .get(productCtrl.list)

router.route('/api/products/:productId')
  .get(productCtrl.read)

router.route('/api/product/image/:productId')
  .get(productCtrl.photo, productCtrl.defaultPhoto)
router.route('/api/product/defaultphoto')
  .get(productCtrl.defaultPhoto)
```

```
router.route('/api/product/:shopId/:productId')
  .put(authCtrl.requireSignin, shopCtrl.isOwner, productCtrl.update)
  .delete(authCtrl.requireSignin, shopCtrl.isOwner, productCtrl.remove)

router.param('shopId', shopCtrl.shopByID)
router.param('productId', productCtrl.productByID)

export default router
```

//модуль shop.routes

```
import express from 'express'
import userCtrl from '../controllers/user.controller'
import authCtrl from '../controllers/auth.controller'
import shopCtrl from '../controllers/shop.controller'

const router = express.Router()

router.route('/api/shops')
  .get(shopCtrl.list)

router.route('/api/shop/:shopId')
  .get(shopCtrl.read)

router.route('/api/shops/by/:userId')
  .post(authCtrl.requireSignin, authCtrl.hasAuthorization, userCtrl.isSeller, shopCtrl.create)
  .get(authCtrl.requireSignin, authCtrl.hasAuthorization, shopCtrl.listByOwner)

router.route('/api/shops/:shopId')
  .put(authCtrl.requireSignin, shopCtrl.isOwner, shopCtrl.update)
  .delete(authCtrl.requireSignin, shopCtrl.isOwner, shopCtrl.remove)

router.route('/api/shops/logo/:shopId')
  .get(shopCtrl.photo, shopCtrl.defaultPhoto)

router.route('/api/shops/defaultphoto')
  .get(shopCtrl.defaultPhoto)

router.param('shopId', shopCtrl.shopByID)
router.param('userId', userCtrl.userByID)

export default router
```

//модуль user.routes

```
import express from 'express'
import userCtrl from '../controllers/user.controller'
import authCtrl from '../controllers/auth.controller'

const router = express.Router()

router.route('/api/users')
  .get(userCtrl.list)
```

```
  .post(userCtrl.create)

router.route('/api/users/:userId')
  .get(authCtrl.requireSignin, userCtrl.read)
  .put(authCtrl.requireSignin, authCtrl.hasAuthorization, userCtrl.update)
  .delete(authCtrl.requireSignin, authCtrl.hasAuthorization, userCtrl.remove)
router.route('/api/stripe_auth/:userId')
  .put(authCtrl.requireSignin, authCtrl.hasAuthorization, userCtrl.stripe_auth, userCtrl.update)

router.param('userId', userCtrl.userByID)

export default router
```

//модуль express

```
import express from 'express'
import path from 'path'
import bodyParser from 'body-parser'
import cookieParser from 'cookie-parser'
import compress from 'compression'
import cors from 'cors'
import helmet from 'helmet'
import Template from './../template'
import userRoutes from './routes/user.routes'
import authRoutes from './routes/auth.routes'
import shopRoutes from './routes/shop.routes'
import productRoutes from './routes/product.routes'
import orderRoutes from './routes/order.routes'
import auctionRoutes from './routes/auction.routes'

// modules for server side rendering
import React from 'react'
import ReactDOMServer from 'react-dom/server'
import MainRouter from './../client/MainRouter'
import { StaticRouter } from 'react-router-dom'

import { ServerStyleSheets, ThemeProvider } from '@material-ui/styles'
import theme from './../client/theme'
//end

//comment out before building for production
import devBundle from './devBundle'

const CURRENT_WORKING_DIR = process.cwd()
const app = express()

//comment out before building for production
devBundle.compile(app)

// parse body params and attache them to req.body
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: true }))
```

```
app.use(cookieParser())
app.use(compress())
// secure apps by setting various HTTP headers
app.use(helmet())
// enable CORS - Cross Origin Resource Sharing
app.use(cors())

app.use('/dist', express.static(path.join(CURRENT_WORKING_DIR, 'dist')))

// mount routes
app.use('/', userRoutes)
app.use('/', authRoutes)
app.use('/', shopRoutes)
app.use('/', productRoutes)
app.use('/', orderRoutes)
app.use('/', auctionRoutes)

app.get('*', (req, res) => {
  const sheets = new ServerStyleSheets()
  const context = {}
  const markup = ReactDOMServer.renderToString(
    sheets.collect(
      <StaticRouter location={req.url} context={context}>
          <ThemeProvider theme={theme}>
            <MainRouter/>
          </ThemeProvider>
      </StaticRouter>
     )
  )
    if (context.url) {
      return res.redirect(303, context.url)
    }
    const css = sheets.toString()
    res.status(200).send(Template({
      markup: markup,
      css: css
    }))
})

// Catch unauthorised errors
app.use((err, req, res, next) => {
  if (err.name === 'UnauthorizedError') {
    res.status(401).json({"error" : err.name + ": " + err.message})
  }else if (err) {
    res.status(400).json({"error" : err.name + ": " + err.message})
    console.log(err)
  }
})

export default app
```

//модуль server

```
import config from './../config/config'
import app from './express'
import mongoose from 'mongoose'
import bidding from './controllers/bidding.controller'

// Connection URL
mongoose.Promise = global.Promise
mongoose.connect(config.mongoUri, { useNewUrlParser: true, useCreateIndex: true, useUnifiedTopology: true, useFindAndModify: true })
mongoose.connection.on('error', () => {
  throw new Error(`unable to connect to database: ${config.mongoUri}`)
})

const server = app.listen(config.port, (err) => {
  if (err) {
    console.log(err)
  }
  console.info('Server started on port %s.', config.port)
})

bidding(server)
```

//модуль App

```
import React from 'react'
import MainRouter from './MainRouter'
import {BrowserRouter} from 'react-router-dom'
import { ThemeProvider } from '@material-ui/styles'
import theme from './theme'
import { hot } from 'react-hot-loader'

const App = () => {
  React.useEffect(() => {
    const jssStyles = document.querySelector('#jss-server-side')
    if (jssStyles) {
      jssStyles.parentNode.removeChild(jssStyles)
    }
  }, [])
  return (
  <BrowserRouter>
      <ThemeProvider theme={theme}>
        <MainRouter/>
      </ThemeProvider>
  </BrowserRouter>
)}

export default hot(module)(App)
```

//модуль main

```
import React from 'react'
import { hydrate } from 'react-dom'
import App from './App'
```

```
hydrate(<App/>, document.getElementById('root'))
```

//модуль routes

```
import React from 'react'
import {Route, Switch} from 'react-router-dom'
import Home from './core/Home'
import Users from './user/Users'
import Signup from './user/Signup'
import Signin from './auth/Signin'
import EditProfile from './user/EditProfile'
import Profile from './user/Profile'
import PrivateRoute from './auth/PrivateRoute'
import Menu from './core/Menu'
import NewShop from './shop/NewShop'
import Shops from './shop/Shops'
import MyShops from './shop/MyShops'
import Shop from './shop/Shop'
import EditShop from './shop/EditShop'
import NewProduct from './product/NewProduct'
import EditProduct from './product/EditProduct'
import Product from './product/Product'
import Cart from './cart/Cart'
import StripeConnect from './user/StripeConnect'
import ShopOrders from './order/ShopOrders'
import Order from './order/Order'
import MyAuctions from './auction/MyAuctions'
import OpenAuctions from './auction/OpenAuctions'
import NewAuction from './auction/NewAuction'
import EditAuction from './auction/EditAuction'
import Auction from './auction/Auction'

const MainRouter = () => {
  return (<div>
      <Menu/>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/users" component={Users}/>
        <Route path="/signup" component={Signup}/>
        <Route path="/signin" component={Signin}/>
        <PrivateRoute path="/user/edit/:userId" component={EditProfile}/>
        <Route path="/user/:userId" component={Profile}/>

        <Route path="/cart" component={Cart}/>
        <Route path="/product/:productId" component={Product}/>
        <Route path="/shops/all" component={Shops}/>
        <Route path="/shops/:shopId" component={Shop}/>

        <Route path="/order/:orderId" component={Order}/>
        <PrivateRoute path="/seller/orders/:shop/:shopId" component={ShopOrders}/>
```

```
        <PrivateRoute path="/seller/shops" component={MyShops}/>
        <PrivateRoute path="/seller/shop/new" component={NewShop}/>
        <PrivateRoute path="/seller/shop/edit/:shopId" component={EditShop}/>
        <PrivateRoute path="/seller/:shopId/products/new" component={NewProduct}/>
        <PrivateRoute path="/seller/:shopId/:productId/edit" component={EditProduct}/>

        <Route path="/seller/stripe/connect" component={StripeConnect}/>
        <PrivateRoute path="/myauctions" component={MyAuctions}/>
        <PrivateRoute path="/auction/new" component={NewAuction}/>
        <PrivateRoute path="/auction/edit/:auctionId" component={EditAuction}/>
        <Route path="/auction/:auctionId" component={Auction}/>
        <Route path="/auctions/all" component={OpenAuctions}/>
      </Switch>
    </div>)
}

export default MainRouter
```

//модуль api-auth

```
const signin = async (user) => {
  try {
    let response = await fetch('/auth/signin/', {
      method: 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      credentials: 'include',
      body: JSON.stringify(user)
    })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}

const signout = async () => {
  try {
    let response = await fetch('/auth/signout/', { method: 'GET' })
    return await response.json()
  } catch(err) {
    console.log(err)
  }
}

export {
  signin,
  signout
}
```

//модуль auth-helper

```
import { signout } from './api-auth.js'
```

```
const auth = {
  isAuthenticated() {
    if (typeof window == "undefined")
      return false

    if (sessionStorage.getItem('jwt'))
      return JSON.parse(sessionStorage.getItem('jwt'))
    else
      return false
  },
  authenticate(jwt, cb) {
    if (typeof window !== "undefined")
      sessionStorage.setItem('jwt', JSON.stringify(jwt))
    cb()
  },
  clearJWT(cb) {
    if (typeof window !== "undefined")
      sessionStorage.removeItem('jwt')
    cb()
    //optional
    signout().then((data) => {
      document.cookie = "t=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;"
    })
  },
  updateUser(user, cb) {
    if(typeof window !== "undefined"){
      if(sessionStorage.getItem('jwt')){
        let auth = JSON.parse(sessionStorage.getItem('jwt'))
        auth.user = user
        sessionStorage.setItem('jwt', JSON.stringify(auth))
        cb()
       }
    }
  }
}

export default auth
```