Державний вищий навчальний заклад
“Прикарпатський національний університет імені Василя Стефаника”
Кафедра інформаційних технологій

УДК 004

# ДИПЛОМНИЙ ПРОЕКТ

Тема Розробка бекенд частини проекту «Покупка автомобілів з Європи»

Спеціальність 121 Інженерія Програмного Забезпечення

## ПОЯСНЮВАЛЬНА ЗАПИСКА

ДП.ІПЗ-24.ПЗ
(позначення)

Рецензент
к.ф.-м.н., доц. Ткачук В. М.
(посада) (підпис) (дата) (розшифровка підпису)

Студент
ІПЗ-41 Сенів Т. О.
(шифр групи) (підпис) (дата) (розшифровка підпису)

Нормоконтролер
к.ф.-м.н., доц. Ткачук В. М.
(посада) (підпис) (дата) (розшифровка підпису)

Керівник дипломного проекту
к. ф.-м. н. Савка І. Я.
(посада) (підпис) (дата) (розшифровка підпису)

Допускається до захисту

Завідувач кафедри

к.т.н., доц. Козленко М. І.
(посада) (підпис) (дата) (розшифровка підпису)

2020

Державний вищий навчальний заклад
«Прикарпатський національний університет імені Василя Стефаника»
Факультет математики та інформатики Кафедра інформаційних технологій
Спеціальність Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Завідувач кафедри Козленко Микола Іванович
„_____”_____ 20___р.

**ЗАВДАННЯ**
**НА ВИКОНАННЯ ДИПЛОМНОГО ПРОЕКТУ**

Студенту Сеніву Тарасу Олеговичу
(прізвище, ім’я, по батькові студента)
1. Тема проекту Розробка бекенд частини проекту «Покупка автомобілів з Європи»
затверджена наказом від „25” жовтня 2019 р.№7
2. Термін здачі студентом закінченого проекту 2020-05-22
3. Вихідні дані до дипломного проекту Стандарт кафедри Інформаційних Технологій ПНУ “Вимоги до змісту та оформлення”, Node.js API, Express API
4. Зміст пояснювальної записки (перелік питань, що їх належить опрацювати):
    1. “Аналіз предметної області”;
    2. “Аналіз архітектури та засобів розробки”;
    3. “Розробка проекту”;
    4. “Бізнес план”;
5. Перелік графічного матеріалу (з точним забезпеченням обов’язкових креслень): “Мета роботи”, “Огляд альтернатив”, “Опис функціоналу”, “Use Case діаграма”, “Модель бази даних”, “Засоби розробки”, “Реалізація API”, “Огляд AWS S3”, “Реферальна система”, “Бізнес план”.

6. Дата видачі завдання 2019.09.11

Керівник _____ Савка І. Я.
(підпис) (розшифровка підпису)
Завдання прийняв до виконання _____ Сенів Т. О.
(підпис) (розшифровка підпису)

# КАЛЕНДАРНИЙ ПЛАН

| Номер і назва етапів дипломного проекту | Термін виконання етапів проекту | Примітка |
|---|---|---|
| 1. Обґрунтування актуальності, формулювання мети, завдання, предмету та об’єкту дослідження проекту | 10.10.2019 | Виконав |
| 2. Опрацювання джерел з теми проекту | 30.10.2019 | Виконав |
| 3. Підготовка I розділу проекту | 15.12.2019 | Виконав |
| 4. Підготовка II розділу проекту | 31.01.2020 | Виконав |
| 5. Підготовка III розділу проекту | 20.02.2020 | Виконав |
| 6. Підготовка IV розділу проекту | 20.03.2020 | Виконав |
| 7. Виправлення зауважень попередніх звітів. Підготовка вступу та висновків | 20.04.2020 | Виконав |
| 8. Оформлення проекту згідно вимог | 10.05.2020 | Виконав |

Студент Сенів Т. О.
(підпис) (розшифровка підпису)

Керівник проекту Савка І. Я.
(підпис) (розшифровка підпису)

# РЕФЕРАТ

Пояснювальна записка: 93 сторінки (без додатків), 72 рисунки, 23 джерела, 1 додаток на 41 сторінці.

Ключові слова: BACKEND, NODE, EXPRESS, HEROKU, APIDOC, REST, API.

Об`єктом дослідження є: платформа для продажу авто.

Мета роботи: розробити бекенд частину додатку для продажу авто.

Стислий опис тексту пояснювальної записки:

Робота описує розробку бекенд частини для вебзастосунку. Досліджено сучасні методи розробки, їх переваги та недоліки. Представлено процес створення дизайну архітектури додатку, вказано основні проблеми при його моделюванні. Описано взаємодію клієнта з API-сервісами, його права та безпеку. Стисло подано основні етапи розробки.

## ABSTRACT

Explanatory note: 93 pages (without appendix), 72 figures, 23 references, 1 appendix on 41 pages.

Key words: BACKEND, NODE, EXPRESS, HEROKU, APIDOC, REST, API.

Object of study: platform for selling vehicles.

Purpose: develop a backend part of the application for vehicle sales.

Brief description of the text of the explanatory note:

The explanatory note describes the development process of backend part for web application. Researched modern development methods, their advantages and disadvantages. Presented process of creation architecture design for application, indicated main problems of its modeling. Described the client's interaction with API-services, his rights and security. The main stages of development are briefly presented.

**ЗМІСТ**

## ПЕРЕЛІК ОСНОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

| | |
|---|---|
| HTTP | Hyper Text Transfer Protocol |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| API | Application Programming Interface |
| REST | Representational State Transfer |
| SOAP | Simple Object Access Protocol |
| UML | Unified Modeling Language |
| JWT | JSON Web Token |
| JS | Javascript |
| CSS | Cascading Style Sheets |

# ВСТУП

Інтернет торгівля – це процес реалізації фізичних і нефізичних товарів за допомогою спеціалізованих інтернет платформ, які дають можливість дистанційно оформити замовлення.

Вагомою перевагою інтернет магазину перед традиційним є значний виграш в кількості товару, який представлений для покупця. Зрозуміло, що традиційний магазин є фізично обмеженим в кількості товару, тоді як електронний варіант обмежується лише наявністю вільного дискового простору на сервері.

В останні роки об’єм цифрового ринку значно зростає. Згідно зі статистикою, ринок електронної торгівлі в Україні зріс в 2019р. на 25% порівняно з попереднім роком і продовжує рости в 2020р. Такий ріст дозволяє Україні наздоганяти рівень Європейської інтернет торгівлі. Стрімкий розвиток ринку зв’язаний з кризою, яка почалася в 2014р. і ліквідувала багато дрібних точок з продажу товарів. Це заставило українців більше звертатись до всесвітньої павутини для пошуку товарів, адже там є значно більший вибір та за більш нижчими цінами порівняно зі стаціонарними магазинами.

Результатом є зміна споживчих звичок українців, перш за все в великих містах України.

Варто зазначити, що насправді інтернет торгівля не ліквідує традиційну торгівлю, а може її доповнювати. Тепер, бізнес без інтернет представлення є вагомим маркетинговим мінусом.

Актуальність теми.

Питання з торгівлі авто та їх пригону з Європи зараз набирає все більше популярності. Сучасний ринок України ще не є повністю заповненим в даній сфері, існує лише декілька основних торгових платформ, які можливо не повністю задовільняють всі потреби українців. Всі вони є однотипними і нічим не відрізняються між собою.

Можливо дані торгівельні системи виконують свою роботу в поставленій задачі, але потрібно подивитись на ринок з іншого боку і можна побачити, що даній сфері ще є куди рости. Потрібно створити нову платформу, яка дає більше можливостей та опцій для продажу та купівлі автомобільних засобів.

Об’єкт дослідження: платформа для продажу авто.

Предмет дослідження: розробка платформи для продажу авто.

Методи дослідження: збір інформації та статистики використання сучасних торгівельних інтернет платформ. Їх взаємодія із користувачем.

Мета дипломного проекту: розробка платформи для зручної торгівлі автомобілями.

Актуальність дослідження: оскільки онлайн торгівля зараз активно розвивається, а онлайн ринок автомобілів не є надто переповненим та має можливості для росту, то важливо розробити нову платформу, яка зможе виділитись своїм функціоналом та охопити більшу аудиторію користувачів.

Прогнозовані результати: повноцінна платформа для онлайн торгівлі автомобілями з широкими можливостями.

Для досягнення мети дипломної роботи поставлено такі завдання:

1. ознайомитися із специфікою даної сфери;
2. провести діагностику вже наявних платформ;
3. на основі отриманих даних розробити нову інтернет платформу для торгівлі автомобілями із додатковими можливостями.

Завдання проекту: розробити онлайн платформу для продажу та замовлень автомобілів.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1 Аналіз ринку автомобілів в Україні

В 2018р. Україна виявилася єдиним ринком в Центральній і Східній Європі, де був зафіксований спад продажів нових автомобілів. В цей час купівля користованих автомобілів була в 3 рази більшою за купівлю нових. Отже відбувся вагомий ріст популярності користованих авто.

Оскільки стан українських доріг переважно не є задовільним, то люди бояться перекупляти авто з рук в руки, адже навіть у практично новому авто може бути багато проблем із роботою, то і про користовані автомобілі не може йти мови.

Багато людей бачать один достойний вихід із ситуації – замовлення авто із Європи.

Причини, по яких авто варто купляти в Європі:

1. кращий технічний стан (невідомо, чи тому, що у Європі дійсно стан доріг набагато кращий, чи тому що там власники міняють авто частіше);
2. повне гарантійне обслуговування в дилерів протягом життя автомобіля;
3. можлива додаткова гарантія після діагностики;
4. ціна нижча за ціну в Україні за те саме авто.

Відомо, що окрім того, що авто із сусідних західних країн будуть працювати довше за наші, також ціни на них значно нижчі. Навіть ціна, включаючи авто та послуги брокера, буде значно меншою за купівлю такого ж авто на авторинку України.

У процесі купівлі автомобіля з Європи можна виділити такі етапи:

1. замовник вибирає авто на одному із європейських сайтів;
2. спеціалісти перевіряються це авто, домовляються з власником і купляють його;

3. спеціалісти приганяють авто із Європи;
4. процес оформлення документів.

## 1.2 Ризики купівлі автомобілів

Варто звернути увагу на те, що при купівлі користованого авто виникає багато ризиків.

Ось деякі із них:

1. неправильна оцінка вартості автомобіля;

Часто людина купує авто під дією емоцій, тому віддає за нього більше ніж воно коштує насправді. Авто може бути ідеальним по зовнішньому стану, на ній немає ніяких видимих дефектів. Але через деякий час користування з нею починаються проблеми. Людина розуміє, що переплатила за товар.

Тому при купівлі авто варто зрозуміти які параметри людині потрібні і яку суму вона готова заплатити. Найкращим варіантом для оцінки авто є відповідні спеціалізовані веб-сайти, оскільки оцінити авто з першого погляду на ринку неможливо.

2. технічний стан автомобіля;

На відкритому ринку ніхто не зможе ідеально провірити авто. Тому дуже багато автомобілів із значними дефектами відправляються на ринки. Спеціалісти технічного маскування з легкістю сховають всі недоліки. За статистикою всі проблеми авто починаються через місяць після його використання. Як наслідок – дорогий ремонт автомобіля аж до повної заміни його двигуна.

3. скручений пробіг;

В нашій країні це поширена практика, оскільки таке порушення ніяк не штрафується і не наказується. Доказати, що пробіг знизила конкретна людина

практично неможливо. Зазвичай такі махінації проводять з машинами, які працювали на таксі. Основна проблема таких автомобів – виснаженість запчастин, оскільки після значного пробігу починаються проблеми із комплектуючими.

4. ремонт корпусу і аварійність;

Якщо автомобіль побував в аварії, то він коштує дуже дешево, а якщо корпус автомобіля був пошкоджений критично, то використовувати машину заборонено. Проте недобросовісні продавці можуть її відновити і продавати по собівартості.

Проблемою таких авто є навіть не втрата грошей, а повна відсутність безпеки автомобіля. При зіткенні ніякого захисного механізму вже не буде, а всі складові корпусу автомобіля вже втратили свою міцність. Тому це може нести серйозну загрозу життю пасажирів.

5. система безпеки.

Цей пункт прямо випливає із попереднього. Навіть при не сильному ударі подушки безпеки можуть «вистрілити. Можна замаскувати відповідні місця і покупець не зможе дізнатись, що в автомобілі відсутня система безпеки.

Як застрахувати себе від проблем при купівлі автомобіля:

1. провести авто через «ланцюжок» із сервісів і спеціалістів для повної перевірки всіх його систем;
2. купляти авто в юридичних осіб;
3. купляти авто у людей, яким можна довіряти.

Третій пункт можна дуже легко реалізувати на платформі для продажу за допомогою сервісу для оцінювання продавця. Тобто людина, яка вже купила авто може виставити рейтинг продавцю. Така система дасть потенційному покупцю зрозуміти, кому можна довіряти, а кому ні.

Варто відмітити, що дизайн системи повинен бути розумним. Тобто першим важливим пунктом є перевірка чи має взагалі даний користувач право

оцінювати цього продавця. Логічно, що право є у людей, які здійснили купівлю. Наступним пунктом є те, що якщо людина вже проголосувала, то система повинна розуміти, що її голос зараховано і у випадку, якщо людина голосує ще раз, то цей голос просто треба змінити на новий.

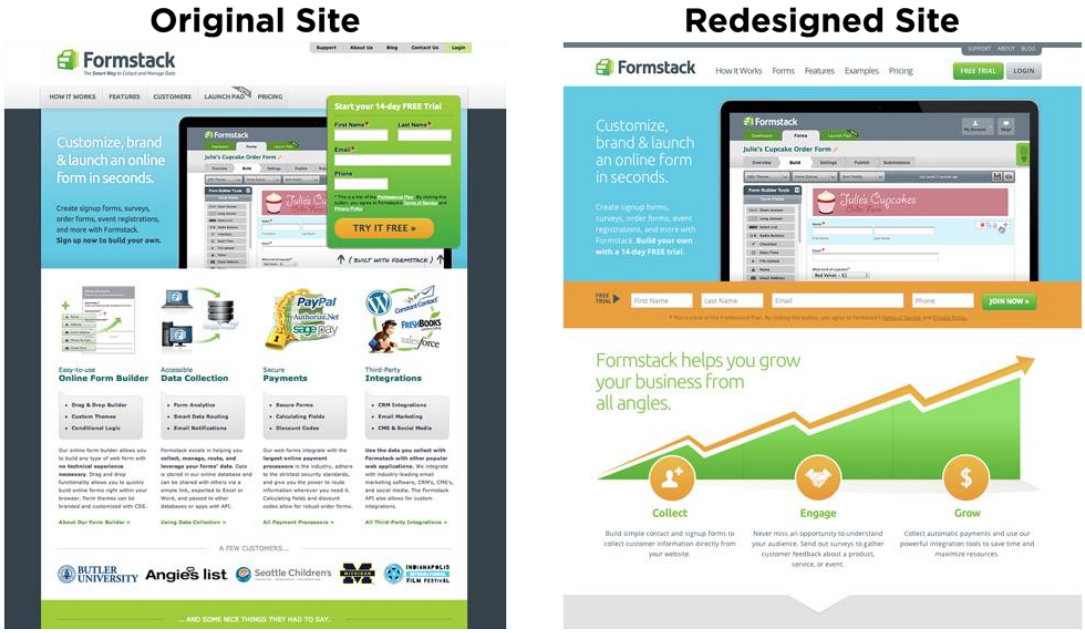
### 1.3 Чинники впливу на клієнта

Важливим критерієм будь-якого веб-застосунку є його дизайн.

Багато власників комерційних веб-сайтів дуже часто недооцінюють значення дизайну для успіху свого бізнесу. SEO-спеціалісти теж не поспішають їх в цьому переконувати, оскільки дизайн ніяк не впливає на пряму індексацію сайту, тобто на йому позиції в пошукових системах, а отже поспіху переробляти його немає.

Цей фактор дійсно не є важливим для пошукових машин, але ось на продажах цей фактор позначається справжнім чином. Причому інколи на дизайн варто звернути найбільше увагу і присвоїти цій роботі найвищий пріоритет.

Справа в тому, що гарний і якісний дизайн викликає найращі позитивні емоції в користувачів сайту. Некрасивий і застарівший дизайн робить наоборот – викликає в користувача велике бажання якнайшвидше закрити вкладку браузера із сайтом. Приклад редизайну сайту зображений на рисунку 1.1.




Рисинок 1.1 - Приклад повного редизайну веб-сайту

Особливо важливу роль дизайн відіграє для торгової інтернет платформи, оскільки користувач заходить на неї із бажанням щось купити. Якщо зовнішній вигляд сайту при цьому відштовхує і не викликає довіри, то купівлю швидше за все він не зробить, а значить всі зусилля, спрямовані на заохочення такого відвідувача були даремними.

Отже головна ціль дизайну – зробити так, щоб веб-сайт ідеально виконував свою основну ціль.

Оновлення дизайну сайту, за умови грамотної реалізації, забезпечує цілий ряд переваг:

1. підвищення лояльності з боку існуючої аудиторії і нових відвідувачів;
2. збільшення коефіцієнта конверсії;
3. зростання загального рівня продажів товарів або послуг;
4. зниження показника відмов і покращення поведінкових факторів;
5. відповідність сайту сучасним технологіям верстки і тенденціям в дизайні;
6. оптимізація структури проекту.

Ще одним важливим фактором успіху сайту є його адаптивність. Багато сайтів створювалися ще дуже давно і про адаптивність тоді ще ніхто не замислювався. Але часи змінилися, і те що раніше розглядалося як приємний бонус – зараз є нагальною потребою.

Зараз частка трафіку з мобільних пристроїв стрімко зростає(рис. 1.2) і в ряді категорій вже перевищує переходи із десктопних комп'ютерів. Це дуже важливий аргумент, чому варто забезпечити якісне відображення сайту для усіх пристроїв.

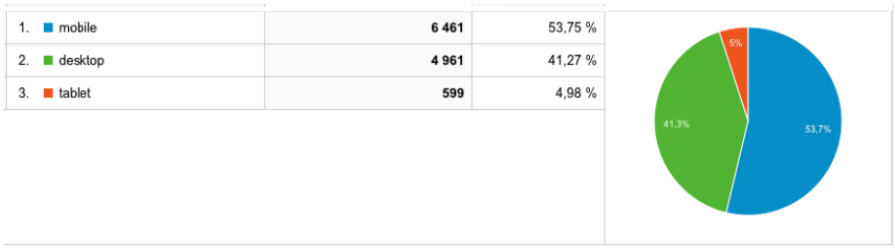



Рисунок 1.2 - Статистика відривання
сайтів з різних пристроїв

Адаптивна верста – обов'язкова умова, якій повинен відповідати кожний сучасний дизайн веб-сайту. На екрані смартфона чи планшета неадаптивний веб-сайт не лише не гарно виглядає, а ще ним дуже незручно, а інколи практично неможливо користуватися. Приклад зображений на рисунку 1.3.


Рисунок 1.3 - Приклади неадаптивного
та адаптивного сайтів

Сучасна верстка, адаптована під всі типи пристроїв вирішує цю проблему, дозволяючи не втрачати прибуток.

Окрім того пошукова система «Google» повідомила, що з літа 2018р. неадаптовані під мобільні пристрої веб-сайти, ранжируватимуться нижче у пошуку.

Також важливим фактором у використанні сайту є:

1. швидкість відкривання сторінок;
2. анімація переходів між сторінками;
3. наявність карти сайту;
4. дотримування правил SEO[14];
5. вік сайту.

   Чим старшим є ресурс, тим вище він буде індексуватись в пошукових системах.

### 1.4 Дослідження правильної архітектури веб-сторінки

Згідно з дослідженнями, більшість людей, перебуваючи на веб-сайті слідують типовим сценаріям поведінки. Ці сценарії змінюються в залежності від розташування і зовнішнього вигляду елементів інтерфейсу сайту. Значить, керуючи ними, можна впливати на поведінку користувачів, сприяти утриманню уваги і здійснення цільових дій, наприклад, купівлі або оформлення заявки.

Елементи сайту, які найбільше володіють увагою користувача:

1. логотип компанії. Він традиційно розташовується в шапці ресурсу, і на нього падає перший погляд відвідувача. Його вивчають 6,48 секунди;
2. основне меню майданчика теж привертає увагу людей. На нього витрачають 6,44 секунди;
3. на пошуковий рядок користувачі виділяють близько 6 секунд свого часу;

4. головна картинка, яка зазвичай розташовується на першому екрані, утримує увагу відвідувачів протягом 5,94 секунди;
5. на прочитання тексту, розміщеного на веб-ресурсі, у людей йде 5,59 секунди;
6. віджети соцмереж віднімають 5,95 секунди у користувачів, а в підвалі сайту вони проводять в середньому 5,25 секунди за одне відвідування.

На прикладі сайту auto.ria.com(рис. 1.4) зображено елементи, які викликають найбільший інтерес у відвідувачів ресурсу. В цілому, перше враження про сайт у людей складається всього лише за 0,2 секунди. До такого висновку прийшли дослідники університету технологій та науки в американському штаті Міссурі[4].

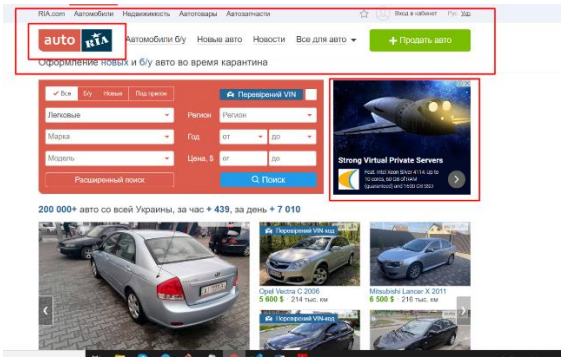



Рисунок 1.4 - Елементи, які привертають найбільше уваги користувачів

Також найкраще привертають увагу фотографії, на яких зображена людина, яка дивиться прямо на користувача сайту. Цей прийом здатний заручитися підтримкою користувача і затримати його погляд.

При цьому дослідження показали, що людям приємніше бачити на веб-ресурсах фотографії звичайних людей, а не представників модельного бізнесу. Такий приклад зображено на рисунку 1.5. Нетематичні зображення теж не варто використовувати, так як вони ігноруються відвідувачами сайту.


Рисунок 1.5 - Приклад зображення,
яке привертає увагу користувача

Варто також розуміти, що є певні ділянки сайту, на які користувач звертає найбільше уваги. Коли людина переходить на сайт, то відразу ж дивиться на верхній кут з лівого боку. Потім його погляд переміщається вправо і трохи вниз.

На рисунку 1.6 показано, які зони на веб-ресурсі приковують до себе увагу відвідувачів. Червоний ділянку - максимум уваги, жовтий - у стані середнього ступеню, зелений - в найменшій.


Рисунок 1.6 - Ділянки, на які користувач
звертає найбільше уваги

Щоб людина прочитала хоча б перший абзац з головним посилом статті, він повинен відрізнятися від решти тексту. Наприклад, збільшити розмір шрифту або застосувати жирне накреслення. Згідно з дослідними даними, такі абзаци привертають до себе увагу 95% користувачів. Але потрібно пам’ятати, що використовувати інший шрифт не можна. Це буде виглядати непрофесійно і викличе відчуття дисгармонії у читача.

Сам текст повинен складатися з коротких рядків, які розташовуються в одну колонку. Не потрібно забувати про абзаци, посилання і візуальне виділення важливих моментів. Текстовий контент повинен бути лаконічним, особливо, якщо мова йде про головну сторінку ресурсу. Ніхто не любить читати довгі тексти.

Найчастіше, користувачі вивчають контент ресурсу по F-патерну. Що це означає? Що люди звертають свою увагу на ті частини сторінки, які при змалюванні нагадують букву F(рис. 1.7).

Рисунок 1.7 - «Патерн F» - елементи, на які відвідувачі звертають найбільше уваги

Цей патерн складається з двох горизонтальних ліній і однієї вертикальної. Щоб донести все необхідне до читачів, необхідно інформацію розмістити вздовж верхньої або нижньої горизонтальних ліній.

Практично будь-який веб-ресурс має навігаційне меню. Найбільшою увагою при цьому заволодіває те меню, яке розташовується у верхній частині сторінки.

Для того, щоб зацікавити користувача, необхідно використовувати великі заголовки. Якщо при цьому вони розташовуються в лівому верхньому кутку, то на них гарантовано звернуть увагу користувачі. Необхідно постаратися тримати лише один великий заголовок, оскільки при їх множинності, користувач розсіює свою увагу і читає лише їх ліві частини. Проте, якщо інформація заголовків дійсно корисна, то він прочитає їх до кінця.

Часу на прочитання заголовка мізерно мало. Зазвичай це менше однієї секунди. Тому важливо з перших же слів зацікавити читача, щоб не втратити його увагу.

### 1.5 Методи оцінювання наявних онлайн платформ

На даний момент існує багато різних платформ, тому для розробки нової платформи потрібно якось їх оцінити. Для оцінки потрібно використовувати не лише їх зовнішній вигляд і конкурентоспроможність з боку SEO, а також швидкість їх роботи.

Зрозуміло, що для вимірювання швидкості потрібно використовувати якийсь надійний та високоефективний сервіс, якому можна довіряти та на основі його вимірювань робити якісь висновки стосовно свого проекту.

Для даного проекту було вирішено використати сервіс «Page Speed Insights» або як його ще називають «Google Page Speed».

Безкоштовний онлайн-сервіс Google Page Speed Insights – це комплексний інструмент для визначення фактичної продуктивності і вибору ефективних шляхів оптимізації сайту. Зручний для використання як на комп'ютері, так і на мобільному пристрої. Не показує абсолютну швидкість сторінки, а аналізує ефективність динаміки завантаження і відтворення в браузері клієнта. При цьому сервіс вираховує як сайт завантажується при різному рівні інтернет з’єднання. Вимірює швидкість завантаження JavaScript,

CSS, усіх картинок. Перевіряє структуру HTML, налаштування шрифтів на сайті, конфігурацію веб-сервера.

Також сервіс дає поради стосовно оптимізації швидкості загрузки веб-додатку, як от оптимізація картинок, вибір їх валідного формату, використання JS та CSS коду.

### 1.6 Аналоги проекту

На даний момент в Україні існує декілька популярних платформ для купівлі автомобілів.

Рейтинг найпопулярніших сайтів з продажу автомобілів очолює добре відомий кожному автолюбителю інтернет-майданчик «AutoRia». На сторінках онлайн ресурсу в основному представлені користовані автомобілі та запчастини. Бажаючі придбати новеньку машину теж зможуть без зусиль знайти для себе необхідний варіант.

Дизайн цього додатку є досить сучасним і адаптивним. Проте є в цього додатку і вагомий мінус. Сервіс «Google page speed» показує лише 50 балів зі 100 по швидкості для мобільних телефонів, що зображено на рисунку 1.8. Такий показник не є високим і класифікується, тому даний сайт можна вважати додатком середньої швидкості.

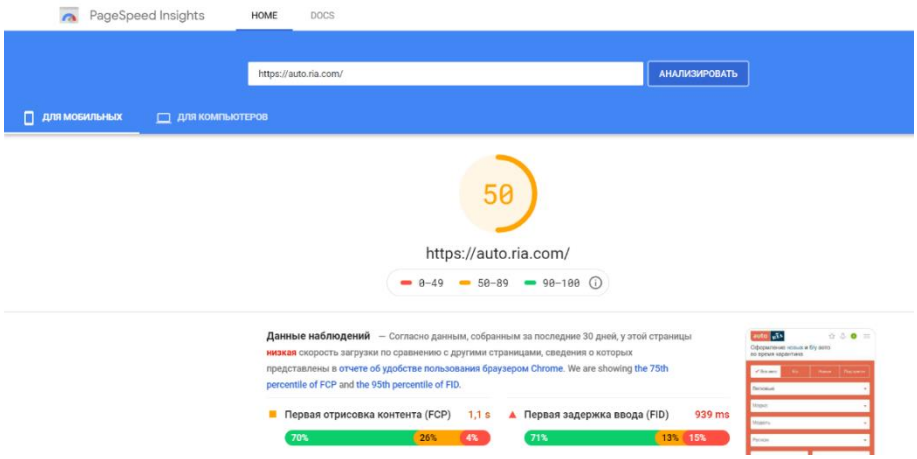



Рисунок 1.8 - Вимірювання швидкодії веб-сайту Autoria
За допомогою «Google page speed» сервісу

Другим за популярністю майданчиком, на якому можна купити авто в Україні, є сайт RST(рис. 1.9). Тут також в основному представлені користовані машини, але можна придбати і зовсім нові авто. На відміну від лідера рейтингу, на цьому інтернет ресурсі немає можливості придбати запчастини.

Даний веб-сайт показує 60 балів по швидкості, що є непоганим результатом, але дизайн сайту є вже застарілим і виглядає не дуже привабливо на фоні інших додатків.

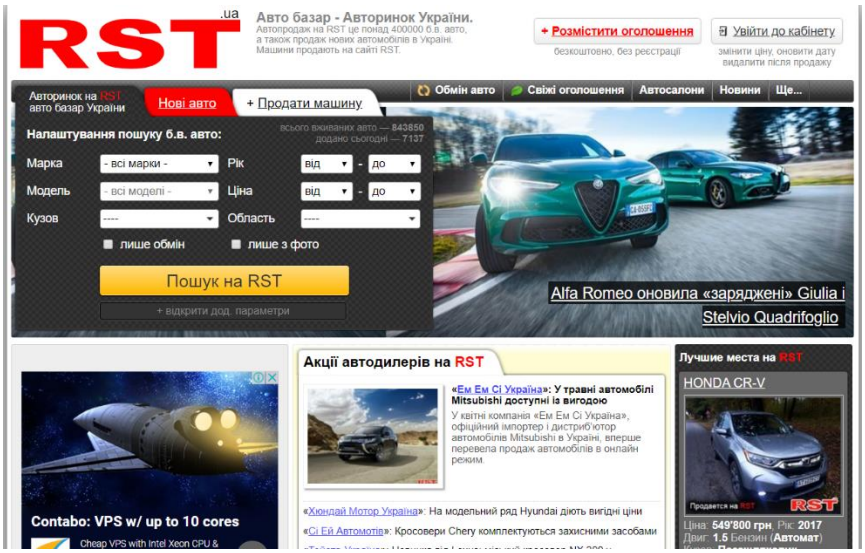



Рисунок 1.9 - Дизайн сайту RST

Трійку лідерів сайтів, де можна купити або продати авто, замикає інтернет-майданчик «Автобазар». Кожен бажаючий може скористатися практично повним спектром послуг - від продажу, обміну або купівлі нових машин до придбання запчастин і машин за розбирання.

Даний сервіс має досить гарний дизайн та показує 60 балів швидкості.

## 1.7 Постановка задачі

Ціллю даної дипломної роботи є розробка бекенд частини для платформи з пригону автомобілів з Європи та їх продажу. Окрім цього потрібно розробити функціонал, який дозволяє третім людям виступати у ролі рефералів в продажі

чужих авто. Дана система буде виконувати функції основного майданчика для ведення комерційної діяльності.

Реалізація поставленої мети передбачає вирішення наступних задач:

1. вибір стеку технологій для реалізації продукту;
2. розробка архітектури бази даних;
3. розподілення всього проекту на основні модулі;
4. реалізація авторизації;
5. реалізація функціоналу відображення всіх продуктів зі списку;
6. реалізація можливості оцінювання користувача;
7. реалізація реферальної системи;
8. можливість відправити запит за купівлю автомобіля;
9. можливість виставити оголошення на замовлення автомобіля;
10. можливість відправити запит на реалізацію замовленого авто.

### 1.8 Актуальність проекту

Розробка сучасної платформи для продажу автомобілів є дуже актуальною в даний час, оскільки купівля користованих авто зараз є як ніколи затребуваною.

Оскільки на даний момент на ринку вже існує декілька популярних сервісів для продажу авто, необхідно створити новий сервіс, який буде виділятися на фоні інших.

Необхідно розробити швидкісний додаток, який зможе швидко завантажуватись на усіх платформах, навіть при поганому інтернет з’єднанні. Повинна існувати можливість продавати авто за допомогою реферальної системи, тобто посторонні люди зможуть допомогти реалізувати товар і отримати за це винагороду, вказану власником даного лоту.

Окрім того можна робити замовлення на пригон авто з боку покупця. А також з боку продавців, давати запити на реалізацію схожого за характеристиками чи такого ж авто.

# 2 АНАЛІЗ АРХІТЕКТУРИ ТА ЗАСОБІВ РОЗРОБКИ

## 2.1 Поняття бекенд розробки

Мови бекенд розробки виконують роботу "поза сценою" для веб-додатків. Це частина проекту, яка підключає додаток до бази даних, керує з'єднаннями користувачів та надає можливість працювати самому веб-додатку. Бекенд частина проекту працює в парі з фронтенд частиною для того, щоб доставити робочий продукт до кінцевого користувача.

Бекенд - це технологія, необхідна для обробки вхідного запиту, генерації та відправлення відповіді клієнту. Зазвичай бекенд включає в себе 3 основні частини:

1. сервер. Це комп'ютер, який приймає запити;
2. додаток. Це програма, запущена на сервері, яка слухає запити, отримує інформацію з бази даних та надсилає відповідь;
3. база даних. Бази даних використовуються для організації та збереження даних.

Сервер - це просто комп'ютер, який слухає вхідні запити. Його принцип роботи зображено на рисунку 2.1. Хоча існують машини, виготовлені та оптимізовані саме для цієї мети, будь-який комп'ютер, підключений до мережі, може виконувати роль сервера. Насправді власний комп’ютер дуже часто використовується як сервер для розробки додатків.

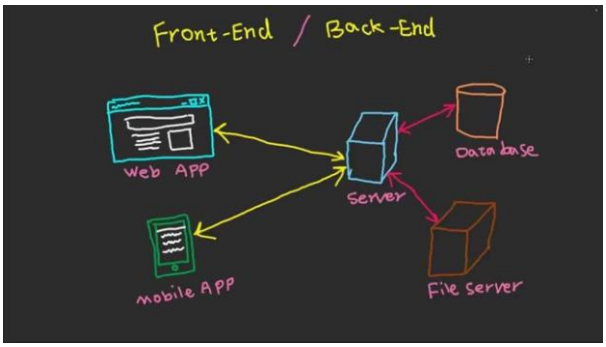



Рисунок 2.1 - Принцип роботи дії веб-додатку

Сервер запускає додаток, який містить логіку щодо відповіді на різні запити на основі методу HTTP та Уніфікованого ідентифікатора ресурсу (URI). Протокол HTTP та URI називаються маршрутом, а їх відповідність на основі запиту називається маршрутизацією.

Деякі з цих функцій обробника будуть проміжними програмами. У цьому контексті проміжне програмне забезпечення - це будь-який код, який виконується між сервером, який отримує запит і надсилає відповідь. Ці функції програмного забезпечення можуть змінювати об'єкт запиту, запитувати базу даних або іншим чином обробляти вхідний запит. Функції проміжного програмного забезпечення зазвичай закінчуються передачею контролю на наступну функцію програмного забезпечення, а не надсиланням відповіді.

Врешті-решт буде викликана функція, яка закінчує цикл відповіді на запит, надсилаючи відповідь HTTP клієнту.

Щоб опублікувати веб-сайт, необхідний статичний або динамічний веб-сервер.

Статичний веб-сервер, або стек, складається з комп'ютера з сервером HTTP (ПО). Такий тип називається «статикою», тому що сервер посилає розміщені файли в браузер «як є».

Динамічний веб-сервер складається з статичного веб-сервера і додаткового програмного забезпечення, найчастіше сервера додатку і бази даних. Він називається «динамічним», тому що такий сервер змінює вихідні файли перед відправкою в браузер по HTTP.

Наприклад, для отримання підсумкової сторінки, яку користувачі переглядають в браузері, програмне забезпечення сервера може заповнити необхідний HTML-шаблон даними з бази даних. Такі сайти, як MDN або Вікіпедія, складаються з тисяч веб-сторінок, але вони не є реальними HTML документами - лише кілька HTML-шаблонів і гігантські бази даних. Ця структура спрощує і прискорює супровід веб-додатків і доставку контента.

## 2.2 Адресний простір

URL (або URL адреса) - це унікальна форма адресу конкретного веб-ресурсу в мережі Інтернет. Вона може посилатися на веб-сайт, якийсь конкретний документ або зображення. Користувачеві Інтернету потрібно вставити цей рядок в поле пошуку, щоб знайти потрібний сайт, документ, папку або зображення. Простою мовою це означає наступне: завдяки URL адресі користувач дізнається інформацію про те, де знаходяться потрібні йому дані.

URL адреса - це абревіатура, що позначає термін Universal Resource Locator (загальний вказівник ресурсу). Вона містить посилання на сервер, який є сховищем шуканого ресурсу. Загалом, URL це шлях з сервера до останнього пристрою (який є платформою роботи користувача). Структуру URL зображено на рисунку 2.2.

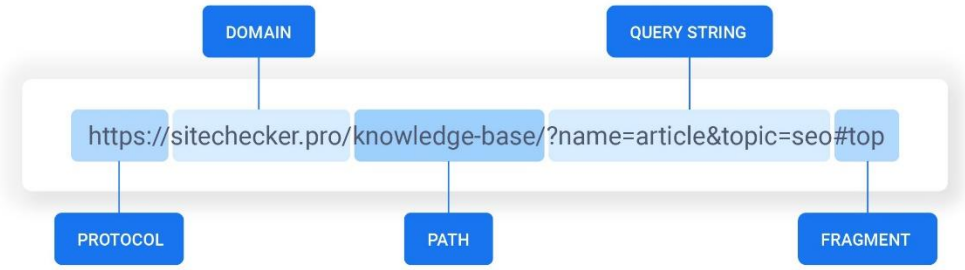



Рисунок 2.2 - Типова будова URL адреси

URL адреса має певну структуру, яка включає:

1. метод доступу до ресурсу, який також називається мережевим протоколом;
2. авторизацію доступу;
3. хости - DNS адресу, яка вказана як IP адреса;

4. порт - ще одна обов'язкова деталь, яка включається в поєднання з IP адресою;
5. трек - визначає інформацію про метод отримання доступу;
6. параметр - внутрішні дані ресурсу.

HTTP (від англ. HyperText Transfer Protocol - протокол передачі гіпертексту) - це прикладний протокол передачі даних в мережі[6, 7]. Схему його роботи зображено на рисунку 2.3. На поточний момент використовується для отримання інформації з веб-сайтів. Протокол HTTP заснований на використанні технології «клієнт-сервер»: клієнт, що відправляє запит, є ініціатором з'єднання; сервер, який отримує запит, виконує його і відправляє клієнту результат.

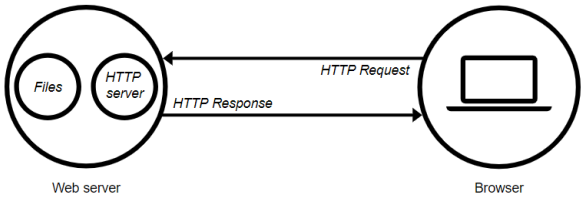



Рисунок 2.3 - Схема роботи HTTP протоколу

HTTPS (від англ. HyperText Transfer Protocol Secure - безпечний протокол передачі гіпертексту) - це розширення протоколу HTTP, що підтримує шифрування за допомогою криптографічних протоколів SSL і TLS.

Різниця між HTTP і HTTPS:

1. HTTPS не є окремим протоколом передачі даних, а являє собою розширення протоколу HTTP з надбудовою шифрування;
2. передаються по протоколу HTTP дані не захищені, HTTPS забезпечує конфіденційність інформації шляхом її шифрування;
3. HTTP використовує порт 80, HTTPS - порт 443.

Забезпечення захищеної передачі даних необхідно на сайтах, де вводиться і передається конфіденційна інформація (особисті дані користувачів, деталі доступу, реквізити платіжних карт) - на будь-яких сайтах

з авторизацією, взаємодією з платіжними системами, поштовими сервісами. Шифрування таких даних дозволить запобігти їх отримання і використання третіми особами.

Для реалізації передачі даних за допомогою HTTPS на веб-сервері, що обробляє запити від клієнтів, повинен бути встановлений спеціальний SSL-сертифікат. Є сертифікати, що захищають тільки один домен. А є сертифікати, які забезпечують захист інформації на всіх піддоменах, і це Wildcard SSL. Шифрування відбувається в обидві сторони - як даних, отриманих клієнтом, так і даних, відправлених на сервер.

Наявність SSL-сертифіката є одним з факторів ранжирування Google, тому перехід на протокол HTTPS підвищує позиції в пошуковій видачі Google. Хоча на поточний момент цей фактор не має основного значення, його вплив на пошуковий результат може збільшуватися в майбутньому.

Отже, що таке SSL? Клієнт робить замовлення на сайті. Щоб оплатити товар, він вводить дані кредитної картки. Коли замовлення оформлене, інформація відправляється на веб-сервер. На цьому етапі її можуть перехопити шахраї.

Якщо на сайті є SSL-сертифікат, між браузером клієнта та сайтом встановлюється захищене з'єднання. В цьому випадку браузер спочатку перетворює номер карти в деякий набір символів і тільки потім відправляє його на сервер. Розшифрувати повідомлення вийде тільки спеціальним ключем, який зберігається на сервері. Якщо шахраї і перехоплять інформацію, вони не зрозуміють, що вона означає. На рисунку 2.4 зображено наявність SSL у різних браузерах.


Рисунок 2.4 - Зображення сайту з SSL-шифруванням в популярних веб-браузерах

## 2.3 Поняття маршрутизації

У веб-розробці, процес, який відповідає за визначення обробника для конкретної запитуваної сторінки, називається маршрутизація. Найчастіше говорять «роутинг».

На рисунку 2.5 кожна адреса зображує конкретний маршрут (роут). Причому, їх можна розділити по типу: статичні і динамічні.

```
https://test.io/code_reviews/4172
https://test.io/courses/programming-basics
https://test.io/account/profile/edit
```

Рисунок 2.5 - Приклади маршрутизації

Статичні маршрути характеризуються тим, що адреса збігається з самим маршрутом. Наприклад, «account/profile/edit». Незважаючи на те, що адреса одна, у різних користувачів вона буде відображати різні дані, що залежить від того, хто зараз авторизований.

А що, якщо у нас є адреси, які позначають одне і теж, але містять параметр. Типовий приклад «/users/5». Без особливих зусиль можна зрозуміти, що за цим

посиланням ми отримаємо інформацію про користувача з номером 5. Але тоді виникає питання: якщо у нас в базі тисячі користувачів, нам доведеться визначати тисячі маршрутів?

У даному випадку можна використати регулярні вирази. Створюється один маршрут, який виглядає приблизно так: «^/users/(\w+)». А далі потрібно просто порівняти цей регулярний вираз з рядком запиту. Отже визначено якийсь один конкретний маршрут, який покриває багато йому схожих запитів.

## 2.4 Взаємодія клієнт-сервер

Дані, які сервер надсилає назад, можуть надходити в різних формах. Наприклад, сервер може давати відповідь у вигляді HTML-файлу, надсилати дані як JSON, а також може надсилати назад лише код статусу HTTP.

Що таке код статусу? Це число з 3 цифр(рис. 2.6), яке видає сервер на запит від клієнта. В залежності від статусу запиту, вирішується яким чином клієнт буде його обробляти. Статус відповіді завжди містить свій унікальний код і окрему коротку інструкцію на анлійській мові, яка відділена від коду знаком пробілу.

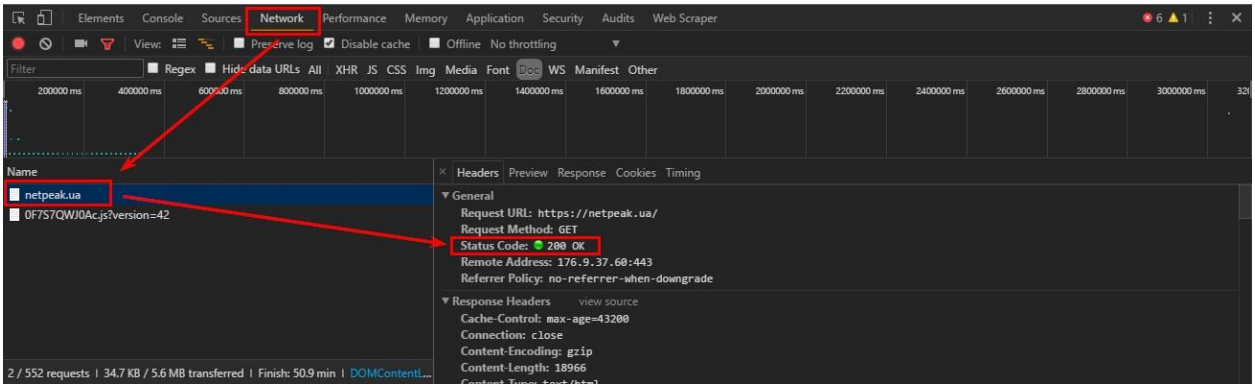



Рисунок 2.6 - Приклад типової відповіді від сервера

Код відповіді від сервера при зверненні до якогось маршруту у першу чергу перевіряються браузерами, і роботами пошукових систем.

Наприклад, неіснуючі сторінки завжди повинні віддавати 404 код. Чому? Google стверджує, що сторінки з кодом відповіді, відмінним від 404 і 410, будуть сканuватися. Тому, якщо не знати, які коди віддають сторінки під час запиту, то можна поставити під загрозу весь проект.

Після того як клієнт звернувся до сервера, серверу може потребуватись звернутись до бази даних. База даних постачає інтерфейс, який дозволяє постійно зберігати дані в особливому порядку. Зберігання даних в базі даних зменшує навантаження на основну пам’ять комп’ютера та дозволяє їх зберегти, якщо головний сервер виходить із ладу. Існує різних типів звернення до бази даних. Клієнт може ініціювати запит на отримати інформації, редагування, чи додавання.

Звернення клієнта до сервера відбувається за допомогою API.

API - це сукупність чітко визначених методів зв'язку між різними компонентами програмного забезпечення.

Більш конкретно, веб-API - це інтерфейс, створений бекендом: множина кінцевих точок, за допомогою яких отримується доступ до конкретних ресурсів.

Веб-API визначається типом запитів, які він може обробляти, а також маршрутами, які він визначає, і типом відповідей, які клієнти можуть очікувати після потрапляння на ці маршрути.

Один веб-API може використовуватися для надання даних для різних фронтенд версій додатків. Наприклад до одного API можуть звертатися декілька HTML-сторінок та мобільний додаток і всі вони будуть отримувати однакові відповіді від сервера.

Існує 2 основних стиля API: SOAP і REST.

SOAP - це стандартизований протокол, який відправляє повідомлення з використанням інших протоколів, таких як HTTP і SMTP. Специфікації SOAP є офіційними веб-стандартами, які підтримуються і розробляються Консорціумом World Wide Web (W3C). На відміну від SOAP, REST - це не

протокол, а архітектурний стиль. Архітектура REST встановлює набір рекомендацій, яких необхідно дотримуватися, якщо ми хочемо надати RESTful веб-службу. Наприклад, існування без збереження стану і використання кодів стану HTTP.

Оскільки SOAP є офіційним протоколом, він поставляється зі строгими правилами і розширеними функціями безпеки, такими як вбудована сумісність з ACID і авторизація. Більш висока складність вимагає більшої пропускної здатності і ресурсів, що може привести до зниження часу завантаження сторінки. REST був створений для вирішення проблем SOAP. Тому у нього більш гнучка архітектура. Він складається тільки з простих рекомендацій і дозволяє розробникам реалізовувати рекомендації по-своєму. Він допускає різні формати повідомлень, такі як HTML, JSON, XML і простий текст, в той час як SOAP допускає тільки XML. REST також є легшою архітектурою, тому веб-сервіси RESTful мають більш високу продуктивність. Через це він став неймовірно популярним в епоху мобільних пристроїв, де навіть кілька секунд мають велике значення (як за часом завантаження сторінки, так і за доходами).

REST розшифровується як передача репрезентативного стану. Це архітектурний стиль, який визначає набір рекомендацій для розробки слабозв’язаних додатків, що використовують протокол HTTP для передачі даних. REST не вказує строго, як реалізувати принципи на більш низькому рівні. Замість цього, рекомендації REST дозволяють розробникам реалізовувати деталі у відповідності зі своїми потребами. Веб-сервіси, створені відповідно до архітектурного стилю REST, називаються веб-сервісами RESTful.

Основні принципи REST:

1. уніфікований інтерфейс - запити від різних клієнтів повинні виглядати однаково, наприклад, один і той же ресурс не повинен мати більше одного URI;

2. поділ клієнт-сервер - клієнт і сервер повинні діяти незалежно один від одного. Вони повинні взаємодіяти один з одним тільки через запити і відповіді;
3. чистота - не повинно бути ніяких сеансів на стороні сервера. Кожен запит повинен містити всю інформацію, яку сервер повинен знати;
4. інформація про кешування - відповіді сервера повинні містити інформацію про те, кешуються відправлені ним дані чи ні. Ресури, які закешувалися повинні надходити з номером версії, щоб клієнт міг не запитувати одні й ті ж дані більш ніж один раз;
5. багаторівнева система. Між клієнтом і сервером, який повертає відповідь, може бути кілька рівнів серверів. Це не повинно впливати ні на запит, ні на відповідь;
6. код за запитом (необов'язково) – коли це необхідно, відповідь може містити виконуваний код (наприклад, JavaScript у відповіді HTML), який може виконати клієнт.

Архітектура REST дозволяє постачальникам API доставляти дані в різних форматах, таких як простий текст, HTML, XML, YAML і JSON, що є однією з його найпопулярніших функцій. Завдяки зростаючій популярності REST, JSON формат, який легко читається людиною також швидко завоював популярність, так як він відмінно підходить для швидкого і якісного обміну даними.

JSON розшифровується як JavaScript Object Notation. Це простий для аналізу і легкий формат обміну даними. Незважаючи на свою назву, JSON повністю незалежний від мови, тому його можна використовувати з будь-якою мовою програмування, а не тільки з JavaScript. Його синтаксис є підмножиною стандарту ECMA-262, 3-е видання. Файли JSON складаються з наборів пар ім'я/значення і упорядкованих списків значень, які є універсальними структури даних, що використовуються більшістю мов програмування. Тому

JSON може бути легко інтегрований з будь-якою мовою. Типовий приклад JSON зображено на рисунку 2.7.

```
{
   "username" : "my_username",
   "password" : "my_password",
   "validation-factors" : {
      "validationFactors" : [
         {
            "name" : "remote_address",
            "value" : "127.0.0.1"
         }
      ]
   }
}
```

Рисунок 2.7 - Приклад JSON-формату

SOAP означає простий протокол доступу до об'єктів. Це протокол обміну повідомленнями для обміну даними в децентралізованому і розподіленому середовищі. SOAP може працювати з будь-яким протоколом прикладного рівня, таким як HTTP, SMTP, TCP або UDP. Він повертає дані одержувачу в форматі XML(рис. 2.8). Безпека, авторизація та обробка помилок вбудовані в протокол і, на відміну від REST, не передбачають прямого зв'язку точка-точка. Тому він добре працює в розподіленому корпоративному середовищі.

```
<?xml version="1.0" encoding="UTF-8"?>
<authentication-context>
   <username>my_username</username>
   <password>my_password</password>
   <validation-factors>
     <validation-factor>
       <name>remote_address</name>
       <value>127.0.0.1</value>
     </validation-factor>
   </validation-factors>
</authentication-context>
```

Рисунок 2.8 - Приклад XML-формату

SOAP слідує формальному і стандартизованому підходу, який визначає, як кодувати XML-файли, які повертаються за допомогою API. Насправді

повідомлення SOAP - це звичайний XML-файл, що складається з наступних частин:

1. конверт (обов'язково) - це початковий і кінцевий теги повідомлення;
2. тема (необов'язково) - містить необов'язкові атрибути повідомлення. Це дозволяє розширювати повідомлення SOAP модульним і децентралізованим способом;
3. тіло (обов'язково) - містить дані XML, які сервер передає одержувачу;
4. несправність (необов'язково) - містить інформацію про помилки, що виникли при обробці повідомлення.

В теперішній час, ймовірно, SOAP продовжить використовуватися для веб-сервісів рівня підприємств, які вимагають високого ступеня безпеки і складних транзакцій. API-інтерфейси для фінансових послуг, платіжних шлюзів, програмного забезпечення CRM, управління ідентифікацією та телекомунікаційних послуг є широко використовуваними прикладами SOAP. Одним з найбільш відомих API-інтерфейсів SOAP є загальнодоступний API-інтерфейс PayPal, який дозволяє приймати платежі через PayPal і кредитні карти, додавати кнопку PayPal на веб-сайт, дозволяти користувачам входити в систему через PayPal і виконувати інші дії, пов'язані з PayPal.

Отже, основні принципи роботи клієнт-сервер:

1. сервер не може ініціювати відповіді без запитів;
2. на кожен запит потрібна відповідь, навіть якщо це лише код статусу 404, який вказує на те, що вміст не знайдено. В іншому випадку клієнт буде нескінченно очікувати відповідь;
3. сервер не повинен надсилати більше однієї відповіді на запит. Це призведе до помилок у коді;

Основні кроки, які необхідні для виконання запиту до сервера:

1. користувач здійснює купівлі в онлайн магазині. Він натискає на зображення обкладинки смартфона, і ця подія клацання робить GET-запит на «/products/66432»;

2. запит користувача пересувається через Інтернет на один із серверів магазину. Це один із повільних кроків у процесі, оскільки запит не може пройти швидше, ніж швидкість світла, і він може мати велику відстань для подорожі. З цієї причини основні веб-сайти з користувачами в усьому світі матимуть багато різних серверів, і вони направлятимуть користувачів на найближчий до них сервер;

3. сервер, який активно слухає запити всіх користувачів, отримує запит користувача;

4. запускаються слухачі подій, які відповідають цьому запиту (HTTP: GET та URI: «/products/66432»). Код, який виконується на сервері в період між запитом і відповіддю, називається проміжним програмним забезпеченням;

5. обробляючи запит, код сервера робить запит до бази даних, щоб отримати більше інформації про цей продукт. База даних містить всю іншу інформацію, яку користувач хоче знати про цей конкретний продукт: його найменуванн, ціна продукту, кілька відгуків про товар та рядок, який забезпечить шлях до зображення продукту;

6. запит на базу даних виконується, і база даних надсилає запитувані дані назад на сервер. Варто зазначити, що запити до бази даних є одним із повільних кроків у цьому процесі. Читання та запис із статичної пам'яті відбувається досить повільно, і база даних може знахолитись не на тому комп’ютері, що і оригінальний. Сам цей запит, можливо, повинен пройти через весь інтернет;

7. сервер отримує потрібні йому дані з бази даних, і тепер він готовий створити і відправити свою відповідь назад клієнту. Цей об’єкт відповіді має всю інформацію, необхідну браузеру, щоб показати користувачу більше деталей (ціна, відгуки, розмір тощо) про конкретний випадок, який його цікавить. Заголовок відповіді буде містити код статусу 200 HTTP, щоб вказати, що запит виявився успішним;

8. відповідь подорожує по Інтернету, назад до комп’ютера користувача;

9. веб-переглядач користувача отримує відповідь і використовує цю інформацію для створення та надання перегляду, який він бачить у кінцевому рахунку.

## 2.5 Розробка архітектури додатку

Для розробки торгової платформи автомобілями необхідно розробити схему основних сутностей у проекті.

Для початку опишемо кожну сутність та наділимо її відповідними правами. Головною сутністю на проекті є користувач(рис. 2.9).


Рисунок 2.9 - Часова діаграма мінімальних дій з боку користувача

Функціонал користувача:

1. можливість зареєструватись на сайті;

2. можливість виставити оголошення про продаж;
3. можливість додати опцію «винагорода за допомогу в продажі»;
4. можливість просувати чужі товари в списку своїх(отже інші користувачі при перегляді чужих товарів не зрозуміють які належать даній людині, а які просто просуваються цією людиною);
5. можливість просувати чужі товари по реферальній системі;
6. можливість опублікувати оголошення про замовлення якогось автомобіля;
7. можливість подати заявку на виконання чужого замовлення.

В даному списку функціоналу варто звернути увагу на «опцію про винагороду». Все повинно відбуватись наступним чином:

- користувач виставляє оголошення;
- при заповненні інформації про товар йому висвітлюється опція, яка дозволяє розповсюджувати цей товар іншим людям по реферальній силці за певну винагороду;
- користувач може пропустити дану опцію, в такому випадку його продукт не зможуть додавати у свій список інші користувачі;
- користувач відмічає дану опцію;
- в такому випадку йому необхідно також вказати тип винагороди за допомогу у продажі;
- можливі опції винагороди: певна стала сума, певний відсоток від продажу;
- користувач заповнює всю іншу необхідну інформацію і завершує реєстрацію.

На рисунку 2.10 зображено типові дії з боку продавця. Варто відмітити, що продавцем можна виступати навіть не маючи свого власного товару.


Рисунок 2.10 - Зображення сутності «продавець»

Достатньо знайти товар, який відмічений опцією «винагорода за допомогу в продажі» та додати його у список своїх товарів. Після цього при переході на цей товар зі сторінки саме цього користувача та його купівлі, система зрозуміє, що товар був проданий саме за допомогою цієї людини.

З рисунку 2.11 можна зрозуміти, що з боку функціоналу додатку існує 2 напрямки розвитку дій для покупця. Перший напрямок – користувач знаходить необхідний товар в списку та надсилає запит на купівлю. Варто відмітити, що в запиті користувач також може вказати ціну за яку він готовий зробити купівлю. В такому випадку продавець має вибір: він може погодити дану пропозицію та закінчити торги, або продовжити свій продаж і відмовити даному клієнту.

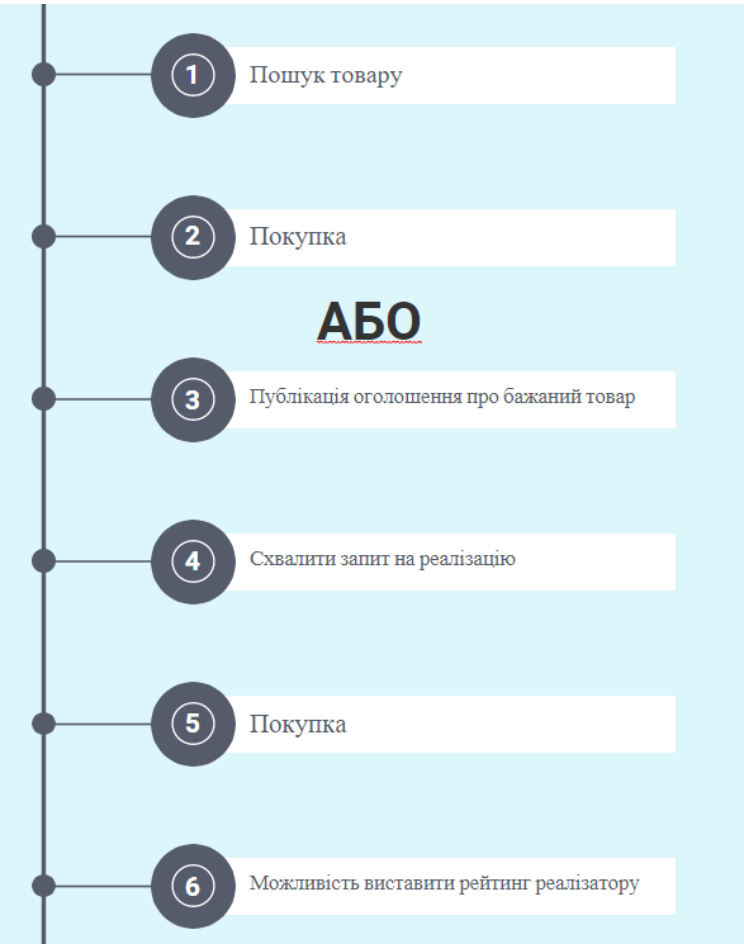



Рисунок 2.11 - Зображення сутності «покупець»

Іншим напрямком для покупця, у випадку, якщо він не знайшов потрібний автомобіль у списку, то у нього є можливість опублікувати своє замовлення на сайті. Він заповнює необхідну інформацію та виставляє своє оголошення. Основними полями в формі є «ціна від» і «ціна до».

Після того, як клієнт виставив своє оголошення, люди які готові реалізувати даний товар надсилають на нього запити. Реалізатори вказують ціну, за яку готові пригнати даний автомобіль, а також дату до якої замовлення повинно бути виконане. Замовник вибирає пропозицію, яка його цікавить та нажимає на кнопку «Підтвердити».

Для того, щоб розробити правильну архітектуру додатку, необхідно структурувати всі сутності, які у нас є. Для структуризації використаємо UML-діаграму.

UML – уніфікована мова моделювання (Unified Modeling Language) - це система позначень, яку можна застосовувати для об'єктно-орієнтованого аналізу і проектування. Її можна використовувати для візуалізації, специфікації, конструювання та документування програмних систем.

Важливо, що UML перекладається як Unified Modeling Language. Головне тут слово Unified. Тобто наші картинки зрозуміємо не тільки ми, а й інші люди, які знають UML. Виходить, це така міжнародна мова малювання схем.

Плюси і мінуси UML проектування.

Мінуси:

1. трата часу;
2. необхідність знання різних діаграм і їх нотацій.

Плюси:

1. можливість подивитися на завдання з різних точок зору;
2. іншим програмістам легше зрозуміти суть завдання і спосіб його реалізації;
3. діаграми є порівняно простими для читання після досить швидкого ознайомлення з їх синтаксисом.

У даному випадку було вирішено використати Use case діаграму. Use Case описує сценарій взаємодії учасників (як правило - користувача і системи). Учасників може бути 2 і більше. Користувачем може виступати як людина, так і інша система.

На Use case діаграмі зображуються:

1. актори - групи осіб або систем, що взаємодіють з нашою системою;
2. варіанти використання (прецеденти) - сервіси, які наша система надає акторам;
3. коментарі;
4. відносини між елементами діаграми.

Найбільш правильний порядок побудови діаграми наступний:

1. виділити групи дійових осіб (які працюють з системою по-різному, часто через різні права доступу);
2. ідентифікувати якомога більше варіантів використання (процесів, які можуть виконувати користувачі). При цьому не слід ділити процеси занадто дрібно, потрібно вибирати лише ті, які дадуть користувачеві значущий результат. Наприклад, касир може «продати товар» (це буде прецедентом), проте «введення штрих-коду товару для отримання ціни» самостійним прецедентом не є;
3. доповнити прецеденти словесним описом (сценарієм):
   - для кожного прецеденту створити розділи: «головна послідовність» і «альтернативні послідовності»;
   - при складанні сценарію потрібно наполегливо ставити замовнику питання «що відбувається?», «що далі?», «що ще може відбуватися?» і записувати відповіді на них.

В результаті досліджень було створено Use case діаграму для даного проекту.

Отже з рисунку 2.12 можна побачити як відбувається загальна робота додатку. User – це людина, яка опублікувати товар та його продати. Finder – користувач, який хоче купити авто.

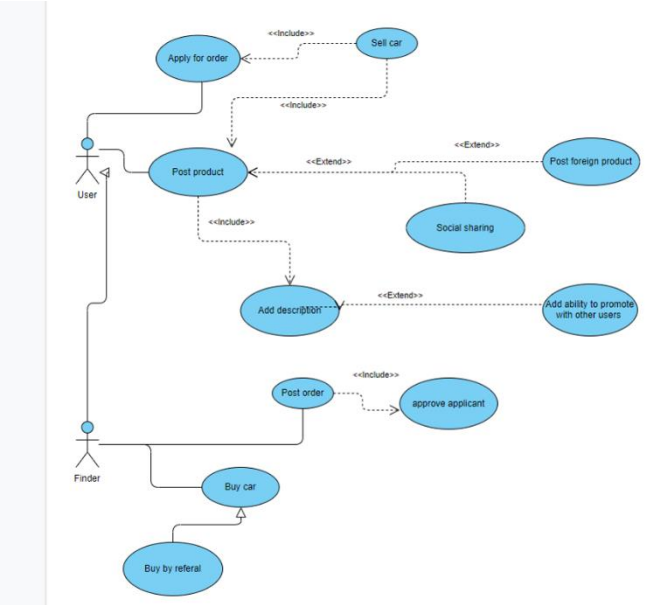



Рисунок 2.12 - Use case діаграма

Після розробки моделі проекту, дуже важливо розробити правильну архітектуру бази даних, оскільки це один з найважливіших етапів розробки будь-якого веб-додатку. Відношення сутностей у базі даних зображено на рисунку 2.13.

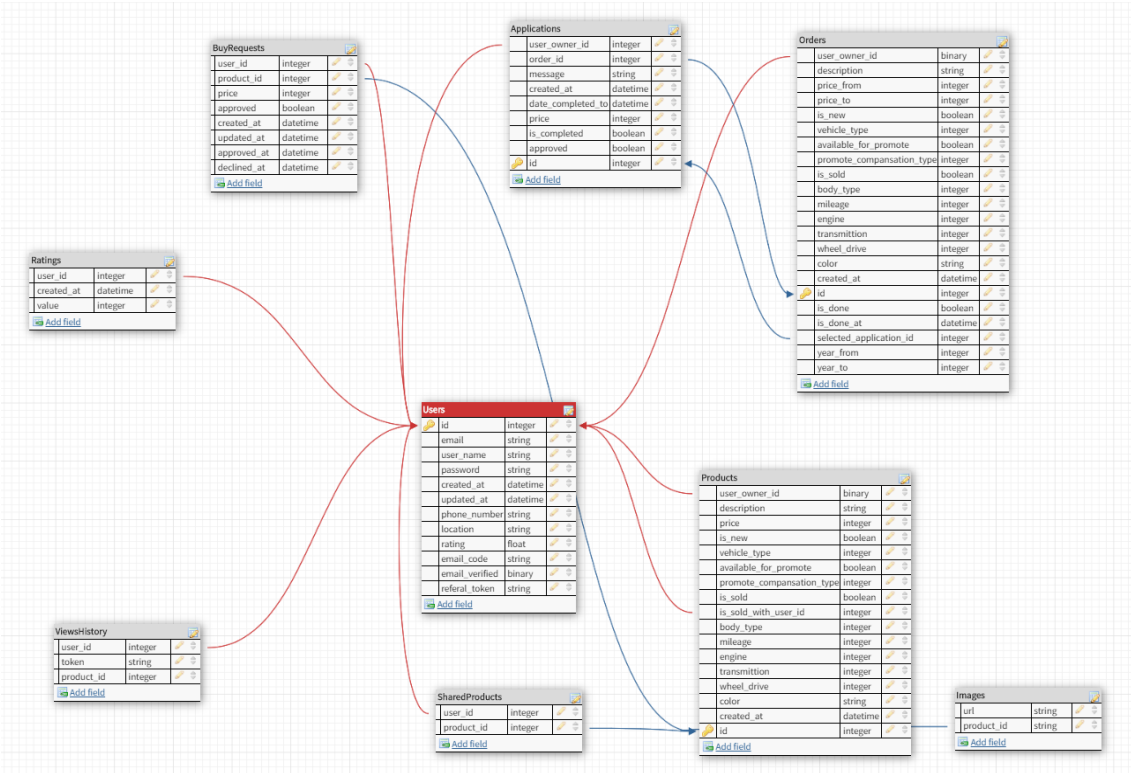



Рисунок 2.13 - Відношення сутностей
у базі даних

Отже база даних складається з таких сутностей:

- користувач(рис. 2.14);

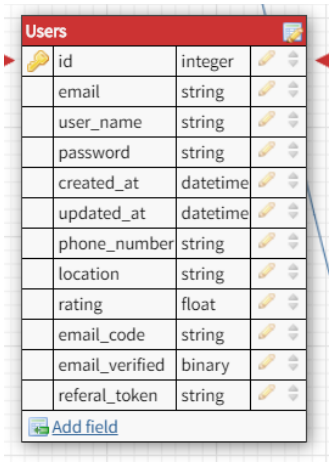



Рисунок 2.14 - сутність «Користувач»

Основними полями тут є інформація про самого користувача, емейл код, який приходить людині на емейл пошту і за допомогою якого людина може верифікувати свій профіль. Рейтинг – оцінка, якою його оцінити інші люди. Реферальний токен – унікальний код користувача, за допомогою якого система зможе визначити, що даний користувач допоміг комусь у продажі.

- список рейтингу користувача(рис. 2.15);

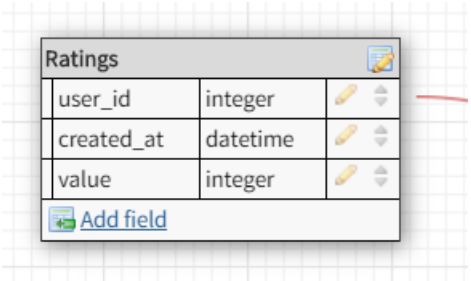



Рисунок 2.15 - сутність «Рейтинг користувача»

Дана модель збирає рейтинг користувача від інших людей

- історія переглядів користувача(рис. 2.16);

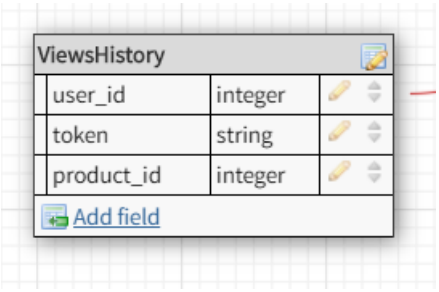



Рисунок 2.16 - сутність «Історія переглядів»

- товари, які належать даному користувачу(рис. 2.17);

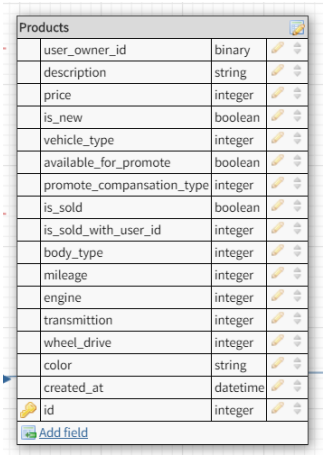



Рисунок 2.17 - сутність «Власні товари користувача»

- товари, які цей користувач поширює(рис. 2.18);

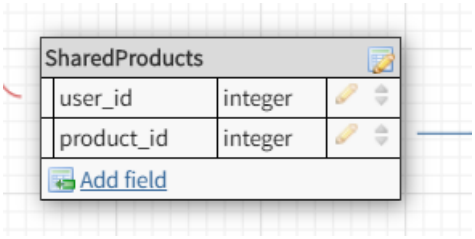



Рисунок 2.18 - сутність
«Товари, які поширив даний користувач»

- зображення, які належать до товару(рис. 2.19);

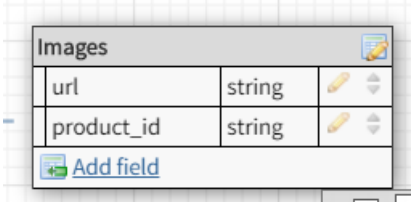



Рисунок 2.19 сутність

«Зображення товару»

- список запитів на купівлю даного товару(рис. 2.20);

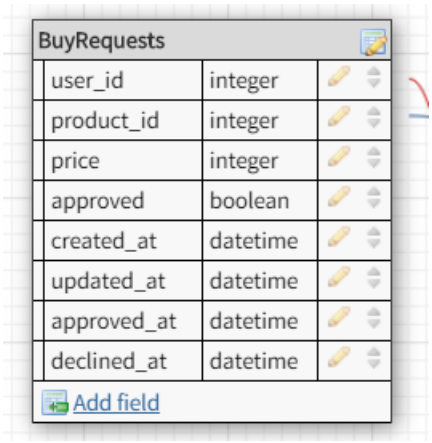



Рисунок 2.20 - сутність
«Список запитів на купівлю товару»

- список замовлень користувача(рис. 2.21);

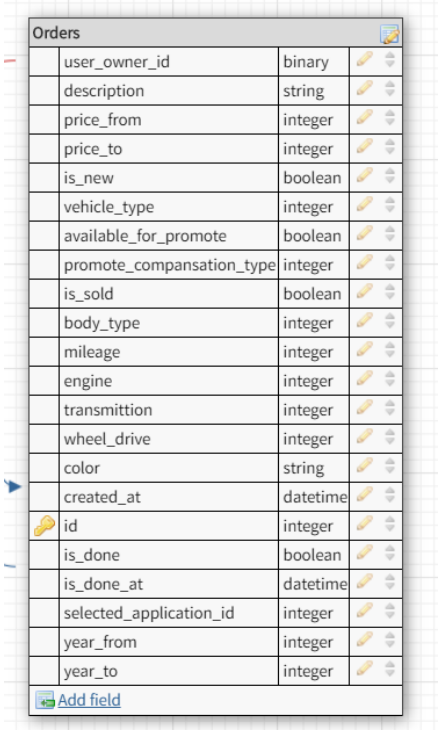



Рисунок 2.21 - сутність
«Замовлення користувача»

- список заявників до цього замовлення(рис. 2.22).

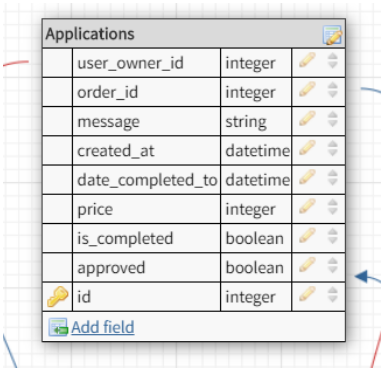



Рисунок 2.22 - сутність
«Заявник до замовлення»

Отже в розробці даної бази даних було використано такі типи зв’язків як «один до багатьох» та «багато до багатьох».

Приклади «один до багатьох» – це:

1. в одного користувача існує багато товарів;
2. в одного користувача існує багато «оцінок»;
3. в одного користувача існує багато товарів з історії переглядів;
4. в одного користувача існує багато замовлень;
5. в одного замовлення існує багато кандидатів на реалізацію;
6. в одного товару існує багато зображень;
7. в одного товару існує багато кандидатів на купівлю.

Приклад «багато до багатьох» – багато користувачів можуть додавати в свої списки однаковий товар.

## 2.6 Засоби розробки проекту

Для розробки даного проекту було вирішено використати технологію Node.js.

Node.js (або просто Node) - це серверна платформа для роботи з JavaScript через рушій V8[2, 3, 13]. JavaScript виконує дію на сторонніх клієнтах, а Node - на серверах. З допомогою Node можна писати повноцінні

масштабні додатки. Node вміє працювати з зовнішніми бібліотеками, виконувати команди JavaScript і виконувати роль веб-сервера.

З Node.js легше масштабувати проекти. При одночасному підключенні до серверів тисяч користувачів Node працює асинхронно, правильно розставляє пріоритети і використовує ресурси. Більшість інших мов для розробки бекенду при підключенні виділяють новий потік.

На рисунку 2.23 зображена схема роботи більшості мов програмування для бекенду. Тобто вони є багатопотоковими. Коли на сервер приходить запит від клієнта, то сервер виділяє для його обробки цілий потік. Він переходить у статус «зайнято» і починає робити якісь обчислення або звертатись до якихось інших API.

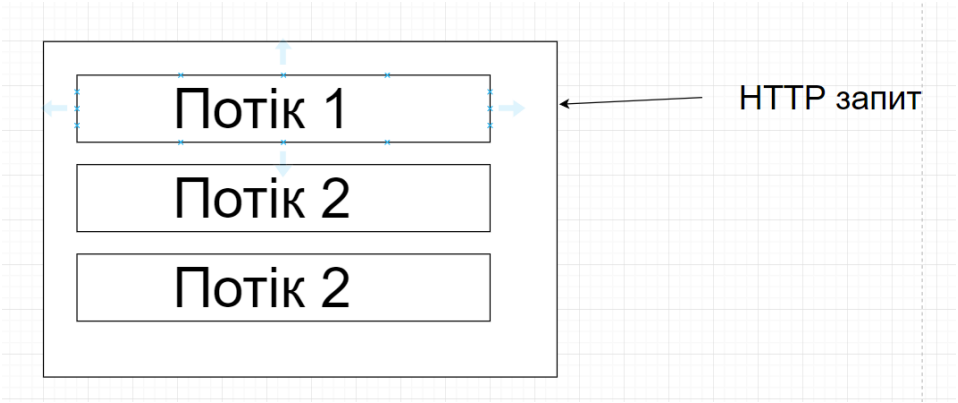



Рисунок 2.23 - Принцип роботи дії більшості мов програмування для бекенд розробки

Наприклад приходить запит, сервер його отримує і робить звернення до бази даних для отримання якоїсь інформації. Після цього сервер робить запит до якогось зовнішнього API і після отримання всіх результатів відправляє відповідь клієнту. Варто зауважити, що увесь цей час потік, який виконує цю роботу є зайнятим і відповідно ніхто до нього не зможе звернутись поки він працює.

Отже вагомим мінусом такої архітектури є те, що вона не дуже підходить для горизонтального розширення. Тобто на такому сервері завжди є якась обмежена кількість потоків, а отже, якщо всі потоки будуть зайняті і

якщо клієнт в цей час звернеться до сервера, то йому необхідно буде почекати до моменту поки не звільниться якийсь потік.

Отже Node.js – це середовище, яке використовує всього лише 1 потік для роботи. Оскільки Node – це асинхронна платформа виконання коду, то при новому з'єднанні йому не потрібно чекати поки новий запит опрацюються до кінця, щоб почати обробку наступного.

Node отримує новий запит і робить наприклад запит до бази даних. Одразу після цього платформа починає працювати з наступним запитом, а обробку першого продовжить зразу ж як отримає відповідь від БД.

На рисунку 2.24 зображений процес обробки запитів на Node.

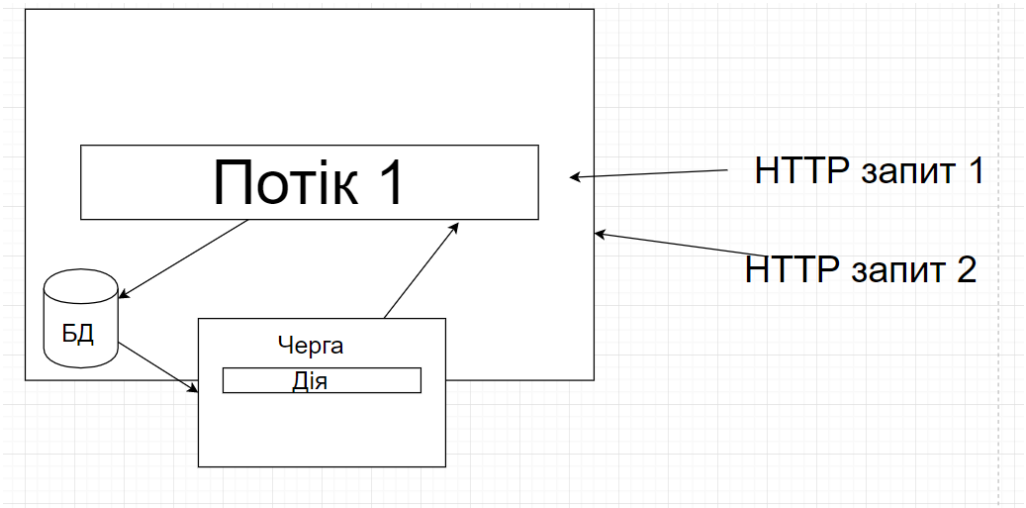



Рисунок 2.24 - Принцип роботи Node.js

На ньому відбувається наступний хід подій:

1. приходить запит 1;
2. Node його обробляє і доходить до моменту коли необхідно отримати інформацію з БД;
3. запит до БД;
4. обробка запиту 2;
5. запит до стороннього API 2;
6. прийшла відповідь від БД;
7. обробка відповіді від БД;

8. надсилання відповіді клієнту та закриття активного з’єднання;

9. обробка відповіді 2;

10. відповідь клієнту 2;

11. закінчення роботи програми.

Недоліком середовища Node є те, що платформа не розрахована для значних обчислень, оскільки обчислення є синхронними, що повністю заблокує потік роботи платформи і не дозволить обробити запити нових клієнтів. Тому, якщо на сервері потрібно проводити якісь значні обчислення, то краще використати якусь іншу мову програмування.

Для збереження даних було вирішено використати нереляційну базу даних MongoDB.

MongoDB - це високоефективна документоорієнтована база даних без конкретних схем даних, яка відноситься до нереляційних баз даних.

Структура: містить у собі велику кількість колекцій, вони містять у собі багато документів (об'єктів), які містять у собі очевидний вміст типу «ключ-значення». Документ має динамічну схему, що означає:

1. документи в одній колекції можуть мати різну кількість пар «ключ-значення»;

2. їх типи можуть відрізнятись.

# 3 РОЗРОБКА ПРОЕКТУ

## 3.1 Загальний огляд

Для пришвидшення розробки було вирішено використовувати фреймворк для Node.js, який називається Express.js[8]. Фреймворк дозволяє робити код структурованим, читабельним та лаконічним. А також містить багато готових рішень, які значно спрощують роботу програміста.

Основна ідея фреймворка полягає у функціях-посередниках (middleware)[11, 12]. Приклад функції зображено на рисунку 3.1. Це функції, які поступово опрацьовують запит користувача, і викликаються ланцюжком одна за одною.

```
app.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

Рисунок 3.1 - Використання middleware функції

Отже функції-посередники повинні отримати на вхід 3 аргументи:

1. request;
2. response;
3. next.

Request(req) – це об’єкт, який містить містить інформацію про звернення клієнта. З нього можна дізнатись метод звернення, параметри звернення, query-параметри і тд.

Response(res) – це об’єкт, який представляє із себе HTTP інформацію, яку сервер надішле клієнту.

Next(next) – це колбек, який необхідно викликати, коли ми хочемо перейти до наступної функції-посередника.

### 3.2 Розгортання проекту

Для того, щоб почати працювати з проектом, необхідно розгорнути середовище, за допомогою якого ми зможемо почати розробку. Спочатку необхідно ініціалізувати сам проект. Для цього необхідно в командний рядок ввести команду «npm init». Для того, щоб вона спрацювала необхідно встановити середовище Node.js на свій робочий комп'ютер. Після цього додаток npm також автоматично встановиться[23].

Дана команда створює базовий шаблон проект, а точніше файл package.json. В даному файлі міститься базова інформація про проект, така як назва, опис. Важливими пунктами є поля «scripts», «dependencies» та «devDependencies». Поле «scripts» дозволяє автоматизувати різноманітні маніпуляції з нашим проектом.

На рисунку 3.2 поле «scripts» містить декілька команд. Наприклад команда «serve» дозволяє спочатку запустити команду «babel-node» з аргументом «./src/www», а «nodemon –exec» викликається з аргументом, який є результатом виконання попередньої команди[9, 10].

```
{
  "name": "join-auto",
  "version": "0.0.0",
  "private": true,
  "main": "build/www.js",
  "scripts": {
    "rebuild": "npm run clean && babel src --out-dir build",
    "clean": "rm -rf build && mkdir build",
    "start": "npm run clean && npm run rebuild && node ./build/www.js",
    "serve": "nodemon --exec babel-node -- ./src/www"
  },
  "engines": {
    "node": "12.16.1"
  },
  "dependencies": {
    "aws-sdk": "^2.673.0",
    "babel-install": "^2.1.0",
```

Рисунок 3.2 - Структура файлу package.json

Dependencies – це модулі, які були використані в процесі розробки нашого додатку.

DevDependencies – те ж саме, що і dependencies, але модулі підключені в цьому полі не потрапляють в фінальний білд додатку.

Для швидкого створення структури проекту, використаємо плагін express-generator. Командою «npm install express-generator –g» глобально встановимо плагін для створення нашого скелетону. Після цього за допомогою командного рядку заходимо в директорію нашого проекту і виконуємо команду «express .». Дана команда згенерувала будову, яка зображена на рисунку 3.3.

```
create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/public
create : myapp/public/javascripts
create : myapp/public/images
create : myapp/routes
create : myapp/routes/index.js
create : myapp/routes/users.js
create : myapp/public/stylesheets
create : myapp/public/stylesheets/style.css
create : myapp/views
create : myapp/views/index.pug
create : myapp/views/layout.pug
create : myapp/views/error.pug
create : myapp/bin
create : myapp/bin/www
```

Рисунок 3.3 - Результат виконання
команди «express .»

Оскільки наш додаток працює за допомогою REST API, то немає необхідності зберігати в директорії папки з шаблонізаторами, стилями та іншими файлами, які використовуються для клієнтської частини. Видаляємо всі відповідні файли.

Двигуном нашого додатку є файл «bin/www.js»(рис. 3.4). Переносимо його в кореневу директорію для більшої зручності.

```
var app = require('./app');
var debug = require('debug')('join-auto:server');
var http = require('http');

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3001');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);

/**
 * Listen on provided port, on all network interfaces.
 */

server.listen(port);
```

Рисунок 3.4 - Структура файлу «www.js»

Отже, для того, щоб запустити наш сервер, імпортуємо модуль http, який вже є вбудованим в  Node структуру. Також імпортуємо app.js файл, та за допомогою команди http.createServer(app) запускаємо сервер та починаємо слухати події. Файл «app.js» – це корінь нашого фреймворку. В даному випадку, передаючи його аргументом в метод, ми переприсвоюємо основні налаштування, які виконується при вхідних та вихідних відповідях нашого сервера. Тобто зосереджуємо контроль додатком за допомогою express.

Для того, щоб нарешті активувати наш сервер, нам необхідно запустити даний файл за допомогою node. Заходимо в директорію з файлом та вводимо команду node www. Сервер запустився та вже може локально слухати події. Проте недоліком такого підходу є те, що сервер необхідно перезапускати кожний раз, коли ми вносимо в код. Оскільки це забирає багато часу при розробці, використаємо рішення, яке автоматично відстежує зміни у файлах та перезапускає сервер. Глобально встановлюємо плагін nodemon та можемо тепер запускати сервер за допомогою команди nodemon www.

В якості бази даних використовуємо Mongo Atlas, що є хмарним сховищем даних. Сервіс пропонує нам підключитись за допомогою рядка. Для того, щоб зберігати константи, створити файл з конфігурацією середовища. Назвемо його .env та всі статичні важливі дані будемо зберігати в ньому. Робота з MongoDB відбувається за допомогою фреймворку mongoose, тому імпортуємо його в наш файл app.js. Процес підключення бази даних зображений на рисунку 3.5.

```
Mongoose.connect(config.BD_BASE_URL, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => {
  console.log('connected success')
  })
  .catch(err => {
    console.log('connection error')
    console.log(err)
})

var app = express();
```

Рисунок 3.5 - Процес підключення бази даних

Для того, щоб клієнт міг звертатись до нашого сервера, потрібно налаштувати опції відповіді headers. Браузер клієнта не зразу робить прямий запит до сервера для взаємодії, а кидає запит «options», на який сервер відправляє відповідь. У відповіді міститься інформація про те, які типи запитів приймає сервер, які може приймати заголовки, а також з яких хостів може звертатись клієнт. Даний запит браузер робить автоматично перед звичайним зверненням. Отже необхідно дозволити клієнту звертатись зі всіх хостів, всіма методами, та всіма заголовками. Дана операція зображена на рисунку 3.6.

```
app.options('*', function (req, res) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header('Access-Control-Allow-Methods', '*');
  res.header("Access-Control-Allow-Headers", "*");
  res.end();
});

app.use('*', function (req, res, next) {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Methods', '*');
  res.header('Access-Control-Allow-Headers', '*');
  next();
});
```

Рисунок 3.6 - Налаштування заголовків відповіді

Варто зазначити, що в «продакшн-стані» такі дії є небезпечними, оскільки не варто дозволяти всім клієнтам звертатись до сервера з будь-якими опціями. Для даних налаштувань були використані функції-посередники, які додають зміни та за допомогою колбеку next(), продовжуть роботу обробки запиту.

### 3.3 Підключення маршрутизації та обробка помилок

Створимо новий файл «index.js» в директорії «routes». Для того, щоб у нас була можливість легко змінити версію API, додамо «обгортку» до звичайного роутера. Тепер у нас всі шляхи, які містяться у файлі «v1Routes» будуть починатися з «/v1/…»[16]. Отже, якщо ми захочемо додати нову версію API, то не потрібно витрачати багато часу на переписування коду, а лише продублювати один рядок і підключити до нього нові маршрути. Результат роботи зображений на рисунку 3.7.

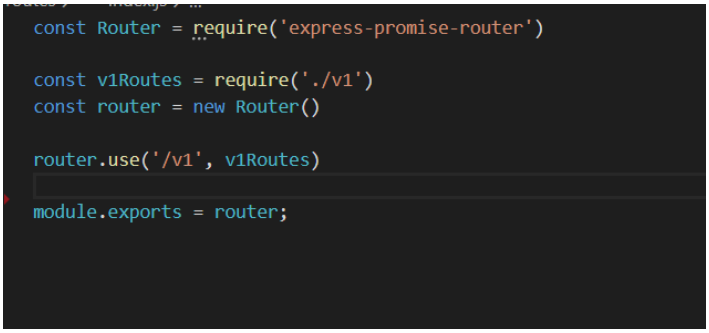



Рисунок 3.7 - Структура файлу routes

Також варто звернути увагу на те, як працює в Node експорт нашого коду назовні із файлу. Для експорту використовуємо «module.exports». Насправді, це дуже легка і зрозуміла поведінка. При компіляції нашого коду Node обгортає кожний файл в самовикликаючу функцію і передає їй аргументи, одним з яких є параметр «module», саме тому ми маємо доступ до цього об’єкту.

На рисунку 3.8 бачимо декілька основних модулів, які ми підключаємо до маршрутизації. До маршрутів «order» і «user» ми додаємо префікси, тому всі вони будуть починатись із цих префіксів.

```
var express = require('express');
var router = require('express-promise-router')();

const authRouter = require('./auth_routes.js')
const vehicleRouter = require('./vehicle_routes.js')
const vehicleOrderRouter = require('./vehicle_order_routes.js')
const userRouter = require('./user_routes.js')

router.use(authRouter)
router.use(vehicleRouter)
router.use('/order', vehicleOrderRouter)
router.use('/user', userRouter)

module.exports = router;
```

Рисунок 3.8 - Структура файлу v1Routes

Кожний файл, підключений до маршрутизації містить інформацію про всі свої внутрішні маршрути та REST-методи, валідації на коректність отриманих даних для кожного маршруту, а також функції-посередники. Структура налаштувань зображена на рисунку 3.9.

```
1  const Router = require('express-promise-router')
2  const authRouter = new Router()
3
4  const { login,
5    register,
6    forgotPassword,
7    resetPassword
8  } = require('@controllers/auth.controller.js')
9
10 const registerValidator = require('@/validators/register.js')
11 const loginValidator = require('@/validators/login.js')
12 const forgotPasswordValidator = require('@/validators/ForgotPassword.js')
13 const ResetPasswordValidator = require('@/validators/ResetPassword.js')
14
15 authRouter.post('/login', loginValidator, login)
16 authRouter.post('/sign-up', registerValidator, register)
17 authRouter.post('/forgot-password', forgotPasswordValidator, forgotPassword)
18 authRouter.post('/reset-password', ResetPasswordValidator, resetPassword)
19
20 module.exports = authRouter
```

Рисунок 3.9 - Приклад простої структури опису роутів

В Node.js існує 2 типи помилок, які сервер повинен повернути користувачу. Перший типу – це помилка «404», яка повідомляє, що даний шлях не знайдено. Друга помилка «500», якою ми повинні повідомити, що стався збій на сервері.

Перший тип помилок відловити дуже просто. Оскільки express перевіряє схожість з запрошеним маршрутом лінійно в порядку, якому ми підключали маршрути, то якщо маршрут знайденим не буде, алгоритм просунеться до кінця списку. Тому достатньо після підключення всіх маршрутів додати ще один, який буде сповіщати про помилку «404»(рис. 3.10).

```
app.use('*', function (req, res, next) {
  return res.status(404).json({
    error: 'Not found'
  })
});
```

Рисунок 3.10 - Обробка 404 помилки

Викликаємо метод «status», який сповіщає клієнта про помилку, а також додатково добавляємо повідомлення, за допомогою методу «json».

Для обробки внутрішніх помилок сервера немає якогось конкретного алгоритму дій. Можна кожний обробник запиту поміщати в try… catch… блок і, відловлюючи помилку, сповіщати про це клієнта.

В даному випадку було використану функцію-посередник для відловлювання такого типу помилок. Дана функція першим аргументом приймає не об’єкт запиту, а саме об’єкт помилки. І обов’язково розташовується в самому кінці коду. Відловивши помилки, приводимо її до якогось конкретного вигляду і відправляємо її клієнту(рис. 3.11).

```
app.use(function (err, req, res, next) {
  if (res.headersSent) {
    return next(err);
  }

  if (!err.yupError && !(err.value && err.value.yupError)) {
    return res.status(err.status || 500).json({
      stack: err.stack,
      message: err.message
    });
  }

  if (!err.inner.length) {
    return res.status(422).json([
      {
        field: err.path,
        error: err.errors[0]
      }
    ]);
  }
```

Рисунок 3.11 - Обробка внутрішніх помилок

### 3.4 Реєстрація користувача

Для початку створимо модель нашого користувача. Для цього створюємо файл «User.js» в папці «models». Об’єкт опису нашої моделі створюється за допомогою класу MongooseSchema, який на вхід приймає об’єкт з описом полів нашого користувача.

Варто зрозуміти, що оскільки ми працюємо з нереляційною базою даних, то немає потреби для сутностей опису якихось складний характеристик користувача створювати нову таблицю. Оскільки ми працюємо з JSON форматом, то достатньо добавити нове поле будь-якого допустимого типу(навіть складного) та в ньому описувати якісь додаткові поля.

Так наприклад, для користувача можна створити такі внутрішні складні поля як «масив рейтингу від користувачів», описати список продуктів, які він просуває та додати історію його переглядів.

На рисунку 3.12 бачимо опис схеми моделі користувача. Отже кожне поле містить свій тип, а для опису складних полів потрібно добавити знак «[]» або «{}» в залежності від його типу.

```
const UserSchema = new MongooseSchema({
  email: String,
  user_name: String,
  password: String,
  created_at: Date,
  updated_at: Date,
  phone_number: String,
  location: String,
  rate_array: [
    {
      from_user: { type: Mongoose.Types.ObjectId, ref: 'Vehicle' },
      created_at: Date,
      value: Number
    }
  ],
  rating: Number,
  is_admin: Boolean,
  email_code: String,
  email_verified: false,
  shared_products: [{ type: Mongoose.Types.ObjectId, ref: 'Vehicle' }],
  referal_token: String,
  views_history: [
    {
      product_id: { type: Mongoose.Types.ObjectId, ref: 'Vehicle' },
      token: String,
      created_at: Date
    }
  ]
})
```

Рисунок 3.12 - Опис полів користувача

Варто звернути увагу на поле «ref». Об’єкт з даним полем відіграє роль вторинного ключа, який посилається на іншу модель. Також містить у собі поле зі значенням «ObjectId», яке позначає, що тип даного рядка – Id, якоїсь іншого документу.

Для того, щоб дати зрозуміти базі даних, що даний каркас являється моделлю mongoDB, варто один раз викликати метод «Mongoose.model», який приймає на вхід назву нашої моделі та саму схему моделі. Дану операцію зображено на рисунку 3.13. Тепер база даних знає, то існує така модель як «Користувач».

```
module.exports = Mongoose.model('User', UserSchema)
```

Рисунок 3.13 - Підключення схеми до моделі

Для роботи з реєстрацією користувача створимо новий контроллер, який назвемо «auth.controller.js». Створимо метод register, який буде створювати нового користувача при виклику. В даному методі дістаємо необхідні поля із запиту клієнта та створюємо новий документ у базі.

Оскільки головна ідея RESTful API полягає у тому, що кожний запит повинен нести всю інформацію про користувача, то реалізувати спосіб, який це зможе відтворити. У даному випадку використаємо JWT токен для автентифікації клієнта(рис. 3.14).

```
[ Header ].[ Payload ].[ Signature ]
```

Рисунок 3.14 - Модель JWT

JWT складається з 3-х основних частин:

1. заголок;
2. опис;
3. підпис.

Заголовок токена відповідає за тип токена та алгоритм шифрування. Опис містить основну інформацію, яку ми у нього шифруємо. Підпис містить зашифрований ключ, який є необхідним для розшифрування опису. Весь токен є строкою, яка розділяється знаком крапки.

Для використання токенів був використаний спеціальний плагін з бібліотеки npm. Першим параметром метод-генератор приймає тіло, яке ми хочемо зашифрувати, а другим секретний ключ, за допомогою якого ми виконуємо шифрування.

На рисунку 3.15 зображена генерація унікального токена для користувача, де «this» означає поточний документ з яким ми зараз працюємо. Поле «_id» – це унікальний ідентифікатор кожного документа в MongoDB, який база генерує сама.

```
UserSchema.methods.generateToken = function () {
  return jwt.sign({id: this._id}, config.JWT_KEY)
}
```

Рисунок 3.15 - Генерація токена

Отже, із рисунку 3.16 зрозуміло, що ми отримуємо дані і за допомогою методу «User.create» створюємо новий документ в базі, після цього генеруємо користувачу унікальний токен і відправляємо йому статус «201», який означає «створено». Окрім статусу також відправляємо основні дані користувача.

```
const register = async (req, res) => {
  const { email, user_name, password, phone_number, location} = req.body

  const user = await User.create({
      email,
      user_name,
      password,
      phone_number,
      location,
      is_admin: false,
      shared_products: [],
  })

  const token = user.generateToken()

  const userDetails = getUserInfo(user)

  return res.status(201).json({
    ...userDetails,
    auth_token: token
  })
}
```

Рисунок 3.16 - Обробка запиту на створення користувача

Варто звернути увагу на те, що на даному зображенні виглядає, ніби ми зберігаємо пароль користувача прямо в базу. Насправді, це не так, оскільки це є дуже небезпечно і вважається непрофесійним маневром.

Для того, щоб зберегти в базу даних зашифрований пароль, використаємо тригер «pre('save')», який самовиконається перед тим зберіганням інформації користувача. На рисунку 3.17 можна побачити, що ми переписуємо наш пароль зашифрованим за допомогою методу «bcrypt.hashSync».

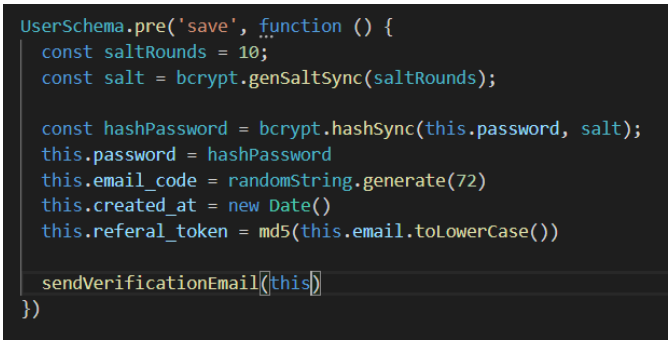



Рисунок 3.17 - Опис маніпуляцій перед зберіганням даних в БД

Окрім цього в даному тригері ми генеруємо код, який необхідний для підтвердження емейлу, поля «created_at» та «referal_token». Останнє поле нам необхідно для реалізацію функціоналу просування чужих товарів. І на кінець викликаємо метод, який реалізує надсилання емейлу користувачу.

Варто звернути увагу на те, що в реалізації методу обробку створення нового користувача, ми не проводимо валідацію даних, які нам присилає клієнт. Зрозуміло, що така поведінка не є коректною. Для валідації даних викликається функція-посередник, яка виконує свою роботу перед тим, як дані доходять до даної частини виконання.

Тобто порядок виконання наступний: сервер приймає запит клієнта, перевіряє чи існує пара, де співпадають запрошені маршрут та метод звертання(у нашому випадку співпадає). Після цього в налаштуваннях даного роута викликаються по черзі всі вказані функції-посередники. На рисунку 3.18

бачимо, що спочатку викликається метода, який проводить валідацію даних і лише після цього відбувається сама обробка.

```
authRouter.post('/sign-up', registerValidator, register)
```

Рисунок 3.18 - Порядок виконання проміжних Функцій

Також потрібно зауважити, що всі імпорти починаються із знаку «@», це не особливість Node.js, а спеціальна додаткова обробка усіх файлів перед тим як запустити сам сервер(рис. 3.19). Даний функціонал був доданий для швидкого і зручного імпорту файлів, щоб не включати файли за допомогою відносних шляхів, оскільки такі імпорти є тяжкими та заплутаними для читання.

```
const registerValidator = require('@/validators/register.js')
const loginValidator = require('@/validators/login.js')
const forgotPasswordValidator = require('@/validators/ForgotPassword.js')
const ResetPasswordValidator = require('@/validators/ResetPassword.js')
```

Рисунок 3.19 - Імпорт валідаторів

Роботою опрацювання файлів займається плагін Babel, який спочатку пробігається через весь проект графоподібним чином і парсить повністю усі файли. Лише після закінчення його роботи відбувається запуск сервера.

Для того, щоб провалідувати об’єкт запиту користувача, було використану спеціалізовану бібліотеку «yup». Після її підключення описуємо «форму» самого об’єкта та вказуємо правила для кожного необхідного поля. Приклад схеми валідатора зображено на рисунку 3.20.

```
const RegisterSchema = yup.object().shape({
  user_name: yup.string().required(),
  email: yup.string().email().required(),
  password: yup.string().min(6).required(),
  phone_number: yup.string().required(),
  location: yup.string().required(),
})
```

Рисунок 3.20 - Опис схеми валідатора

Щоб ініціалізувати саму валідацію викликаємо метод «validate» у об’єкту нашої схеми(рис. 3.21). Поле «abortEarly» додається в параметри методу як додаткова опція і означає, що ми повинні провести повне валідування для того, щоб отримати всю інформацію про помилки.

```
module.exports = async (req, res, next) => {
  const { user_name, email, password, phone_number, location } = req.body

  try {
    await RegisterSchema.validate({
      user_name,
      email,
      password,
      phone_number,
      location
    }, {
        abortEarly: false
    })

    const existingUser = await UserModel.findOne({ email })

    if (existingUser) {
      throw new yup.ValidationError(
        'This user already exist',
        { ...req.body, yupError: true },
        'email'
      )
    }

    next()
  } catch (err) {
    next({...err, yupError: true})
  }
}
```

Рисунок 3.21 - Валідація запиту

Якщо усі поля вірні, провіряємо чи існує уже користувач з такою поштою, то існує то також ініціалізуємо помилку з відповідним повідомленням. Якщо була знайдена якась помилка, то блоку «catch» перехоплює її та передає в глобальний обробник помилок, який був підключений в корені проекту. У випадку, коли помилок немає, то передаємо роботу наступному обробнику нашого запиту.

### 3.5 Авторизація користувача

Авторизація користувача відбувається в тому ж самому контролері, що і реєстрація. Щоб користувач зміг провести авторизацію, необхідно отримати 2 поля: емейл користувача і його пароль. Після цього проводимо валідацію

відповідних даних. Коли валідатор закінчив свою роботу, переходимо до обробки запиту.

З рисунку 3.22 можна побачити, що спочатку ми провіряємо чи існує користувач з введеним емейлом. Якщо такого користувача немає, то відповідно відправляємо на клієнт помилку, що емейл невірний. Якщо ж такий емейл є в базі, то звіряємо тоді пароль, який ми прийняли від клієнта з наявним у базі. Оскільки в базі записаний зашифрований пароль, то зрівнюємо за допомогою методу «bcrypt.compare». Якщо паролі не співпадають, то вертаємо ту саму помилку, по тому ж полю.

```
const login = async (req, res) => {
  const { email, password } = req.body

  const existingUser = await User.findOne({ email })

  if (!existingUser) {
    throw new yup.ValidationError(
      'Check your credentials.',
      { ...req.body, yupError: true },
      'email'
    )
  }
  const passwordMatch = await bcrypt.compare(password, existingUser._doc.password);

  if (!passwordMatch) {
    throw new yup.ValidationError(
      'Check your credentials.',
      { ...req.body, yupError: true},
      'email'
    )
  }
```

Рисунок 3.22 - Обробка авторизації

Якщо всі перевірки були пройдені, то генеруємо новий токен користувача і повертаємо йому необхідну інформацію. Зі свого боку користувач повинен зберегти цей токен[21], додавати в заголовки до кожного запиту, який потребує авторизації. Дана операція зображена на рисунку 3.23.

```
const token = existingUser.generateToken()
const userDetails = getUserInfo(existingUser)


return res.status(200).json({
  ...userDetails,
  auth_token: token
})
```

Рисунок 3.23 - Генерація відповіді клієнту

### 3.6 Відновлення паролю

Для відновлення паролю користувача було використано пару із двох API. Виклик першого дає серверу знати про те, що користувач хоче відновити пароль. Для цього сервер генерує унікальний токен і відправляє його користувачу через пошту, що зображено на рисунку 3.24. Користувач повинен перейти з пошти по силці із спеціальним токеном та виконати запит, який встановлює новий пароль.

```
const forgotPassword = async (req, res) => {
  const { email } = req.body

  const user = await User.findOne({ email })

  if (!user) {
    return res.status(400).json({
      email: 'User doesn\'t exist'
    })
  }

  const userDetails = user._doc

  const resetPasswordInstance = await ResetPassword.create({
    email: userDetails.email,
    created_at: new Date(),
    token: user.generateForgotPasswordToken()
  })

  sendResetEmail(resetPasswordInstance)

  return res.status(200).json(resetPasswordInstance._doc)
}
```

Рисунок 3.24 - відновлення паролю

Отже, створюється новий документ, який містить в собі унікальний токен, дату створення та емейл користувача. Всі дані є обов’язковими. Також спочатку виконується перевірка чи користувач з таким емейлом існує у базі. Валідація запиту зображена на рисунку 3.25.

```
const resetPassword = async (req, res) => {
  const { token, password } = req.body

  const resetPasswordInstance = await ResetPassword.findOne({ token })

  if (!resetPasswordInstance) {
    return res.status(422).json({
      password: 'token is invalid'
    })
  }

  const { email, created_at } = resetPasswordInstance._doc
  const now = new Date().getTime()
  const createdAtPlusDay = created_at.getTime() + (1 * 24 * 60 * 60 * 1000)

  if (now > createdAtPlusDay) {
    await resetPasswordInstance.deleteOne()

    return res.status(422).json({
      password: 'token is expired'
    })
  }
```

Рисунок 3.25 - Валідація запиту
для перезапису

Після того, як користувач відкрив сторінку з вводом нового пароль, він повинен передати сам пароль та токен, який він отримав із емейлу. Якщо всі дані введені вірно, то шукаємо документ з деталями відновлення. Якщо такого документу знайдено не було, то відповідно токен є невірним. Відправляємо відповіне повідомлення клієнту.

В іншому випадку дістаємо дату створення цього документу. Якщо документ був створений давніше ніж 1 день тому, то значить він вже не є валідним. Сповіщаємо про це користувача.

Після всіх перевірок створюємо новий пароль та зразу його зашифровуємо, що зображено на рисунку 3.26.

```
const saltRounds = 10;
const salt = bcrypt.genSaltSync(saltRounds);

const user = await User.findOneAndUpdate({
  email
}, {
    password: bcrypt.hashSync(password, salt)
})

await resetPasswordInstance.deleteOne()

return res.status(200).json(user._doc)
```

Рисунок 3.26 - Створення нового паролю

Після цього шукаємо в базі користувача, якому належить емейл з якого він перейшов та обновляємо його пароль. Видаляємо документ, який відповідає за перезапис. Отже якщо користувач ще раз перейде з емейлу за посиланням, то вже не зможе відновити пароль, оскільки документ з відновлення паролю вже видалений з бази. Після цього користувач знову може авторизуватись за допомогою нового паролю.

### 3.7 Створення оголошення про продаж

Створимо нову модель, яка буде відповідати за опис автомобіля. Також додамо новий файл з налаштуваннями маршрутизації та новий контролер. Модель автомобіля також в собі містить список запитів на його купівлю(рис. 3.27) та зображення, які до нього прив'язані.

```
images: [String],
buy_requests: [
  {
    user_id: { type: Mongoose.Types.ObjectId, ref: 'User' },
    comment: String,
    price: Number,
    approved: Boolean,
    created_at: Date,
    updated_at: Date,
    approved_at: Date,
    declined_at: Date
  }
]
```

Рисунок 3.27 - Короткий опис схеми автомобіля

Отже зображення – це масив рядків, а точніше посилань по яких ми зможемо дістатись до них. Список запитів – масив, який окрім відповідної особи також містить якусь додаткову інформацію.

Створення нового оголошення відбувається дуже просто: отримуємо дані від клієнта, проводимо їх валідацію та в разі успіху створюємо новий документ у відповідній колекції.

Для зберігання картинок існує 3 основні методи:

1. зберігати їх на власному сервері;

2. зберігати їх у базі;
3. зберігати у хмарі.

Перший варіант не підходить, оскільки він обмежує горизонтальний ріст нашого додатку. Тобто, завжди буде необхідно приєднувати нові машини для зберігання даних.

Другий варіант вирішує цю проблему, але створює нову. Зберігання файлів у MongoDB – дуже затратне, оскільки навіть 1 зображення займає дуже багато місця, що є дуже неефективним та дорогим способом.

В даному проекті використано 3 варіант. Його плюсами є дешевизна та легкість інтеграції. Було вибрано AWS S3 хмару. Інтерфейс для редагування «корзини» зображений на рисунку 3.28. Після реєстрації в сервісі, потрібно створити нову політику конфіденційності та нового користувача, прив'язати до нього дану політику, а також наділити правами за допомогою яких він зможе програмно завантажувати файли в корзину.

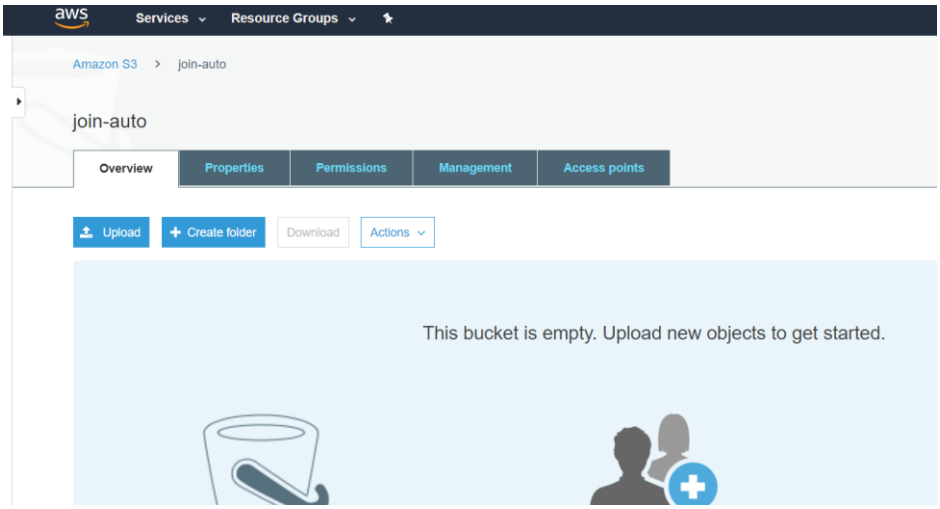



Рисунок 3.28 - Інтерфейс AWS S3

Для інтеграції сервісу в проект була використана бібліотека «aws-sdk»[15]. Заповнюємо базову конфігурацію(access_key_id, secret_id, region), яку ми отримали при створенні користувача та можемо використовувати всі можливості сервісу aws.

Запит із файлом прийшов на сервер з типом form data. Для того, щоб його зчитати, використаємо бібліотеку multer. Multer – це функція, яка приймає об'єкт з описом нашого файлу. Туди входить фільтрування файлів, які задовільняють наші критерії, максимальний розмір файлу, а також місце його зберігання.

Фільтр файлів – це функція, яка на вхід приймає інформацію про файл та колбек функцію, яку ми повинні викликати з параметром true або false, що відповідно означає файл підходить під критерій, або ні.

Також у разі, якщо файл не підходить під критерій, то повертаємо помилку(рис. 3.29). Дана помилка зупинить зберігання файлів у хмару.

```
const fileFilter = (req, file, cb) => {
  if (file.mimetype === 'image/jpeg'
    || file.mimetype === 'image/png'
    || file.mimetype === 'image/jpg') {
    cb(null, true)
  } else {
    cb(null, false)
    return cb(new Error('Only .png, .jpg and .jpeg format allowed!'));
  }
}
```

Рисунок 3.29 - Фільтрування файлів

Після фільтрації файлів потрібно налаштувати опції їх зберігання. Вказуються такі параметри, як назва корзини для зберігання, налаштування кешу, додаткова інформація про файл, тип цього файлу, права для цього файлу, а також, що найголовніше – його назва. В даному випадку, щоб не допустити повторювання назв файлів, генеруємо його назву із пари «теперішня дата + оригінальна назва файлу»(рис 3.30).

```
storage: multerS3({
  s3: aws.s3,
  bucket: aws.bucket,
  cacheControl: 'max-age=31536000',
  metadata: function (req, file, cb) {
    cb(null, { fieldName: file.fieldname });
  },
  contentType: multerS3.AUTO_CONTENT_TYPE,
  acl: 'public-read',
  key: function (req, file, cb) {
    cb(null, Date.now().toString() + file.originalname)
  },
})
```

Рисунок 3.30 - Налаштування файлу при зберіганні

Після того, як файл завантажився в хмарище, отримуємо його пряме посилання та зберігаємо у базу[22].

Оскільки створювати оголошення можуть лише зареєстровані користувачі, нам треба обмежувати права доступу на дану дію. Для цього була використана функція-посередник, яка перед валідацією даних провіряє чи даний користувач є авторизованим.

Для цього дістаємо поле «Authorization» із заголовків запиту від клієнта. По даному полю клієнт повинен відправити свій токен, який він отримав при авторизації. В токені зашифрований унікальний ідентифікатор користувача. Якщо ідентифікатор не найдений, повертаємо клієнту помилку «401 Не авторизовано». В іншому випадку завершуємо нашу валідацію та переходимо до наступних кроків(рис. 3.31).

Після створення оголошення, будь-який користувач зможе отримати про нього інформацію по шляху «/products/:id» за допомогою методу GET.

```
module.exports = (req, res, next) => {
  const authToken = req.header(AUTH_HEADER_NAME)
  const userId = checkToken(authToken)

  if (!userId) {
    return res.status(401).json({})
  }

  req.token = authToken
  req.body.current_user_id = userId.id
  next()
}
```

Рисунок 3.31 - Перевірка авторизації

### 3.8 Отримання списку товарів

Для отримання списку автомобілів необхідні також базові поля для їх фільтрації, а також значення поточної сторінки та кількість товарів, яку ми можемо показати на одній сторінці.

Для сортування використаємо такі поля, як «ціна», «пробіг», «дата створення». Також зчитаємо порядок, можливі значення: 1 – від меншого до

більшого, -1 – від більшого до меншого. Для фільтрування використаємо практично всі поля, які доступні при створенні оголошення. Дістанемо всю інформацію із запиту, відфільтруємо дані, залишивши лише ті, які нам необхідні для запиту до БД(рис. 3.32).

```
const findModel = filterNulls({
  price: { $lte: newPriceTo || price_to, $gte: price_from },
  is_new,
  vehicle_type,
  available_for_promote,
  is_sold,
  body_type,
  mileage,
  engine,
  transmittion,
  wheel_drive,
  color
});
```

Рисунок 3.32 - Модель запиту

Спочатку нам необхідно отримати загальну кількість всіх доступних товарів. Це можна зробити за допомогою методу «Model.countDocuments».

Сортуємо дані, задаємо порядок їх відображення, переходимо на сторінку, яка була вказана у запиті, беремо ту кількість документів, яка теж була у запиті. Для легкої роботи із пагінацією був написаний клас Pagination та його обгортка PaginationWrapper, яка нам дозволяє легко задавати дані «ланцюжком»[5, 18, 19, 20].

Відправляємо клієнту дані про товари, а також дані про пагінацію. Для відображення пагінації також був написаний клас, який приймає об’єкт опцій та його форматує для легкого читання(рис. 3.33).

```
const pagination = new PaginationWrapper()
  .setPage(page)
  .setLimit(limit)
  .setCount(count)
  .build()
```

Рисунок 3.33 - Опис пагінації

### 3.9 Купівля автомобіля

Для купівлі авто також необхідно бути авторизованим, тому спочатку провіряємо чи клієнт відповідає цьому критерію. Користувач, який хоче купити авто, методом POST відправляє ідентифікатор авто, ціну, яку він готовий заплатити, а також додатковий коментар до запиту.

Спочатку необхідно перевірити чи є даний автомобіль у базі. Після цього перевіряємо чи користувач, якому належить дане оголошення не зробив виклик на його купівлю. Якщо автомобіль вже проданий, то кидати запит на купівлю теж вже немає сенсу.

Якщо даний користувач вже кидав такий запит саме на дане оголошення, то просто обновлюємо наявні дані без створення нового документу. В іншому випадку створюємо новий документ та зберігаємо.

Зі сторони продавця можна або прийняти запит, або його відхилити. В будь-якому разі, спочатку необхідно провірити чи користувач, який проводить маніпуляції з оголошенням має на це право. Якщо все добре і авто продане, то зберігаємо інформацію. Також вказуємо коли воно було продане та кому.

### 3.10 Просування автомобіля

Є можливість додати дане авто в список своїх товарів. Для цього необхідно спочатку перевірити чи користувач не є власником авто. Потім, чи дане авто вже не додане в список цього користувача. Якщо всі валідації були пройдені, то додаємо авто в поле «shared_products».

Тепер при перегляді товарів на профілі цього користувача буде незрозуміло кому вони належать. Тобто людина, яка зайде на профіль, подумає, що всі продукти належать даному користувачу.

Отже, при перегляді продуктів потрібно дістати за з'єднати 2 поля: «просунуті товари» та приватні авто, а також інтегрувати сюди пагінацію. Для виконання даного завдання був використаний aggregation framework від MongoDB. Даний фреймворк – це метод, який на вхід приймає масив конкретних операцій, які необхідно провести з базою даних.

В даному випадку були проведені такі операції:

1. знайти профіль з ID користувача;
2. взяти всі товари, які він просуває та зберегти як «shared_products»;
3. взяти з іншої колекції всі товари, які належать даному користувачу;
4. зберегти дані товари як «own_products»;
5. 2 типи товарів з'єднати та зберегти як «items».

Після даних маніпуляцій отримуємо загальну кількість всіх товарів та зберігаємо. Вказуємо налаштування сортування як «від нового до старого», встановлюємо значення пагінації, яке було отримано від клієнта. Відправляємо список товарів користувачу.

На рисунку 3.34 зображено декілька основний команд для взаємодії із БД. Команда «match» шукає в базі документ, який підходить до заданої умови. В даному випадку ми шукаємо документ з конкретним ідентифікатором. «Group» змінює вигляд документу, «unwind» говорить про те, що кожний елемент вкладеного масиву варто розглядати, як окремий документ. Метод «lookup» дозволяє отримати документи із іншої колекції по умові.

```
const concatItemsRequestSchema = [
    {
      $match: { _id: Mongoose.Types.ObjectId(id) },
    },
    {
      $group: { _id: '$shared_products' }
    },
    {
      $unwind: '$_id'
    },
    {
      $lookup: { from: 'vehicles', localField: '_id', foreignField: '_id', as: 'product' }
    },
```

Рисунок 3.34 - Приклад маніпуляцій із БД

Ми описуємо такі поля, як назву колекції, ключ даного документу по якому ми шукаємо, ключ із іншої колекції, а також назву поля, в яке ми зможемо занести результат.

Тепер при перегляді автомобіля, він буде записуватись в історію переглядів користувача. В опції буде переданий параметр, який вказує, що дана людина здійснила перегляд товару через іншого користувача. При купівлі авто, система зможе вказати, що воно було продане за допомогою користувача з профілю якого перейшов даний покупець. Для цього перед отриманням інформації про автомобіль викликається функція-посередник «addToViewsHistory», яка перевіряє чи в рядку запиту існує параметр «q», що є унікальний для кожного користувача. Якщо він є, то значить перегляд товару здійснений за допомогою цієї людини. Провіряємо чи в історії вже існує запис про даний продукт. Якщо існує то завершуємо виконання функції. В іншому випадку додаємо товар в історію та вказуємо параметр «q».

### 3.11 Рейтинг продавця

В додатку існує можливість виставити рейтинг профілю. Зробити це можуть лише користувачі, які взаємодіяли якось із цим профілем. Рейтинг передається методом POST та приймає поле оцінки (1-5) та ідентифікатор користувача, якого ми оцінюєм. Перевіряємо чи такий користувач існує, якщо ні – повертаємо помилку «404». Якщо людина хоче оцінити сама себе, також повертаємо помилку.

Оцінити профіль можуть 2 типи користувачів:

1. користувачі, за допомогою яких було продано товари з даного профілю;
2. користувачі, які купили автомобіль з цього профіль.

Для перевірки під першу категорію, знаходимо всі товари, які належать даній людині та які вже були продано. В кожному товарі провіряємо чи поле «is_sold_with_user_id» дорівнює ідентифікатору людини, яка робить оцінку. Якщо такого товару знайдено не було, переходимо до наступної категорії. Провіряємо всі товари, які також були продані, але по полі «is_sold_to».

Якщо права користувача на голосування не відносяться до однієї з двох категорій, то повертаємо помилку «403» – немає прав. В іншому випадку додаємо даний рейтинг в масив голосування «rate_array», а також знаходимо нове середнє значення оцінки та записуємо його в поле «rating».

Для даного обчислення був використаний метод reduce[1] (рис. 3.35), який відноситься до JavaScript ES6. Він приймає на вхід 2 параметри – колбек та початкове значення.  Колбек функція також приймає на вхід 2 параметри – попереднє значення обчислення та поточний масив елементу. Сумуємо до попереднього значення поточну оцінку та в результаті отримуємо суму всіх оцінок користувача. Розділяємо дане значення на кількість елементів та отримуємо середнє значення.

```
userToRate.rating = userToRate.rate_array.reduce((prev, curr) => {
  return prev + curr.value
}, 0) / userToRate.rate_array.length
```

Рисунок 3.35 - Обчислення середньої оцінки

## 3.12 Замовлення автомобіля

Для створення нового замовлення використаємо ті ж поля, що і для оголошення(окрім зображень). Після їх валідації, створимо новий документ в колекції «orders». Щоб податись на реалізацію замовлення потрібно зробити виклик «/order/application» методом POST з полями дати реалізації, ціни та повідомлення. Для того, щоб додати даний запит в базу, спочатку провіримо чи він вже існує. Дана операція зображена на рисунку 3.36.

```
const orderFromPersonAlreadyExists = await Order.aggregate([
  { $unwind: '$applications' },
  { $group: { _id: '$applications'}},
  { $lookup: { from: 'applications', localField: '_id', foreignField: '_id', as: 'app_id' } },
  { $unwind: '$app_id' },
  { $group: { _id: '$app_id.applicant_id'}},
  { $match: { _id: Mongoose.Types.ObjectId(current_user_id) }}
]).then(value => {
  return value[0]
})
```

Рисунок 3.36 - Перевірка на наявність
запиту від користувача

Якщо такого запиту ще немає, то додамо його в поле applications до документу замовлення.

З боку власника оголошення можна дати згоду на виконання запиту, або його відхилити. У разі згоди, також потрібно записати в поле «selected_application» посилання на погоджений запит.

Якщо запит був погоджений, то власний оголошення також може відмітити його як виконане. Для цього йому необхідно зробити виклик «order/complete» методом «PUT», а також передати ідентифікатор свого замовлення. Після виклику запишуться поля «is_done» в значення true та «is_done_at» в значення теперішньої дати.

### 3.13 Перегляд профілю

Є можливість переглянути чужий профіль користувача. В такому випадку окрім інформації про профіль, буде переданий список всіх товарів, які відносяться до даного користувача. Окрім приватний товарів, сюди також будуть добавлені товари, які просуває даний користувач. Людина, яка переглядає цей список не може відрізнити який товар відноситься до даного продавця, а який ні.

Окрім цього, користувачу необхідно повернути інформацію про те, чи може він виставити рейтинг даному профілю. Для цього необхідно зробити перевірки по категоріях, які були описані в розділі 3.11.

Переглядаючи власний профіль, користувач може окремо переглянути свої товари, та товари, які він просуває.

Методом PUT по шляху «/user/update» користувач може передати такі ж деталі, як при реєстрації та їх оновити(рис. 3.37). Оскільки може передати не всю інформацію, то варто обновити лише поля, які не є пустими.

```
const settingsToChange = filterNulls({
  phone_number,
  location,
  user_name
})

for (let propertyName in settingsToChange) {
  user[propertyName] = settingsToChange[propertyName]
}

await user.save()
```

Рисунок 3.37 - Оновлення деталей користувача

# 4 БІЗНЕС ПЛАН

## 4.1 Резюме

З розвитком інтернету, який все більше стає більше невід'ємною частиною нашого життя, багато розмов про процес купівлі автомобіля перенеслись в Інтернеті. Ця «цифрова передача з уст в уста» може бути надзвичайно потужною, оскільки люди тепер не просто розповідають близьким друзям про свій досвід – вони розповідають усім. Що це означає? Зараз споживачі публікують свою думку в соціальних мережах, тому сотні, тисячі і навіть мільйони людей можуть бачити, що думають інші. Статистика оглядів в Інтернеті стверджує, що 84% користувачів оцінюють цифрові відгуки настільки, наскільки цінують прямі особисті рекомендації від друга чи члена сім'ї.

Зараз понад 50% клієнтів шукають послуги за допомогою онлайн-платформ, тому важливо не лише показати людині товар, а зрозуміти наскільки доброчесним є його продавець.

Дана платформа буде використовуватись для продажу, купівлі та замовлення авто. Орієнтована на людей, які хочуть продати авто, чи вибрати його зі списку наявних, або навіть описати бажане авто та замовити його. За допомогою системи рейтингу можна легко оцінити з яким саме продавцем варто мати справу. Також в додатку існує реферальна система, за допомогою якої можна заробити на чужій продажі, просто реалізувавши товар. Отже, за допомогою даного проекту також існує можливість дистанційного заробітку.

Хороша оцінка вартості має важливе значення для втримання витрат проекту в межах бюджету. Багато витрат може виникнути протягом життєвого циклу розробки проекту і точний метод оцінки може бути різницею між вдалим і невдалим планами. Оцінку витрат можна легко змоделювати, проте

важко її реалізувати. Проекти несуть ризики, а ризики приносять несподівані витрати.

Оцінка витрат - це процес, який враховує ці фактори і розраховує бюджет, який відповідає фінансовим зобов'язанням, необхідним для успішного проекту. Оцінка вартості проекту стосується всього, починаючи від побудови макету до розробки повноцінного додатку. Це все коштує грошей, тому чим точнішою буда загальна сума, тим легше можна досягти поставленої мети.

Для реалізації технічної сторони даного проекту необхідно 200 000грн. Також ще 120 000грн. необхідно для проведення маркетингової компанії. Основне джерело фінансування – пошук зацікавлених інвесторів. Орієнтований дохід за перший рік роботи – 1 000 000грн, основу якого буде складати прибуток з рекламних компаній.

Окрім основних витрат на проект, необхідно орендувати vps-сервер та доменне ім’я. Орієнтований розхід на рік – не більше 25 000грн. Організаційно-правовою формою ведення даного бізнесу є партнерство.

За розрахунками за перший рік роботи чистий прибуток від платформи складе мінімум 500 000грн. Оскільки ідея проекту є новою, а тема з продажу автомобілів є популярною на даний момент в Україні, то відповідно існують досить високі шанси успіху.

### 4.2 Маркетинг

Побудова дивовижного продукту для бізнесу - це половина роботи. Так, більше 70 відсотків людей по всьому світу володіють смартфоном. Тому майбутнє маркетингу, безумовно, лежить у мобільних додатках. Отже, платформа повинна підтримувати адаптивний дизайн.

Наступним важливим кроком розвитку додатку є вибір впливової людини для просування. Насправді велика кількість користувачів

стверджують, що вони використовують ту чи іншу програму, оскільки її рекомендували люди, яким вони довіряють або переглядають їх в Інтернеті. Отже, варто заплатити за те, щоб соціальні медіа та безпосередні впливові люди були частиною маркетингового плану додатку.

Необхідно вибрати правильну людину. Треба переконатися, що вона добре підходить саме для цього проекту. Маркетинговий вплив стосується залучення, тому необхідно вибрати впливових людей із великою кількістю активних підписників, більшість з яких – цільові користувачі. Слід ознайомитись з особистістю та її аудиторією впливу, щоб зрозуміти, що подобається її послідовникам. Бренди іноді роблять помилку у партнерстві з такими людьми, особистість яких принципово несумісна з їх напрямком, що призводить до того, що реклама просто ігнорується.

Маркетинговий план мобільних додатків повинен містити демонстраційне відео на 30-50 секунд, яке чітко і стисло підкреслює цінність додатку. Люди – візуальні істоти, тому варто зацікавити людину красивою картинкою. Необхідно, щоб відео виглядало, як огляд продукту, щоб залучитись увагою користувачів. Розповсюджувати відео необхідно на всіх можливих платформах, в тому числі і платних.

В даному випадку додаток зіткнувся з середнім рівнем конкуренції на ринку. Отже, все рівно необхідно зацікавити користувача використати саме дану платформу.

Результатом роботи над проектом є платформа для продажу та купівлі автомобілів. Отже, основна ціль проекту – торгівля автомобіля та можливість заробітку за допомогою реферальної системи.

Робота над проектом була ретельно спланована та контролювалась в процесі розробки, був реалізований весь задуманий функціонал. Продукт виглядає конкурентноспроможним, тому можна сказати що результат роботи є успішним.

Дана платформа націлена на онлайн ринок торгівлі автомобілями, що зараз є дуже актуальним. З точки зору користувача, продукт може виділитись широким функціоналом та сучасним дизайном.

Такий функціонал, як можливість заробітку за допомогою реалізації чужого автомобіля є унікальним на даний момент та виділяє даний продукт на фоні інших. Також можна виділити таку функцію, як замовлення авто, яка також не часто зустрічається на онлайн ринку.

Після того, як покупець кинув запит на купівлю авто та продавець його погодив, вони повинні зустрітись та узгодити всі інші деталі продажу. За аналогічним принципом відбувається і замовлення автомобіля.

Даний продукт може містити баги та технічні недоробки, що є основним його недоліком. Також є імовірність, що клієнтам сервісу буде важко до кінця зрозуміти його функціонал, особливо ту частину, яка стосується реферальної системи. Проте, всі знайдені помилки будуть виправлятись, а також буде проведена робота над спрощенням реферальної системи, щоб була можливість легко донести її основну ідею до кінцевого користувача.

На даний момент платформа орієнтована на користувачів, які проживають в Україні. В додатку поки що підтримується лише українська мова, проте в майбутньому, якщо проект зможе окупити себе, то поступово він буде розвиватись і в інших сусідніх країнах. Потенційними клієнтами даного сервісу є всі люди старше 18, які цікавляться у купівлі чи продажі автомобіля.

За спостереженнями, в Україні зараз різко зріс попит на автомобілі, особливо це стосується бюджетних моделей. По цій причині додаток даного напрямку є дуже актуальним.

### 4.3 Конкуренти

Існує декілька схожих сервісів, які є лідерами на ринку. Перевагою даної платформи над іншими є унікальний функціонал, який повинен охопити більшу аудиторію потенційних клієнтів.

Аналіз їх сайтів проводиться на регулярній основі, постійно досліджується їх функціонал, його недоліки та переваги. Результати досліджень впливають на розробку даної платформи. Для реклами конкуренти зазвичай використовують сервіси Google, а також короткі телевізійні відео ролики. Багато клієнтів переходять на веб-сайти конкурентів через запити у пошукових системах, тому варто добре налаштувати SEO-оптимазацію, щоб мати змогу змагатись за ключові місця у пошуку.

На жаль немає жодного «підступного трюку», який дозволить домінувати в органічному пошуку.

Проте, можливо "зламати" вербальний "код" і отримати значний органічний пошуковий трафік, аналізуючи веб-сайти конкурентів, щоб знайти можливості використання ключових слів, що не використовуються на них.

Це основа, яка дає змогу створювати оригінальний корисний контент, який шукатимуть відвідувачі.

Більшість людей вважає, що визначення основних тем, про які слід писати, є відносно простим заняттям. Проте варто дослідити хоча б кілька ключових слів, які відносяться до теми, профільтрувати дані, які пов’язані із цими словами, а потім вже починати планувати вміст навколо цього. Тому варто написати кілька сторінок з контентом, які зможуть описати загальний посил веб-сайту, а також в майбутньому розробити блог та регулярно оновлювати пости, оскільки це позитивно впливає на рейтинг сайту в пошукових системах.

### 4.4 Оцінка продаж

Оцінка продаж – це спосіб прогнозування, який допоможе приблизно оцінити майбутні доходити бізнесу. Даний підхід допоможе уникати деяких ризиків або передбачати свої проблеми.

Варто відмітити, що прогнозування продажів - це основна концепція управління бізнесом, яка зазвичай не враховується для інтернет-магазинів, але вона може запропонувати конкурентні переваги при ретельному використанні.

Іноді окрім дослідження конкурентів та аналізування ринку для правильної оцінки свої прибутків, необхідно також зробити багато внутрішньої роботи. Потрібно переконатись, що дані фактори знаходяться в хороших межах:

1. коефіцієнт відвідувань та конверсій: це два показники, які потрібно тримати під контролем, не зважаючи ні на що;
2. джерела трафіку: необхідно дослідити який відсоток трафіку органічний, соціальний або з платних джерел. Якщо є розсилка, то треба переконатись, що вона приносить відвідувачів, в іншому випадку зрозуміти причину проблеми;
3. функціонал: якщо стоїть задача запустити нову функцію в користування, то необхідно уважно стежити за роботою додатку та як активно ця функція використовується;
4. акції: варто придумати можливість додавати більше акцій для користувачів, адже такий підхід збільшить продажі;
5. наявність товарів: завжди необхідно слідкувати за тим, щоб виставлялись нові оголошення з високою частотою. Якщо це не так, потрібно заманити потенційних продавців будь-якими способами, оскільки це основа сайту;

6. персонал: необхідно дослідити чого бракує сайту та вирішити це за допомогою необхідних працівників. Потрібно слідкувати за витратами на їх зарплати та регулювати їх якість та кількість;
7. внутрішня політика: потрібно реалізувати хороше обслуговування клієнтів. Коли клієнт купив неякісне авто, або у нього виникли інші проблеми, то варто придумати як їх можна швидко вирішувати, оскільки такі ситуації сильно впливають на потік відвідувачів.

Також існують зовнішні фактори, які можуть прямо впливати на прибуток. Це може означати або можливість, або загрозу - у будь-якому випадку варто тримати це під контролем.

Конкуренти: потрібно ознайомитися з їхньою активністю, рекламними кампаніями, новими випусками, акціями та знижками та відстежувати їх позиціонування за ключовими словами.

Періоди високого попиту: зрозуміло, що існують періоди, в залежності від яких попит зростає або зменшується. Необхідно користуватись цим та регулювати фінансування на маркетинг в такі часи.

Тенденції: безумства, які раптом з’являються і застають всіх зненацька. Необхідно намагатися передбачити ці тенденції, навіть якщо це помилка. Такі тенденції можуть стосуватися, наприклад, нового революційного функціоналу сайту чи його нової адаптації.

Отже зрозуміло, що оцінка прибутку сайту – це комплексна задача, яку на старті роботи додатку оцінити дуже важко. Можна лише зробити деякі припущення.

За найгіршим сценарієм потоку користувачів, потрібно ризикнути та вкласти більше коштів в рекламу проекту.

За сприятливих умов дохід з проекту буде стабільним, що буде спонукати до розробки нового функціоналу, який допоможе збільшити потік клієнтів.

### 4.5 Підбір персоналу

Для розробки даного проекту необхідно 4 людини: 1 фронтенд розробник, 1 бекенд розробник, 1 веб-дизайнер та проектний менеджер. Основна функція менеджера - переконатися, що команда будує потрібний продукт. Ця особа несе відповідальність за визначення пріоритетності завдань та планування обсягу роботи на наступні кілька місяців. Іншими словами – це точка зв'язку між командою, зацікавленими сторонами та користувачами.

Психологічна сумісність членів команди також може вплинути на результативність навіть більше, ніж їх кваліфікація. Отже, варто також спостерігати за тим, як члени команди ставляться один до одного. Необхідно докласти усіх зусиль для вирішення виникаючих конфліктів та заохочувати партнерство та командний дух.

Потрібно створити середовище, де люди можуть вільно обговорювати та пропонувати ідеї - це створює родючий грунт для вирощування успішного проекту.

### 4.6 Джерела фінансування

Основне джерело фінансування – інвестиції.

Процес пошуку грошей повинен відповідати потребам компанії. Де та як шукати гроші, залежить від самої компанії та того, який тип грошей потрібен[17]. Існує величезна різниця, наприклад, між високорозвиненою Інтернет-компанією, яка шукає венчурне фінансування, та місцевим роздрібним магазином.

Бізнес венчурного капіталу часто неправильно розуміється. Багато компаній-початківців скаржаться на такі підприємства за те, що вони не змогли інвестувати в новий бізнес та ризикувати.

Фахівці з венчурного капіталу шукають підприємства, які, на їхню думку, можуть призвести до великого збільшення вартості бізнесу протягом декількох років. Вони знають, що більшість підприємств з високим рівнем ризику провалюються.

Для даного проекту необхідно залучити ангельські інвестиції.

Хоча ангельські інвестиції дуже схожі на венчурний капітал (і їх часто плутають), є важливі відмінності. По-перше, інвестори-ангели - це групи чи особи, які вкладають власні гроші. По-друге, ангельські інвестори, як правило, вкладають кошти в компанії на більш ранніх стадіях зростання, тоді як венчурний капітал зазвичай чекає кілька років зростання до якогось рівня успіху.

# ВИСНОВКИ

В даній роботі було проведено аналіз предметної області, а саме дослідження сучасного онлайн ринку автомобілів. Було описано основні ризики при купівлі автомобілів та як їх можна обійти. Також вказано основні чинники, які впливають на людину, коли вона вперше відкриває веб-сайт. На нашу думку, представлено найкращу будову веб-сторінки, яка допоможе зацікавити користувача та донести йому важливу інформацію. Були виділені основні на даний момент фаворити, вказані їхні переваги та недоліки.

Досліджено сучасні засоби розробки програмного забезпечення та окреслено їх основні особливості. Описано платформу Node.js, вказано чому та в яких випадках її варто використовувати для розробки.

Було вказано основні особливості проекту, спроектовано архітектуру, яку легко підтримувати та розширювати. Описано як можна інтегрувати сторонні сервіси з Node.

Розроблено бекенд частину додатку, яка використовує найсучасніші методи розробки та реалізовує весь задуманий функціонал. Описано права користувачів, найкращі способи їх авторизації, безпечне збереження паролів та їх відновлення.

Даний додаток готовий до комерційного запуску та зможе використовуватись людьми, які хочуть придбати чи продати автомобіль та відкриває для них увесь реалізований функціонал.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

## REFERENCES


1. Колуччі Б., Копот М., Кіркбрид Ф., Річардсон Н. Професійний Javascript. 2019. 664 с.
2. Ореллі Т. Розробка з Node, 2 видання. 2016. 288 с.
3. Документація Node.js. URL: https://nodejs.org/uk/docs/ (дата звернення: 14.01.2020)
4. Дослідження першого враження при відкритті сайту. URL: https://www.creativepl.com/blog/how-long-do-you-have-make-first-impression-online-marketing (дата звернення: 14.01.2020)
5. Патерни проектування. URL: https://echo.lviv.ua/dev/5432 (дата звернення: 22.01.2020)
6. Поняття HTTP протокола. URL: https://uk.wikipedia.org/wiki/HTTP (дата звернення: 01.02.2020)
7. Коди стану HTTP. URL: https://uk.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D0%BA%D0%BE%D0%B4%D1%96%D0%B2_%D1%81%D1%82%D0%B0%D0%BD%D1%83_HTTP (дата звернення: 01.02.2020)
8. Express довідник. URL: https://expressjs.com/ru/ (дата звернення: 10.02.2020)
9. Поняття package.json. URL: https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/ (дата звернення: 22.02.2020)

10. Налаштування Node проекту. URL:
https://www.twilio.com/docs/usage/tutorials/how-to-set-up-your-node-js-and-express-development-environment
(дата звернення: 01.03.2020)
11. Документація Node.js body-parser. URL:
https://www.npmjs.com/package/body-parser
(дата звернення: 05.03.2020)
12. EcmaScript 6. URL: http://es6-features.org/#Constants
(дата звернення: 05.03.2020)
13. Потоки Node. URL: https://webdraftt.com/tutorial/nodejs/streams
(дата звернення: 10.03.2020)
14. SEO для початківців. URL:
https://support.google.com/webmasters/answer/7451184?hl=ru
(дата звернення: 12.03.2020)
15. Amazon S3 API Вступ. URL:
https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
(дата звернення: 22.03.2020)
16. Як розробити REST в Node. URL: https://rAPIdAPI.com/blog/nodejs-express-REST-API-example
(дата звернення: 22.03.2020)
17. Як отримати інвестиції у проект. URL:
https://www.quicksprout.com/how-to-get-your-startup-funded/
(дата звернення: 30.04.2020)
18. Ореллі Т. Вивчення Javascript патернів. 2012. 254с.
19. Ореллі Т. Патерни JavaScript: побудова кращих програм за допомогою патернів програмування та дизайну. 2010. 236с.
20. Закас. М. Об’єктно-орієнтоване програмування JavaScript. 2014. 120с.
21. M. Kozlenko, V. Tkachuk, and M. Dutchak, "Software implementation of microcomputer based intrusion detection and prevention system with binary

neural network," in Proc. 2nd International Scientific-Practical Conference "Problems of Cyber Security of Information and Telecommunication Systems" (PCSITS), O. Oksiiuk et al, Eds. Taras Shevchenko National University of Kyiv, Kyiv, Ukraine, Apr. 11-12, 2019, pp. 371-373.

22. I. Lazarovich and Y. Nikolaychuk, "Method of randomization and its application for adaptive data compression," Second IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings, Lviv, 2003, pp. 362-364, doi: 10.1109/IDAACS.2003.1249587.
23. П. Федорук і М. Дутчак, “Побудова бази знань адаптивних систем дистанційного навчання на основі фреймової та продукційної моделей представлення знань,” Управляючі системи і машини (УСіМ), №5, с.35-42, 2012.

# ДОДАТОК А

## Код проекту

www.js

```
#!/usr/bin/env
node

  #!/usr/bin/env node

  /**
   * Module dependencies.
   */

  var app = require('./app');
  var debug = require('debug')('join-auto:server');
  var http = require('http');

  /**
   * Get port from environment and store in Express.
   */

  var port = normalizePort(process.env.PORT || '3001');
  app.set('port', port);

  /**
   * Create HTTP server.
   */

  var server = http.createServer(app);

  /**
   * Listen on provided port, on all network interfaces.
   */

  server.listen(port);
  server.on('error', onError);
  server.on('listening', onListening);

  /**
   * Normalize a port into a number, string, or false.
   */

  function normalizePort(val) {
    var port = parseInt(val, 10);

    if (isNaN(port)) {
      // named pipe
      return val;
    }

    if (port >= 0) {
      // port number
      return port;
    }

    return false;
  }

  /**
```

```
    * Event listener for HTTP server "error" event.
    */

   function onError(error) {
     if (error.syscall !== 'listen') {
       throw error;
     }

     var bind = typeof port === 'string'
       ? 'Pipe ' + port
       : 'Port ' + port;

     // handle specific listen errors with friendly messages
     switch (error.code) {
       case 'EACCES':
         console.error(bind + ' requires elevated privileges');
         process.exit(1);
         break;
       case 'EADDRINUSE':
         console.error(bind + ' is already in use');
         process.exit(1);
         break;
       default:
         throw error;
     }
   }

   /**
    * Event listener for HTTP server "listening" event.
    */

   function onListening() {
     var addr = server.address();
     var bind = typeof addr === 'string'
       ? 'pipe ' + addr
       : 'port ' + addr.port;
     debug('Listening on ' + bind);
   }

app.js
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
const Mongoose = require('mongoose')
const config = require('./config.js')
const formidableMiddleware = require('express-formidable');

Mongoose.connect(config.BD_BASE_URL,       {       useNewUrlParser:       true,
useUnifiedTopology: true })
  .then(() => {
  console.log('connected success')
  })
  .catch(err => {
    console.log('connection error')
    console.log(err)
})

var app = express();
```

```
app.options('*', function (req, res) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header('Access-Control-Allow-Methods', '*');
  res.header("Access-Control-Allow-Headers", "*");
  res.end();
});

app.use('*', function (req, res, next) {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Methods', '*');
  res.header('Access-Control-Allow-Headers', '*');
  next();
});

var indexRouter = require('./routes/index');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
// app.use(formidableMiddleware({
//   encoding: 'utf-8',
//   multiples: true, // req.files to be arrays of files
// }));



app.use('/', indexRouter);

app.use('*', function (req, res, next) {
  return res.status(404).json({
    error: 'Not found'
  })
});

app.use(function (err, req, res, next) {
  if (res.headersSent) {
    return next(err);
  }

  if (!err.yupError && !(err.value && err.value.yupError)) {
    return res.status(err.status || 500).json({
      stack: err.stack,
      message: err.message
    });
  }

  if (!err.inner.length) {
    return res.status(422).json([
      {
        field: err.path,
        error: err.errors[0]
      }
    ]);
  }

  const errors = err.inner.map(item => {
    return {
      field: item.path,
      error: item.errors[0]
    }
  })
```

```
  return res.status(422).json(errors);
});

module.exports = app;

router.js
var express = require('express');
var router = require('express-promise-router')();

const authRouter = require('./auth_routes.js')
const vehicleRouter = require('./vehicle_routes.js')
const vehicleOrderRouter = require('./vehicle_order_routes.js')
const userRouter = require('./user_routes.js')

router.use(authRouter)
router.use(vehicleRouter)
router.use('/order', vehicleOrderRouter)
router.use('/user', userRouter)

module.exports = router;

auth_routes.js
const Router = require('express-promise-router')
const authRouter = new Router()

const { login,
  register,
  forgotPassword,
  resetPassword
} = require('@controllers/auth.controller.js')

const registerValidator = require('@/validators/register.js')
const loginValidator = require('@/validators/login.js')
const forgotPasswordValidator = require('@/validators/ForgotPassword.js')
const ResetPasswordValidator = require('@/validators/ResetPassword.js')

authRouter.post('/login', loginValidator, login)
authRouter.post('/sign-up', registerValidator, register)
authRouter.post('/forgot-password', forgotPasswordValidator, forgotPassword)
authRouter.post('/reset-password', ResetPasswordValidator, resetPassword)

module.exports = authRouter

user_routes.js
const Router = require('express-promise-router')
const userRouter = new Router()

const checkAuth = require('@/middleware/checkAuth.js')
const ViewUserInfoValidator = require('@/validators/user/ViewUserInfo.js')

const {
  ViewUserInfo,
  ViewUserProducts,
  UserUpdateDetails,
  ViewOwnProfile,
  ViewUserSharedProducts,
  ViewUserOwnProducts,
  RateUser
} = require('@/controllers/user.controller.js')

userRouter.post('/rate', checkAuth, RateUser)
userRouter.put('/update', checkAuth, UserUpdateDetails)
userRouter.get('/my-profile', checkAuth, ViewOwnProfile)
```

```
userRouter.get('/products/shared', checkAuth, ViewUserSharedProducts)
userRouter.get('/products/own', checkAuth, ViewUserOwnProducts)
userRouter.get('/products/:id', ViewUserInfoValidator, ViewUserProducts)
userRouter.get('/:id', ViewUserInfoValidator, ViewUserInfo)

module.exports = userRouter
```

vehicle_order_routes.js

```
const Router = require('express-promise-router')
const checkAuth = require('@/middleware/checkAuth.js')
const vehicleOrderRouter = new Router()

const {
  addNewOrder,
  ViewOrder,
  ViewOrderApplications,
  MakeOrderApplication,
  ApproveApplication,
  DeclineApplication,
  ApplicationHandler,
  MakeOrderDone
} = require('@/controllers/order.controller.js')

const AddNewOrderValidator = require('@/validators/Order/AddNewOrder.js')
const ViewOrderValidator = require('@/validators/Order/ViewOrder.js')
const                      ViewOrderApplicationsValidator                    =
require('@/validators/Order/ViewOrderApplications.js')
const                       ApplicationHandlerValidator                      =
require('@/validators/Order/ApplicationHandler.js')
const MakeOrderDoneValidator = require('@/validators/Order/MakeOrderDone.js')

vehicleOrderRouter.post('/post', checkAuth, AddNewOrderValidator, addNewOrder)
vehicleOrderRouter.get('/:id', ViewOrderValidator, ViewOrder)
vehicleOrderRouter.get('/applications/:id',    ViewOrderApplicationsValidator,
ViewOrderApplications)
vehicleOrderRouter.post('/application', checkAuth, MakeOrderApplication)
vehicleOrderRouter.post('/application/approve',                     checkAuth,
ApplicationHandlerValidator, ApplicationHandler, ApproveApplication)
vehicleOrderRouter.post('/application/decline',                     checkAuth,
ApplicationHandlerValidator, ApplicationHandler, DeclineApplication)
vehicleOrderRouter.put('/complete',     checkAuth,     MakeOrderDoneValidator,
MakeOrderDone)

module.exports = vehicleOrderRouter
```

vehicle_routes.js

```
const Router = require('express-promise-router')
const vehicleRouter = new Router()

const { postProduct,
  getProductDetails,
  getProducts, buyAuto,
  handleBuyRequst
} = require('@/controllers/vehicle.contoller.js')

const checkAuth = require('@/middleware/checkAuth.js')
const addToViewsHistory = require('@/middleware/addToViewsHistory.js')

const                              approveBuyRequest                          =
require('@/controllers/vehicle/approveBuyRequest.js')
const                              declineBuyRequest                          =
require('@/controllers/vehicle/declineBuyRequest.js')
```

```
const                          addProductToOwnList                          =
require('@/controllers/vehicle/addProductToOwnList.js')
const                       removeProductFromOwnList                        =
require('@/controllers/vehicle/removeProductFromOwnList.js')

const PostVehicleValidator = require('@/validators/PostVehicle.js')
const GetVehicleValidator = require('@/validators/GetVehicle.js')
const GetVehicleListValidator = require('@/validators/GetVehicleList.js')
const BuyAutoValidator = require('@/validators/Vehicle/BuyAuto.js')
const                        HandleBuyRequestValidator                      =
require('@/validators/Vehicle/HandleBuyRequest.js')
const                      AddProductToOwnListValidator                     =
require('@/validators/Vehicle/AddProductToOwnList.js')
const                   RemoveProductFromOwnListValidator                   =
require('@/validators/Vehicle/RemoveProductFromOwnList.js')

vehicleRouter.post('/product/post',      checkAuth,      PostVehicleValidator,
postProduct)
vehicleRouter.get('/products/:id',       checkAuth,       GetVehicleValidator,
addToViewsHistory, getProductDetails)
vehicleRouter.get('/products', GetVehicleListValidator, getProducts)
vehicleRouter.post('/product/buy', checkAuth, BuyAutoValidator, buyAuto)
vehicleRouter.post('/product/approve',  checkAuth,  HandleBuyRequestValidator,
approveBuyRequest, handleBuyRequst)
vehicleRouter.post('/product/decline',  checkAuth,  HandleBuyRequestValidator,
declineBuyRequest, handleBuyRequst)
vehicleRouter.post('/product/add',   checkAuth,   AddProductToOwnListValidator,
addProductToOwnList)
vehicleRouter.delete('/product/:id',                                    checkAuth,
RemoveProductFromOwnListValidator, removeProductFromOwnList)

module.exports = vehicleRouter

user.js
const Mongoose = require('mongoose')
const MongooseSchema = require('mongoose').Schema
const bcrypt = require('bcrypt')
const randomString = require('randomstring')
const jwt = require('jsonwebtoken')
const config = require('@/config')
const md5 = require('md5')

const UserSchema = new MongooseSchema({
  email: String,
  user_name: String,
  password: String,
  created_at: Date,
  updated_at: Date,
  phone_number: String,
  location: String,
  rate_array: [
    {
      from_user: { type: Mongoose.Types.ObjectId, ref: 'Vehicle' },
      created_at: Date,
      value: Number
    }
  ],
  rating: Number,
  is_admin: Boolean,
  email_code: String,
  email_verified: false,
  shared_products: [{ type: Mongoose.Types.ObjectId, ref: 'Vehicle' }],
  referal_token: String,
```

```
  views_history: [
    {
      product_id: { type: Mongoose.Types.ObjectId, ref: 'Vehicle' },
      token: String,
      created_at: Date
    }
  ]
})

const sendVerificationEmail = () => {

}

UserSchema.pre('save', function () {
  const saltRounds = 10;
  const salt = bcrypt.genSaltSync(saltRounds);

  const hashPassword = bcrypt.hashSync(this.password, salt);
  this.password = hashPassword
  this.email_code = randomString.generate(72)
  this.created_at = new Date()
  this.referal_token = md5(this.email.toLowerCase())

  sendVerificationEmail(this)
})

UserSchema.methods.generateToken = function () {
  return jwt.sign({id: this._id}, config.JWT_KEY)
}

UserSchema.methods.generateForgotPasswordToken = function () {
  return randomString.generate(72)
}

module.exports = Mongoose.model('User', UserSchema)

vehicle.js
const Mongoose = require('mongoose')
const MongooseSchema = require('mongoose').Schema
const config = require('@/config')
const User = require('@/models/User.js')

const VehicleSchema = new MongooseSchema({
  user_owner: { type: Mongoose.Types.ObjectId, ref: 'User' },
  description: String,
  price: Number,
  is_new: Boolean,
  vehicle_type: Number,
  available_for_promote: Boolean,
  promote_compensation: Number,
  promote_compensation_type: Number,
  is_sold: Boolean,
  is_sold_to: { type: Mongoose.Types.ObjectId, ref: 'User' },
  is_sold_with_user_id: { type: Mongoose.Types.ObjectId, ref: 'User' },
  body_type: Number,
  mileage: Number,
  engine: Number,
  transmittion: Number,
  wheel_drive: Number,
  color: String,
  created_at: Date,
  images: [String],
  buy_requests: [
```

```
    {
      user_id: { type: Mongoose.Types.ObjectId, ref: 'User' },
      comment: String,
      price: Number,
      approved: Boolean,
      created_at: Date,
      updated_at: Date,
      approved_at: Date,
      declined_at: Date
    }
  ]
})

VehicleSchema.pre('remove', async function () {
  const users = await User.find({ 'shared_products': this._id})

  users.forEach(user => {
    const     docIndex     =     user.shared_products.findIndex(product     =>
product._id.equals(this._id))
    user.shared_products.splice(docIndex, 1)
    user.save()
  })
})

module.exports = Mongoose.model('Vehicle', VehicleSchema)

ResetPassword.js
const Mongoose = require('mongoose')
const MongooseSchema = require('mongoose').Schema

const ResetPasswordSchema = new MongooseSchema({
  email: String,
  created_at: Date,
  token: String
});

module.exports = Mongoose.model('ResetPassword', ResetPasswordSchema)

Order.js
const Mongoose = require('mongoose')
const MongooseSchema = require('mongoose').Schema
const config = require('@/config')
const User = require('@/models/User.js')

const OrderSchema = new MongooseSchema({
  user_owner: { type: Mongoose.Types.ObjectId, ref: 'User' },
  title: String,
  description: String,
  price_from: Number,
  price_to: Number,
  is_new: Boolean,
  vehicle_type: Number,
  body_type: Number,
  mileage: Number,
  engine: Number,
  transmittion: Number,
  wheel_drive: Number,
  color: String,
  is_done: Boolean,
  is_done_at: Date,
  year_from: Number,
  year_to: Number,
  model: String,
```

```
  selected_application: { type: Mongoose.Types.ObjectId, ref: 'Application' },
  applications: [
    { type: Mongoose.Types.ObjectId, ref: 'Application' }
  ]
})

module.exports = Mongoose.model('Order', OrderSchema)

Application.js
const Mongoose = require('mongoose')
const Schema = Mongoose.Schema

const ApplicationSchema = new Schema({
  applicant_id: { type: Mongoose.SchemaTypes.ObjectId, ref: 'User'},
  order_id: { type: Mongoose.SchemaTypes.ObjectId, ref: 'Order' },
  message: String,
  created_at: Date,
  date_complete_to: Date,
  price: Number,
  is_completed: Boolean,
  approved: Boolean
})

module.exports = Mongoose.model('Application', ApplicationSchema)

auth.contoller.js
const User = require('@/models/User')
const ResetPassword = require('@/models/ResetPassword')
const qs = require('qs')
const yup = require('yup')
const bcrypt = require('bcrypt')

const getUserInfo = (user) => {
  const userParsed = qs.parse(user)

  const newObj = userParsed._doc

  delete newObj.is_admin
  delete newObj.email_code
  delete newObj.__v

  return newObj
}

/**
 * @api {post} /login Login
 * @apiName Login
 * @apiGroup Auth
 *
 * @apiParam {String} email
 * @apiParam {String} password
 * @apiSuccessExample {json} Success-Response:
 *  { "auth_token": "token_goes_here" }
 *  Save this to local storage and add to each request as Authorization header
 */
const login = async (req, res) => {
  const { email, password } = req.body

  const existingUser = await User.findOne({ email })

  if (!existingUser) {
    throw new yup.ValidationError(
      'Check your credentials.',
```

```
      { ...req.body, yupError: true },
      'email'
    )
  }
  const passwordMatch = await bcrypt.compare(password,
existingUser._doc.password);

  if (!passwordMatch) {
    throw new yup.ValidationError(
      'Check your credentials.',
      { ...req.body, yupError: true},
      'email'
    )
  }

  const token = existingUser.generateToken()
  const userDetails = getUserInfo(existingUser)


  return res.status(200).json({
    ...userDetails,
    auth_token: token
  })
}

/**
 * @api {post} /sign-up Register
 * @apiName Register
 * @apiGroup Auth
 *
 * @apiParam {String} email
 * @apiParam {String} user_name
 * @apiParam {String} password
 * @apiParam {String} phone_number
 * @apiParam {String} location
 * @apiSuccessExample {json} Success-Response:
 *  { "auth_token": "token_goes_here" }
 *  Save this to local storage and add to each request as Authorization header
 */

const sendResetEmail = () => {

}

const register = async (req, res) => {
  const { email, user_name, password, phone_number, location} = req.body

  const user = await User.create({
      email,
      user_name,
      password,
      phone_number,
      location,
      is_admin: false,
      shared_products: [],
  })

  const token = user.generateToken()

  const userDetails = getUserInfo(user)


  return res.status(201).json({
```

```
    ...userDetails,
    auth_token: token
  })
}

/**
 * @api {post} /forgot-password Forgot password
 * @apiName Forgot password
 * @apiGroup Auth
 *
 * @apiParam {String} email
 * @apiSuccessExample {json} Success-Response:
 *  { "token": "token_goes_here" }
 *  Reset password token
 */

const forgotPassword = async (req, res) => {
  const { email } = req.body

  const user = await User.findOne({ email })

  if (!user) {
    return res.status(400).json({
      email: 'User doesn\'t exist'
    })
  }

  const userDetails = user._doc

  const resetPasswordInstance = await ResetPassword.create({
    email: userDetails.email,
    created_at: new Date(),
    token: user.generateForgotPasswordToken()
  })

  sendResetEmail(resetPasswordInstance)

  return res.status(200).json(resetPasswordInstance._doc)
}

/**
 * @api {post} /reset-password Reset password
 * @apiName Reset password
 * @apiGroup Auth
 *
 * @apiParam {String} token
 * @apiParam {String} password
 */

const resetPassword = async (req, res) => {
  const { token, password } = req.body

  const resetPasswordInstance = await ResetPassword.findOne({ token })

  if (!resetPasswordInstance) {
    return res.status(422).json({
      password: 'token is invalid'
    })
  }

  const { email, created_at } = resetPasswordInstance._doc
  const now = new Date().getTime()
  const createdAtPlusDay = created_at.getTime() + (1 * 24 * 60 * 60 * 1000)
```

```
  if (now > createdAtPlusDay) {
    await resetPasswordInstance.deleteOne()

    return res.status(422).json({
      password: 'token is expired'
    })
  }

  const saltRounds = 10;
  const salt = bcrypt.genSaltSync(saltRounds);

  const user = await User.findOneAndUpdate({
    email
  }, {
      password: bcrypt.hashSync(password, salt)
  })

  await resetPasswordInstance.deleteOne()

  return res.status(200).json(user._doc)
}

module.exports = {
  login,
  register,
  forgotPassword,
  resetPassword
}
Order.controller.js
const addNewOrder = require('@/controllers/order/addNewOrder.js')
const ViewOrder = require('@/controllers/order/ViewOrder.js')
const                          ViewOrderApplications                          =
require('@/controllers/order/ViewOrderApplications.js')
const                           MakeOrderApplication                          =
require('@/controllers/order/MakeOrderApplication.js')
const                            ApproveApplication                           =
require('@/controllers/order/ApproveApplication.js')
const                            DeclineApplication                           =
require('@/controllers/order/DeclineApplication.js')
const                            ApplicationHandler                           =
require('@/controllers/order/ApplicationHandler.js')
const MakeOrderDone = require('@/controllers/order/MakeOrderDone.js')

module.exports = {
  addNewOrder,
  ViewOrder,
  ViewOrderApplications,
  MakeOrderApplication,
  ApproveApplication,
  DeclineApplication,
  ApplicationHandler,
  MakeOrderDone
}

addNewOrder.js
const Order = require('@/models/Order.js')

/**
 * @api {post} /order/post Post new order
 * @apiName addNewOrder
 * @apiGroup Order
 *
```

```
 * @apiHeader {String} Authorization
 * @apiParam {String} title
 * @apiParam {String} description
 * @apiParam {Number} price_from
 * @apiParam {Number} price_to
 * @apiParam {Number} is_new
 * @apiParam {Number} vehicle_type
 * @apiParam {Number} body_type
 * @apiParam {Number} mileage
 * @apiParam {Number} engine
 * @apiParam {Number} transmittion
 * @apiParam {Number} wheel_drive
 * @apiParam {Number} color
 * @apiParam {String} model
 * @apiParam {Number} year_from
 * @apiParam {Number} year_to
 *
 */

const addNewOrder = async (req, res) => {
  const {
    current_user_id,
    title,
    description,
    price_from,
    price_to,
    is_new,
    vehicle_type,
    body_type,
    mileage,
    engine,
    transmittion,
    wheel_drive,
    color,
    model,
    year_from,
    year_to
  } = req.body

  const order = await Order.create({
    user_owner: current_user_id,
    title,
    description,
    price_from,
    price_to,
    is_new,
    vehicle_type,
    body_type,
    mileage,
    engine,
    transmittion,
    wheel_drive,
    color,
    model,
    year_from,
    year_to,
    applications: []
  })

  return res.status(201).json(order)
}

module.exports = addNewOrder
```

```
ApplicationHandler.js
const Order = require('@/models/Order.js')
const User = require('@/models/User.js')
const Application = require('@/models/Application.js')

/**
 * @api {post} /order/application/approve ApproveApplication
 * @apiName ApproveApplication
 * @apiGroup Order
 *
 * @apiParam {String} application_id
 *
 *
 */

const makeOrderApplication = async (req, res, next) => {
  const {
    current_user_id,
    application_id
  } = req.body

  const application = await Application.findById(application_id)
    .populate({ path: 'order_id' })

  if (!application) {
    return res.status(404).json({
      error: 'Application not found'
    })
  }

  const orderOwner = await User.findById(application.order_id.user_owner)

  if (!orderOwner._doc._id.equals(current_user_id)) {
    return res.status(404).json({
      error: 'Application not found'
    })
  }

  req.body.application = application
  req.body.orderOwner = orderOwner

  next()
}

module.exports = makeOrderApplication

ApproveApplication.js
const Order = require('@/models/Order.js')
const User = require('@/models/User.js')
const Application = require('@/models/Application.js')

/**
 * @api {post} /order/application/approve ApproveApplication
 * @apiName ApproveApplication
 * @apiGroup Order
 *
 * @apiParam {String} application_id
 *
 *
 */

const makeOrderApplication = async (req, res) => {
```

```
  const {
    application_id,
    application
  } = req.body

  if (application.order_id.selected_application) {
    return res.status(403).json({
      error: 'Order already filled'
    })
  }

  application.approved = true
  application.order_id.selected_application = application_id

  await application.save()
  await application.order_id.save()

  return res.status(200).json(application)
}

module.exports = makeOrderApplication

DeclineApplication.js
const Order = require('@/models/Order.js')
const User = require('@/models/User.js')
const Application = require('@/models/Application.js')

/**
 * @api {post} /order/application/decline declineApplication
 * @apiName declineApplication
 * @apiGroup Order
 *
 * @apiParam {String} application_id
 *
 *
 */

const makeOrderApplication = async (req, res) => {
  const {
    current_user_id,
    application_id,
    application,
    orderOwner
  } = req.body

  const existingSelectedApplication =
application.order_id.selected_application

  if (existingSelectedApplication &&
existingSelectedApplication._id.equals(application_id)) {
    application.order_id.selected_application = null
  }

  application.approved = false

  await application.save()
  await application.order_id.save()

  return res.status(200).json({})
}

module.exports = makeOrderApplication
```

```
MakeOrderApplication.js
const Order = require('@/models/Order.js')
const Application = require('@/models/Application.js')
const Mongoose = require('mongoose')
const { USER_BASE_INFO_FIELDS } = require('@/constants.js')

/**
 * @api {post} /order/application makeOrderApplication
 * @apiName makeOrderApplication
 * @apiGroup Order
 *
 * @apiParam {String} id order id
 * @apiParam {String} message
 * @apiParam {Number} price
 * @apiParam {Date} date_complete_to
 *
 *
 */

const makeOrderApplication = async (req, res) => {
  const {
    current_user_id,
    message = null,
    price,
    date_complete_to = null,
    id,
  } = req.body

  const order = await Order.findById(id)
    .populate({ path: 'user_owner', select: USER_BASE_INFO_FIELDS })

  if (!order) {
    return res.status(404).json({
      error: 'Order not found'
    })
  }

  if (order.user_owner._id.equals(current_user_id)) {
    return res.status(403).json({
      error: 'You can not make application for own product!'
    })
  }

  const orderFromPersonAlreadyExists = await Order.aggregate([
    { $unwind: '$applications' },
    { $group: { _id: '$applications'}},
    { $lookup: { from: 'applications', localField: '_id', foreignField: '_id',
as: 'app_id' } },
    { $unwind: '$app_id' },
    { $group: { _id: '$app_id.applicant_id'}},
    { $match: { _id: Mongoose.Types.ObjectId(current_user_id) }}
  ]).then(value => {
    return value[0]
  })

  if (orderFromPersonAlreadyExists) {
      return res.status(403).json({
        error: 'Application already created'
      })
  }

  const application = await Application.create({
    message,
```

```
    price: price || order.price,
    date_complete_to,
    created_at: new Date(),
    is_completed: false,
    applicant_id: current_user_id,
    order_id: id
  })

  order.applications.push(application)

  await order.save()

  return res.status(200).json(application)
}

module.exports = makeOrderApplication
```

MakeOrderDone.js

```
const Order = require('@/models/Order.js')
const User = require('@/models/User.js')
const Application = require('@/models/Application.js')

/**
 * @api {put} /order/complete makeOrderDone
 * @apiName makeOrderDone
 * @apiGroup Order
 *
 * @apiParam {String} order_id
 *
 *
 */

const makeOrderApplication = async (req, res, next) => {
  const {
    order_id,
    current_user_id
  } = req.body

  const order = await Order.findById(order_id)
    .populate('selected_application')
    .populate('user_owner')

  if (!order) {
    return res.status(404).json({
      error: 'Order not found'
    })
  }

  if (!order.user_owner._id.equals(current_user_id)) {
    return res.status(404).json({
      error: 'Order not found'
    })
  }

  order.is_done = true
  order.is_done_at = new Date()

  const selectedApplication = order.selected_application

  if (selectedApplication) {
    selectedApplication.is_completed = true
  }
```

```
  await order.save()

  if (selectedApplication) {
    await selectedApplication.save()
  }

  return res.status(200).json()
}

module.exports = makeOrderApplication

ViewOrder.js
const Order = require('@/models/Order.js')
const Mongoose = require('mongoose')
const { USER_BASE_INFO_FIELDS} = require('@/constants.js')

/**
 * @api {get} /order/:id View order
 * @apiName View order
 * @apiGroup Order
 *
 * @apiParam {String} id
 *
 */

const addNewOrder = async (req, res) => {
  const {
    id,
  } = req.params

  const order = await Order.findById(id)
    .populate({ path: 'user_owner', select: USER_BASE_INFO_FIELDS })

  if (!order) {
    return res.status(404).json({
      error: 'Order not found'
    })
  }

  delete order._doc.applications

  return res.status(200).json(order)
}

module.exports = addNewOrder

ViewOrderApplication.js
const Order = require('@/models/Order.js')
const Mongoose = require('mongoose')

const SortWrapper = require('@/helpers/Sort.js')
const filterNulls = require('@/helpers/filterFromUndefined.js')
const      {      PaginationWrapper,      PaginationFormatter      }      =
require('@/helpers/Pagination.js')

/**
 * @api {get} /order/applications/:id viewOrderApplications
 * @apiName viewOrderApplications
 * @apiGroup Order
 *
 * @apiParam {String} id
 * @apiDescription additional queries ?page=1&limit=20
 * Where page is curr page and limit is items per page
```

```
 */

const viewOrderApplications = async (req, res) => {
  const {
    id,
  } = req.params

  const {
    page = 1,
    limit = 20
  } = req.query


  //find by id
  //get just applications field
  //split this field on documents (each value of array will become separate
doc)
  //count that documents and write it to field value
  const count = await Order.aggregate([
    { $match: { _id: Mongoose.Types.ObjectId(id) } },
    { $group: { _id: '$applications' } },
    { $unwind: '$_id' },
    { $count: 'value' },
    { $project: {value: '$value'}}
  ]).then(count => count[0] && count[0].value)

  const pagination = new PaginationWrapper()
    .setPage(page)
    .setLimit(limit)
    .setCount(count || 0)
    .build()

  const applications = await Order.findById(id)
    .populate({
      path: 'applications',
      populate: { path: 'applications' },
      options: {
        sort: { created_at: -1 },
        skip: pagination.getSkippedItemsCount(),
        limit: pagination.getPerPage()
      }
    }).then(order => order.applications)



  // const applications = await Order.aggregate([
  //   { $match: { _id: Mongoose.Types.ObjectId(id) } },
  //   { $sort: { 'applications.created_at': -1 } },
  //   { $group: { _id: '$applications' } },
  //   { $unwind: '$_id' },
  //   { $skip: pagination.getSkippedItemsCount() },
  //   { $limit: pagination.getPerPage() },
  // ]
  // )

  if (!applications) {
    return res.status(404).json({
      error: 'Applications not found'
    })
  }

  return res.status(200).json({
    applications,
```

```
    pagination: new PaginationFormatter(pagination)
  })
}

module.exports = viewOrderApplications
```

user.controller.js

```
const ViewUserInfo = require('@/controllers/user/ViewUserInfo.js')
const ViewUserProducts = require('@/controllers/user/ViewUserProducts.js')
const UserUpdateDetails = require('@/controllers/user/UserUpdateDetails.js')
const ViewOwnProfile = require('@/controllers/user/ViewOwnProfile.js')
const                          ViewUserSharedProducts                          =
require('@/controllers/user/ViewUserSharedProducts.js')
const                           ViewUserOwnProducts                            =
require('@/controllers/user/ViewUserOwnProducts.js')
const RateUser = require('@/controllers/user/RateUser.js')

module.exports = {
  ViewUserInfo,
  ViewUserProducts,
  UserUpdateDetails,
  ViewOwnProfile,
  ViewUserSharedProducts,
  ViewUserOwnProducts,
  RateUser
}
```

RateUser.js

```
const User = require('@/models/User.js')
const Vehicle = require('@/models/Vehicle.js')
const { USER_BASE_INFO_FIELDS } = require('@/constants.js')
const filterNulls = require('@/helpers/filterFromUndefined.js')

/**
 * @api {post} /user/rate RateUser
 * @apiName RateUser
 * @apiGroup User
 *
 * @apiHeader {String} Authorization
 *
 * @apiParam {Number} rate 1-5
 *
 */
const rateUser = async (req, res) => {
  const {
    current_user_id,
    user_id,
    rate
  } = req.body
  const userToRate = await User.findById(user_id)

  if (!userToRate) {
    return res.status(404).json({
      error: 'Not found'
    })
  }

  if (userToRate._id.equals(current_user_id)) {
    return res.status(403).json({
      error: 'Error'
    })
  }
```

```
  let canRate = true
  const userToRateProducts = await Vehicle.find({ user_owner: userToRate._id,
is_sold: true})

  const productToRate = userToRateProducts.find(product => product &&
product.is_sold_with_user_id &&
product.is_sold_with_user_id.equals(current_user_id))
  const userBoughtProduct = userToRateProducts.find(product => product &&
product.is_sold_to && product.is_sold_to.equals(current_user_id))

  if (!productToRate && !userBoughtProduct) {
    canRate = false
  }

  if (!canRate) {
    return res.status(403).json({
      error: 'You can not rate this user'
    })
  }

  const existingRate = (userToRate.rate_array || []).find(rate =>
rate.from_user.equals(current_user_id))

  if (existingRate) {
    return res.status(403).json({
      error: 'You\'ve already rated this user'
    })
  }

  userToRate.rate_array.push({
    from_user: current_user_id,
    value: rate,
    created_at: new Date()
  })
  userToRate.rating = userToRate.rate_array.reduce((prev, curr) => {
    return prev + curr.value
  }, 0) / userToRate.rate_array.length

  await userToRate.save()

  return res.status(200).json(userToRate)
}

module.exports = rateUser
```

UserUpdateProfile.js

```
const User = require('@/models/User.js')
const { USER_BASE_INFO_FIELDS } = require('@/constants.js')
const filterNulls = require('@/helpers/filterFromUndefined.js')

/**
 * @api {put} /user/update updateUserInfo
 * @apiName updateUserInfo
 * @apiGroup User
 *
 * @apiHeader {String} Authorization
 *
 * @apiParam {String} user_name
 * @apiParam {String} password
 * @apiParam {String} phone_number
 * @apiParam {String} location
 *
```

```
 */
const UserUpdateDetails = async (req, res) => {
  const {
    current_user_id,
    phone_number,
    location,
    user_name
  } = req.body
  const user = await User.findById(current_user_id)

  if (!user) {
    return res.status(404).json({
      error: 'Not found'
    })
  }

  if (!user._id.equals(current_user_id)) {
    return res.status(401).json({
      error: 'Permission denied'
    })
  }

  const settingsToChange = filterNulls({
    phone_number,
    location,
    user_name
  })

  for (let propertyName in settingsToChange) {
    user[propertyName] = settingsToChange[propertyName]
  }

  await user.save()

  return res.status(200).json(user)
}

module.exports = UserUpdateDetails

ViewOwnProfile.js
const User = require('@/models/User.js')

/**
 * @api {get} /user/my-profile ViewOwnProfile
 * @apiName ViewOwnProfile
 * @apiGroup User
 *
 * @apiHeader {String} Authorization
 *
 */
const ViewOwnProfile = async (req, res) => {
  const { current_user_id } = req.body

  const user = await User.findById(current_user_id, '-shared_products')

  if (!user) {
    return res.status(404).json({
      error: 'Not found'
    })
  }

  return res.status(200).json(user)
}
```

```
module.exports = ViewOwnProfile

ViewUserInfo.js
const User = require('@/models/User.js')
const { USER_BASE_INFO_FIELDS } = require('@/constants.js')
var Mongoose = require('mongoose');
const Vehicle = require('@/models/Vehicle.js')

/**
 * @api {get} /user/:id ViewUserInfo
 * @apiName ViewUserInfo
 * @apiGroup User
 *
 * @apiParam {id} user_id
 *
 */
const ViewUserInfo = async (req, res) => {
  const { id } = req.params
  const { current_user_id = null } = req.body

  const user = await User.findById(id, USER_BASE_INFO_FIELDS)

  if (!user) {
    return res.status(404).json({
      error: 'Not found'
    })
  }

  let canRate = true;

  const userAlreadyRated = (user.rate_array || []).find(rate => rate.from_user
=== Mongoose.Types.ObjectId(current_user_id))
  const  userToRateProducts  =  await  Vehicle.find({  user_owner:  user._id,
is_sold: true })

  const userHelpedToSell = userToRateProducts.find(product => {
    return       product       &&       product.is_sold_with_user_id       &&
product.is_sold_with_user_id.equals(current_user_id)
  })
  const userBoughtProduct = userToRateProducts.find(product => {
    return          product          &&          product.is_sold_to          &&
product.is_sold_to.equals(current_user_id)
  })

  if ((!userHelpedToSell && !userBoughtProduct) || userAlreadyRated) {
    canRate = false
  }

  return res.status(200).json({...user._doc, can_rate: canRate || false})
}

module.exports = ViewUserInfo

ViewUserOwnProducts.js
const User = require('@/models/User.js')
const Vehicle = require('@/models/Vehicle.js')
const      {      PaginationWrapper,      PaginationFormatter      }      =
require('@/helpers/Pagination.js')
const Mongoose = require('mongoose')

/**
 * @api {get} /user/products/own ViewUserOwnProducts
```

```
 * @apiName ViewUserOwnProducts
 * @apiGroup User
 * @apiDescription Api only for private user profile view
 * @apiHeader {String} Authorization
 *
 */
const ViewUserOwnProducts = async (req, res) => {
  const { current_user_id } = req.body

  const {
    page = 1,
    limit = 20
  } = req.query

  const count = await Vehicle.countDocuments({ user_owner: current_user_id })

  const pagination = new PaginationWrapper()
    .setPage(page)
    .setLimit(limit)
    .setCount(count)
    .build()

  const products = await Vehicle.find({ user_owner: current_user_id})
    .sort({created_at: -1 })
    .skip(pagination.getSkippedItemsCount())
    .limit(pagination.getPerPage())

  return res.status(200).json({
    products,
    pagination: new PaginationFormatter(pagination)
  })
}

module.exports = ViewUserOwnProducts

ViewUserProducts.js
const User = require('@/models/User.js')
const Vehicle = require('@/models/Vehicle.js')
const      {      PaginationWrapper,      PaginationFormatter      }      =
require('@/helpers/Pagination.js')
const Mongoose = require('mongoose')

/**
 * @api {get} /user/products/:id ViewUserProducts
 * @apiName ViewUserProducts
 * @apiGroup User
 *
 * @apiParam {id} user_id
 *
 */
const ViewUserProducts = async (req, res) => {
  const { id } = req.params

  const {
    page = 1,
    limit = 20
  } = req.query

  const concatItemsRequestSchema = [
      {
        $match: { _id: Mongoose.Types.ObjectId(id) },
      },
      {
```

```
        $group: { _id: '$shared_products' }
      },
      {
        $unwind: '$_id'
      },
      {
        $lookup: { from: 'vehicles', localField: '_id', foreignField: '_id',
as: 'product' }
      },
      {
        $group: { _id: '$product' }
      },
      {
        $unwind: '$_id'
      },
      {
        $replaceRoot: { newRoot: '$_id' }
      },
      {
        $group: { _id: Mongoose.Types.ObjectId(id), shared_products: { $push:
'$$ROOT' } }
      },
      {
        $lookup:  { from:  'vehicles',  localField:  '_id',  foreignField:
'user_owner_id', as: 'own_products' }
      },
      {
        $project:   {   items:   {   $concatArrays:   ['$shared_products',
'$own_products'] }, _id: 0 }
      },
      {
        $unwind: '$items'
      },
      {
        $replaceRoot: { newRoot: '$items' }
      }
  ]

  const productsCount = await User.aggregate([
    ...concatItemsRequestSchema,
    { $count: 'count' }
  ]).then(value => {
    return value[0] && value[0].count
  })

  const pagination = new PaginationWrapper()
    .setPage(page)
    .setLimit(limit)
    .setCount(productsCount || 0)
    .build()

  const products = await User.aggregate([
    ...concatItemsRequestSchema,
    {
      $sort: { created_at: -1 },
    },
    {
      $skip: pagination.getSkippedItemsCount(),
    },
    {
      $limit: pagination.getPerPage()
    },
    {
```

```
      $project: { buy_requests: 0, is_sold_with_user_id: 0}
    }
  ])

  return res.status(200).json({
    products,
    pagination: new PaginationFormatter(pagination)
  })
}

module.exports = ViewUserProducts
```

ViewUserSharedProducts.js

```
const User = require('@/models/User.js')
const Vehicle = require('@/models/Vehicle.js')
const      {      PaginationWrapper,      PaginationFormatter      }      =
require('@/helpers/Pagination.js')
const Mongoose = require('mongoose')

/**
 * @api {get} /user/products/shared ViewUserSharedProducts
 * @apiName ViewUserSharedProducts
 * @apiGroup User
 * @apiDescription Api only for private user profile view
 * @apiHeader {String} Authorization
 *
 */
const ViewUserSharedProducts = async (req, res) => {
  const { current_user_id } = req.body

  const {
    page = 1,
    limit = 20
  } = req.query

  const concatItemsRequestSchema = [
    {
      $match: { _id: Mongoose.Types.ObjectId(current_user_id) },
    },
    {
      $group: { _id: '$shared_products' }
    },
    {
      $unwind: '$_id'
    },
    {
      $replaceRoot: { newRoot: '$_id' }
    }
  ]

  const productsCount = await User.aggregate([
    ...concatItemsRequestSchema,
    { $count: 'count' }
  ]).then(value => {
    return value[0] && value[0].count
  })

  const pagination = new PaginationWrapper()
    .setPage(page)
    .setLimit(limit)
    .setCount(productsCount || 0)
    .build()
```

```
  const products = await User.aggregate([
    ...concatItemsRequestSchema,
    {
      $sort: { created_at: -1 },
    },
    {
      $skip: pagination.getSkippedItemsCount(),
    },
    {
      $limit: pagination.getPerPage()
    }
  ])

  return res.status(200).json({
    products,
    pagination: new PaginationFormatter(pagination)
  })
}

module.exports = ViewUserSharedProducts

vehicle.controller.js
const User = require('@/models/User')
const Vehicle = require('@/models/Vehicle.js')
const getProducts = require('@/controllers/vehicle/getProducts.js')
const buyAuto = require('@/controllers/vehicle/buyAuto.js')
const handleBuyRequst = require('@/controllers/vehicle/handleBuyRequst.js')
const upload = require('src/helpers/uploadImages.js');
const yup = require('yup')

/**
 * @api {post} /product/post postProduct
 * @apiName postProduct
 * @apiGroup Product
 *
 * @apiHeader {String} Authorization token
 *
 * @apiParam {String} id user owner id
 * @apiParam {String} description
 * @apiParam {Number} price
 * @apiParam {Number} is_new 0 or 1
 * @apiParam {Number} vehicle_type
 * @apiParam {Number} available_for_promote  0 or 1
 * @apiParam {Number} promote_compensation_type
 * @apiParam {Number} body_type
 * @apiParam {Number} mileage
 * @apiParam {Number} engine
 * @apiParam {Number} transmittion
 * @apiParam {Number} wheel_drive
 * @apiParam {Number} color
 * @apiParam {Array} Images
 */

const uploadFile = upload.array('images', 10)

const postProduct = async (req, res) => {
  const {
    description,
    price,
    is_new,
    vehicle_type,
    available_for_promote,
    promote_compensation,
```

```
    promote_compensation_type,
    body_type,
    mileage,
    engine,
    transmittion,
    wheel_drive,
    color,
    current_user_id,
  } = req.body

  const userOwner = await User.findById(current_user_id)

  if (!userOwner) {
    return res.status(404).json({
      'email': 'User not found'
    })
  }

  const uploadPromise = new Promise((resolve, rej) => {
    uploadFile(req, res, err => {
      if (err) {
        return res.status(422).json([
          {
            field: 'images',
            error: err.message
            }
          ])
      }

      const images = req.files.map(image => image.location)
      resolve(images)
    })
  })

  const images = await uploadPromise

  const vehicle = await Vehicle.create({
    user_owner: current_user_id,
    description,
    price,
    is_new,
    vehicle_type,
    available_for_promote,
    promote_compensation,
    promote_compensation_type,
    is_sold: false,
    body_type,
    mileage,
    engine,
    transmittion,
    wheel_drive,
    color,
    created_at: new Date(),
    images
  })

  return res.status(200).json(vehicle)
}

/**
 * @api {get} /products/:id getProductDetails
 * @apiName getProductDetails
 * @apiGroup Product
```

```
 *
 *
 * @apiParam {String} id id of product
 */

const getProductDetails = async (req, res) => {
  const { vehicle } = req.body

  return res.status(200).json(vehicle)
}


module.exports = {
  postProduct,
  getProductDetails,
  getProducts,
  buyAuto,
  handleBuyRequst
}

addProductToOwnList.js
const Vehicle = require('@/models/Vehicle.js')
const User = require('@/models/User.js')
const { USER_BASE_INFO_FIELDS } = require('@/constants.js')

/**
 * @api {post} /product/add addProductToOwnList
 * @apiDescription Add foreigh product to own list
 * @apiName addProductToOwnList
 * @apiGroup Product
 *
 * @apiHeader {String} Authorization
 *
 * @apiParam {String} product_id id of product
 */

const addProductToOwnList = async (req, res) => {
  const {
    current_user_id,
    product_id
  } = req.body

  const product = await Vehicle.findOne({ _id: product_id})
    .populate({ path: 'user_owner', select: USER_BASE_INFO_FIELDS})

  if (!product) {
    return res.status(404).json({
      error: 'Product not found'
    })
  }

  if (product.user_owner._id.equals(current_user_id)) {
    return res.status(403).json({
      error: 'Product already added'
    })
  }

  const user = await User.findById(current_user_id)

  const shared_products = user.shared_products || []

  const     productAreadyShared     =     shared_products.find(product     =>
product._id.equals(product_id))
```

```
  if (productAreadyShared) {
    return res.status(403).json({
      error: 'Product already added'
    })
  }

  shared_products.push(product)
  user.shared_products = shared_products;

  user.save()

  return res.status(200).json(product)
}

module.exports = addProductToOwnList
```

ApproveBuyRequest.js

```
/**
 * @api {post} /product/approve approveBuyRequst
 * @apiDescription When someone wants to buy your product
 * @apiName approveBuyRequst
 * @apiGroup Product
 *
 * @apiHeader {String} Authorization
 *
 * @apiParam {String} product_id id of product
 * @apiParam {String} request_id id of request
 */

const approveBuyRequst = async (req, res, next) => {
  req.body.request_value = true
  next()
}

module.exports = approveBuyRequst
```

BuyAuto.js

```
const User = require('@/models/User')
const Vehicle = require('@/models/Vehicle.js')
const USER_BASE_INFO_FIELDS = 'email user_name phone_number _id location'

/**
 * @api {post} /product/buy buyAuto
 * @apiName buyAuto
 * @apiGroup Product
 *
 * @apiHeader {String} Authorization
 *
 * @apiParam {String} product_id id of product
 * @apiParam {String} comment id of product
 * @apiParam {Number} price price you want buy for
 */

const buyAuto = async (req, res) => {
  let {
    current_user_id,
    product_id,
    comment,
    price
  } = req.body
  product_id = product_id.trim()
  comment = comment.trim()
```

```
  const vehicle = await Vehicle.findOne({ _id: product_id })
    .populate({ path: 'user_owner', select: USER_BASE_INFO_FIELDS })

  const { _id } = vehicle.user_owner
  const ownerVehicleId = _id.toString().trim()

  const { buy_requests } = vehicle._doc

  if (current_user_id === ownerVehicleId) {
    return res.status(403).json({
      error: `Couldn't buy own product.`
    })
  }

  const alreadyApproved = buy_requests.find(item => {
    return item.approved
  })

  if (alreadyApproved) {
    return res.status(403).json({
      error: 'Product already approved'
    })
  }

  const existingRequest = buy_requests.find(item => {
    return item.user_id.toString() === current_user_id
  })

  if (existingRequest) {
    const now = new Date()

    existingRequest.price = price
    existingRequest.approved = null
    existingRequest.updated_at = now
    existingRequest.comment = comment

    await vehicle.save()

    return res.status(200).json({
      approved: false,
      updated_at: now
    })
  }

  buy_requests.push({
    user_id: current_user_id,
    comment,
    approved: null,
    created_at: new Date(),
    updated_at: null,
    price
  })

  await vehicle.save()

  return res.status(201).json({
    approved: false
  })
}

module.exports = buyAuto
```

DeclineBuyRequest.js

```
/**
 * @api {post} /product/decline declineBuyRequst
 * @apiDescription  When someone wants to buy your product.
 * You can even decline it when its already accepted
 * @apiName declineBuyRequst
 * @apiGroup Product
 *
 * @apiHeader {String} Authorization
 *
 * @apiParam {String} product_id id of product
 * @apiParam {String} request_id id of request
 */

const declineBuyRequst = async (req, res, next) => {
  req.body.request_value = false
  next()
}

module.exports = declineBuyRequst
```

getProducts.js

```
const Vehicle = require('@/models/Vehicle.js')
const SortWrapper = require('@/helpers/Sort.js')
const filterNulls = require('@/helpers/filterFromUndefined.js')
const      {      PaginationWrapper,      PaginationFormatter      }      =
require('@/helpers/Pagination.js')

/**
 * @api {get} /products getProducts
 * @apiName getProducts
 * @apiGroup Product
 *
 *
 * @apiParam {Number} price_from
 * @apiParam {Number} price_to
 * @apiParam {Number} is_new 0 or 1
 * @apiParam {Number} vehicle_type
 * @apiParam {Number} available_for_promote 0 or 1
 * @apiParam {Number} is_sold  0 or 1
 * @apiParam {Number} body_type
 * @apiParam {Number} mileage
 * @apiParam {Number} engine
 * @apiParam {Number} transmittion
 * @apiParam {Number} wheel_drive
 * @apiParam {Number} color
 * @apiParam {String} sort default='created_at'
 * @apiParam {Number} order 1 or -1. Default is -1 (From greater to less)
 * @apiParam {Number} page default is 1
 * @apiParam {Number} limit default is 20. Products per page
 */

module.exports = async (req, res) => {
  let {
    price_from = 0,
    price_to,
    is_new,
    vehicle_type,
    available_for_promote,
    is_sold,
    body_type,
    mileage,
    engine,
```

```
  transmittion,
  wheel_drive,
  color,
  sort = 'created_at',
  order = -1,
  page = 1,
  limit = 20
} = req.query

const AVAILABLE_SORT_TYPES = ['price', 'mileage', 'created_at']
const DEFAULT_SORT_FIELD = 'created_at'

const sortOptions = new SortWrapper()
  .setType(sort)
  .setOkTypes(AVAILABLE_SORT_TYPES)
  .setDefault(DEFAULT_SORT_FIELD)
  .setOrder(order)
  .build()

price_from = price_from ? +(price_from.trim()) : 0
price_to = price_to ? +(price_to.trim()) : null
let newPriceTo;

if (!price_to) {
  newPriceTo = await new Promise((res, rej) => {
    Vehicle
      .find({})
      .select('price')
      .sort({ "price": -1 })
      .limit(1)
      .exec(function (err, doc) {
        if (err) rej(err)

        res(doc[0].price)
      });
  })
}


const findModel = filterNulls({
  price: { $lte: newPriceTo || price_to, $gte: price_from },
  is_new,
  vehicle_type,
  available_for_promote,
  is_sold,
  body_type,
  mileage,
  engine,
  transmittion,
  wheel_drive,
  color
});

const count = await Vehicle.countDocuments(findModel)

const pagination = new PaginationWrapper()
  .setPage(page)
  .setLimit(limit)
  .setCount(count)
  .build()

const items = await Vehicle.find(findModel)
  .sort({ [sortOptions.getFieldName()]: sortOptions.getOrder() })
```

```
    .skip(pagination.getSkippedItemsCount())
    .limit(pagination.getPerPage())

  return res.status(200).json({
    count,
    pagination: new PaginationFormatter(pagination),
    items
  })
}

handleBuyRequest.js
const Vehicle = require('@/models/Vehicle.js')
const User = require('@/models/User.js')
const { USER_BASE_INFO_FIELDS } = require('@/constants.js')

const handleCompensation = async (vehicle, requestToAccept, customer) => {
  if (!vehicle.promote_compensation) {
    return
  }

  const    productIsPromoted    =    customer.views_history.find(product    =>
product.product_id === vehicle._id)

  if (!productIsPromoted) {
    return
  }

  const    IsSoldWithUser    =    await    User.findOne({    referal_token:
productIsPromoted.token })

  if (!BoughtWithUser) {
    return
  }

  vehicle.is_sold_with_user_id = IsSoldWithUser._id

  await vehicle.save()
}

const handleBuyRequst = async (req, res) => {

  let {
    current_user_id,
    product_id,
    request_id,
    request_value
  } = req.body
  request_id = request_id.trim()
  product_id = product_id.trim()

  const vehicle = await Vehicle.findOne({ _id: product_id })
    .populate({ path: 'user_owner', select: USER_BASE_INFO_FIELDS })

  const productOwnerId = vehicle.user_owner._id.toString()

  if (productOwnerId !== current_user_id) {
    return res.status(403).json({
      error: 'Permission denied'
    })
  }

  const acceptedUserRequest = vehicle.buy_requests
    .find(item => item.approved)
```

```
  if (acceptedUserRequest && request_value) {
    return res.status(403).json({
      error: 'Buy request is already accepted'
    })
  }

  const declinedUserRequest = vehicle.buy_requests
    .find(item => item.approved === false)

  if (declinedUserRequest && !request_value) {
    return res.status(403).json({
      error: 'Buy request is already declined'
    })
  }

  const requestToAccept = vehicle.buy_requests.find(item => {
    return item._id.toString() === request_id
  })

  if (!requestToAccept) {
    return res.status(404).json({
      error: 'Request not found'
    })
  }

  requestToAccept.approved = request_value

  if (request_value) {
    const customer = await User.findById({ _id: requestToAccept.user_id })
    requestToAccept.approved_at = new Date()
    requestToAccept.declined_at = null
    vehicle.is_sold = true
    vehicle.is_sold_to = customer._id
    await handleCompensation(vehicle, requestToAccept, customer)
  } else {
    requestToAccept.declined_at = new Date()
    requestToAccept.approved_at = null
  }

  await vehicle.save()

  return res.status(201).json(requestToAccept)
}

module.exports = handleBuyRequst
```

RemoveProductFromOwnList.js

```
const Vehicle = require('@/models/Vehicle.js')
const User = require('@/models/User.js')
const { USER_BASE_INFO_FIELDS } = require('@/constants.js')

/**
 * @api {delete} /product/:id removeProductToOwnList
 * @apiDescription Delete product from own list.
 *  If owner of product call this api,
 * the product will be removed totally!
 * @apiName removeProductToOwnList
 * @apiGroup Product
 *
 * @apiHeader {String} Authorization
 *
 * @apiParam {id} product_id id of product
```

```
 */

const removeProductToOwnList = async (req, res) => {
  const {
    current_user_id,
  } = req.body

  const { id: product_id} = req.params

  const product = await Vehicle.findOne({ _id: product_id })
    .populate({ path: 'user_owner', select: USER_BASE_INFO_FIELDS })

  if (!product) {
    return res.status(404).json({
      error: 'Product not found'
    })
  }

  if (product.user_owner._id.equals(current_user_id)) {
    if (!product.is_sold) {
      await product.delete()
    }

    return res.status(200).json()
  }

  const user = await User.findById(current_user_id)

  const shared_products = user.shared_products || []

  const productAreadySharedIndex = shared_products.findIndex(product =>
product._id.equals(product_id))

  if (productAreadySharedIndex === -1) {
    return res.status(404).json({
      error: 'Product not found'
    })
  }

  shared_products.splice(productAreadySharedIndex, 1)
  user.shared_products = shared_products;

  user.save()

  return res.status(200).json()
}

module.exports = removeProductToOwnList

Pagination.js
class Pagination {
  constructor(perPage, currentPage, count) {
    this._perPage = perPage
    this._currentPage = currentPage
    this._totalPages = Math.max(Math.ceil(count / perPage), 1)
    this.count = count
  }

  getCurrentPage() {
    return Math.min(this._currentPage, this._totalPages)
  }

  getSkippedItemsCount() {
```

```
    return Math.max(0, this.getPerPage() * (this.getCurrentPage() - 1))
  }

  getPerPage() {
    return this._perPage
  }

  getTotalPages() {
    return this._totalPages
  }
}

const MAX_LIMIT = 100
const MIN_LIMIT = 1
const MIN_PAGE_VALUE = 1

class PaginationWrapper {
  setPage(page) {
    this.page = Math.max(MIN_PAGE_VALUE, +page || 0)

    return this
  }

  setLimit(limit) {
    this.limit = Math.max(MIN_LIMIT, Math.min(MAX_LIMIT, +limit || 0))

    return this
  }

  setCount(count) {
    this.count = count || 0

    return this
  }

  build() {
    return new Pagination(this.limit, this.page, this.count, this)
  }
}

class PaginationFormatter {
  constructor(pagination) {
    this.pagination = pagination

    return this.format()
  }

  format() {
    return {
      current_page: this.pagination.getCurrentPage(),
      per_page: this.pagination.getPerPage(),
      total_pages: this.pagination.getTotalPages(),
      count: this.pagination.count
    }
  }
}

module.exports = {
  PaginationWrapper,
  PaginationFormatter
}
```

Sort.js

```
class SortType {
  constructor(type, availableTypes, defaultType, order) {
    let newType;
    let newOrder = order > 0 ? 1 : -1

    if (!availableTypes.includes(type.toLowerCase())) {
      newType = defaultType
    } else {
      newType = type.toLowerCase()
    }

    this.type = newType
    this.order = newOrder
  }

  getFieldName() {
    return this.type
  }

  getOrder() {
    return this.order
  }
}

module.exports = class SortWrapper {
  setType(type) {
    this.type = type

    return this
  }

  setOkTypes(types) {
    this.types = types

    return this
  }

  setOrder(order) {
    this.order = order

    return this
  }

  setDefault(def) {
    this.default = def

    return this
  }

  build() {
    return new SortType(this.type, this.types, this.default, this.order)
  }
}
```

UploadImages.js

```
const aws = require('@/services/aws.js')
const multer = require('multer')
var multerS3 = require('multer-s3')

const fileFilter = (req, file, cb) => {
  if (file.mimetype === 'image/jpeg'
    || file.mimetype === 'image/png'
    || file.mimetype === 'image/jpg') {
```

```
    cb(null, true)
  } else {
    cb(null, false)
    return cb(new Error('Only .png, .jpg and .jpeg format allowed!'));
  }
}

var upload = multer({
  fileFilter: fileFilter,
  limits: {
    fileSize: 1024 * 1024 * 5 //1024 bytes * 1024 kb * 5 we are allowing only
5 MB files
  },
  storage: multerS3({
    s3: aws.s3,
    bucket: aws.bucket,
    cacheControl: 'max-age=31536000',
    metadata: function (req, file, cb) {
      cb(null, { fieldName: file.fieldname });
    },
    contentType: multerS3.AUTO_CONTENT_TYPE,
    acl: 'public-read',
    key: function (req, file, cb) {
      cb(null, Date.now().toString() + file.originalname)
    },
  })
})

module.exports = upload
```

AddToViewsHistory.js

```
const User = require('@/models/User.js')
var Mongoose = require('mongoose');
const Vehicle = require('@/models/Vehicle.js')

const MAX_HISTORY_LENGTH = 50

module.exports = async (req, res, next) => {
  const {
    current_user_id
  } = req.body
  const { id : product_id} = req.params
  const { q = null } = req.query

  const vehicle = await Vehicle.findById(product_id)

  if (!vehicle) {
    return res.status(404).json({
      id: 'Product not found'
    })
  }

  req.body.vehicle = vehicle

  const user = await User.findById(current_user_id)

  if (!user) {
   next()
  }

  if (!q) {
    next()
```

```
    return
  }

  const productAlreadyInHistory = user.views_history.find(product => {
    return product.product_id && product.product_id.equals(product_id)
  })

  if (productAlreadyInHistory) {
    next()

    return
  }

  if (user.views_history.length < MAX_HISTORY_LENGTH) {
    user.views_history.push({
      product_id: Mongoose.Types.ObjectId(product_id),
      token: q,
      created_at: new Date()
    })

    await user.save()

    next()

    return
  }

  await sortedViewsByDate.splice(0, 1, {
    product_id: Mongoose.Types.ObjectId(product_id),
    token: q,
    created_at: new Date()
  })

  user.views_history.shift()
  user.views_history.push({
    product_id: Mongoose.Types.ObjectId(product_id),
    token: q,
    created_at: new Date()
  })

  user.save()

  next()
}

checkAuth.js
const AUTH_HEADER_NAME = require('@/constants.js').AUTH_HEADER_NAME
const jwt = require('jsonwebtoken')
const config = require('@/config')

const checkToken = (token) => {
  try {
    jwt.verify(token, config.JWT_KEY);
    const currentUserId = jwt.decode(token, config.JWT_KEY)

    return currentUserId
  } catch (e) {
    return false
  }
}

module.exports = (req, res, next) => {
  const authToken = req.header(AUTH_HEADER_NAME)
```

```
  const userId = checkToken(authToken)

  if (!userId) {
    return res.status(401).json({})
  }

  req.token = authToken
  req.body.current_user_id = userId.id
  next()
}
```