МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття освітнього ступеня бакалавра

на тему: «**Розробка клієнтської частини маркетплейсу вживаних автомобілів»**

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Кухарчук Сергій Віталійович

Керівник: *викладач кафедри інформаційних технологій та аналітики даних*
Мацевич Денис Володимирович

Рецензент: *ФОП в галузі інформаційних технологій, викладач кафедри менеджменту та маркетингу НаУОА*
Шпак Андрій Григорович

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ
Завідувач кафедри інформаційних технологій та аналітики даних _____
(проф., д.е.н. Кривицька О.Р.)
Протокол №10 від 28 травня 2025 р.

Острог, 2025

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: *Розробка клієнтської частини маркетплейсу вживаних автомобілів*

Автор: *Кухарчук Сергій Віталійович*

Науковий керівник: *викладач-стажист кафедри ІТАД, Мацевич Денис Володимирович*

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: 64 с., 22 рис., 2 табл., 3 додатки, 18 джерел.

Ключові слова: *маркетплейс вживаних автомобілі, клієнтська частина, фронтенд-розробка, Angular, веб-технології, користувацький інтерфейс, дизайн лендінгу, модульна архітектура, Signal API, NgRx Signal Store, Change Detection, Tailwind CSS, Smart/Dumb компоненти, standalone-компоненти, ліниве завантаження.*

Короткий зміст праці:

У кваліфікаційній роботі розроблено клієнтську частину для маркетплейсу вживаних автомобілів «Kolesko». Робота охоплює повний цикл створення сучасного фронтенд-застосунку з акцентом на продуктивності, архітектурній гнучкості та позитивному користувацькому досвіді. На початковому етапі проведено аналіз предметної області та конкурентних рішень. Для реалізації обрано фреймворк Angular, що дозволило створити масштабовану архітектуру із застосуванням модульності, лінивого завантаження, стратегії Change Detection OnPush та Signal API. Спроєктовано та реалізовано інтуїтивно зрозумілий, візуально привабливий та адаптивний дизайн лендінгу. Для ефективного управління даними, особливо в складних розділах, як каталог, інтегровано NgRx Signal Store. Застосовано архітектурні підходи, такі як Smart/Dumb компоненти та standalone-компоненти для покращення читабельності та повторного використання коду. Розробка велася з використанням системи контролю версій Git та платформи GitHub. Результатом є технологічно обґрунтоване рішення, що відповідає стандартам фронтенд-розробки та готове до подальшого розвитку.

_____

ANNOTATION
of qualification paper
for bachelor's degree

Theme: *Development of the Client-Side of a Used Cars Marketplace*

Author: *Serhii Vitaliiovych Kukharchuk*

Scientific supervisor: *Lecturer-Intern of the Department of ITAD, Matsevych Denys Volodymyrovych*

Defended «……» …… of 20__ року.

Explanatory note to the qualification work: *64 pages, 22 figures, 2 tables, 3 appendices, 18 references.*

Keywords: *used cars marketplace, client-side, frontend development, Angular, web technologies, user interface, landing page design, modular architecture, Signal API, NgRx Signal Store, Change Detection, Tailwind CSS, Smart/Dumb components, standalone components, lazy loading.*

Summary of the paper:
This qualification work presents the development of the client-side for the used cars marketplace "Kolesko". The project covers the full cycle of creating a modern frontend application with a focus on performance, architectural flexibility, and positive user experience. The initial stage included an analysis of the domain and competing solutions. The Angular framework was chosen for implementation, enabling the creation of a scalable architecture using modularity, lazy loading, the OnPush Change Detection strategy, and the Signal API. An intuitive, visually appealing, and responsive landing page design was designed and implemented. For efficient data management—especially in complex sections such as the catalog—the NgRx Signal Store was integrated. Architectural approaches such as Smart/Dumb components and standalone components were applied to improve code readability and reusability. The development process utilized the Git version control system and the GitHub platform. The result is a technologically grounded solution that meets frontend development standards and is ready for further development.

_____

ЗМІСТ

ВСТУП

В епоху стрімкої цифровізації та зростання онлайн-торгівлі, автомобільний ринок зазнає значних трансформацій, де електронні платформи стають ключовим інструментом для купівлі-продажу вживаних автомобілів. Ці маркетплейси еволюціонують у складні інтерактивні системи, де якісний, функціональний та естетично привабливий клієнтський інтерфейс є критично важливим для забезпечення позитивного досвіду користувача та успіху платформи загалом. Сучасний інтерфейс має не тільки забезпечувати виконання основних функцій, але й бути інтуїтивно зрозумілим, зручним у навігації та візуально привабливим, що сприяє ефективній взаємодії користувачів з контентом.

Актуальність теми дослідження обумовлена постійно зростаючим попитом на вживані автомобілі та необхідністю створення конкурентоспроможних онлайн-платформ, що відповідають сучасним очікуванням користувачів щодо швидкодії та зручності. В умовах високої конкуренції на ринку маркетплейсів, ключовим фактором успіху стає не лише функціональність, але й продумана реалізація клієнтської частини, що забезпечує легку навігацію, швидке завантаження та відображення інформації, а також приємний візуальний досвід. Професійно розроблена клієнтська частина, зокрема з використанням сучасних фреймворків та бібліотек, дозволяє підвищити рівень довіри до платформи, збільшити час перебування користувачів на сайті та сприяти зростанню конверсії.

Мета дослідження – розробка високоефективної та зручної у використанні клієнтської частини маркетплейсу вживаних автомобілів з використанням сучасних веб-технологій.

Задачі дослідження:

- Провести аналіз предметної області, включаючи дослідження ринку вживаних автомобілів, визначення потреб цільової аудиторії та вивчення успішних практик реалізації клієнтських інтерфейсів маркетплейсів.

- Спроектувати модульну архітектуру клієнтської частини вебдодатку, що забезпечить масштабованість, повторне використання компонентів та зручність підтримки.
- Розробити та реалізувати клієнтську частину вебплатформи, включаючи користувацький інтерфейс, бізнес-логіку, взаємодію з API та оптимізацію продуктивності, використовуючи фреймворк Angular, а також забезпечити адаптивну верстку та стилізацію за допомогою бібліотеки Tailwind CSS.
- Інтегрувати додаткові бібліотеки та інструменти для розширення функціональних можливостей та оптимізації продуктивності клієнтської частини, а також реалізувати сучасні підходи до управління станом та оптимізації рендерингу.
- Організувати процес розробки та управління кодом за допомогою системи контролю версій Git та сервісу GitHub.

Об'єкт дослідження – клієнтська частина вебплатформи маркетплейсу вживаних автомобілів.

Предмет дослідження – сучасні інструментальні засоби, методи та технології розробки клієнтської частини вебдодатків, включаючи компонентний підхід, управління станом та оптимізацію продуктивності.

Методи дослідження включають системний аналіз предметної області та існуючих рішень, порівняльний аналіз технологій та інструментів розробки, проектування та реалізація архітектури клієнтської частини із застосуванням об'єктно-орієнтованого та реактивного підходів програмування, юзабіліті-тестування, а також тестування та налагодження програмного забезпечення.

Розробка якісної клієнтської частини маркетплейсу з використанням обраних технологій та підходів дозволить створити платформу, що відрізняється високим рівнем швидкодії, зручності використання та ефективності взаємодії користувачів із системою, що, в свою чергу, сприятиме підвищенню конкурентоспроможності та потенційного успіху проєкту на ринку онлайн-платформ з продажу вживаних автомобілів.

РОЗДІЛ 1.

ТЕОРЕТИЧНА ЧАСТИНА

1.1. Аналіз платформ-конкурентів для продажу вживаних автомобілів

На українському ринку онлайн-продажу автомобілів існує декілька потужних платформ, кожна з яких має свої унікальні характеристики та переваги.

---

1. AUTO.RIA (Фото сайту зображено на рис. 1.1).

Рис. 1.1. Основна сторінка сайту з доменом auto.ria.com

Джерело: https://auto.ria.com/uk/

Характеристика платформи:

AUTO.RIA є найбільшим і найпопулярнішим сайтом для продажу автомобілів

в Україні, що вирізняється комплексним підходом до онлайн-транзакцій.

Особливості інтерфейсу та дизайну:

- Сучасний, інтуїтивно зрозумілий дизайн.
- Багатофункціональна система фільтрів.
- Яскраве та деталізоване візуальне представлення автомобілів.

Ключові функціональні можливості:

- Розміщення детальних оголошень.
- Перевірка історії автомобіля.
- Онлайн-страхування.
- Розширений функціонал для пошуку та порівняння авто.

---

1. RST.ua (Фото сайту зображено на рис. 1.2).

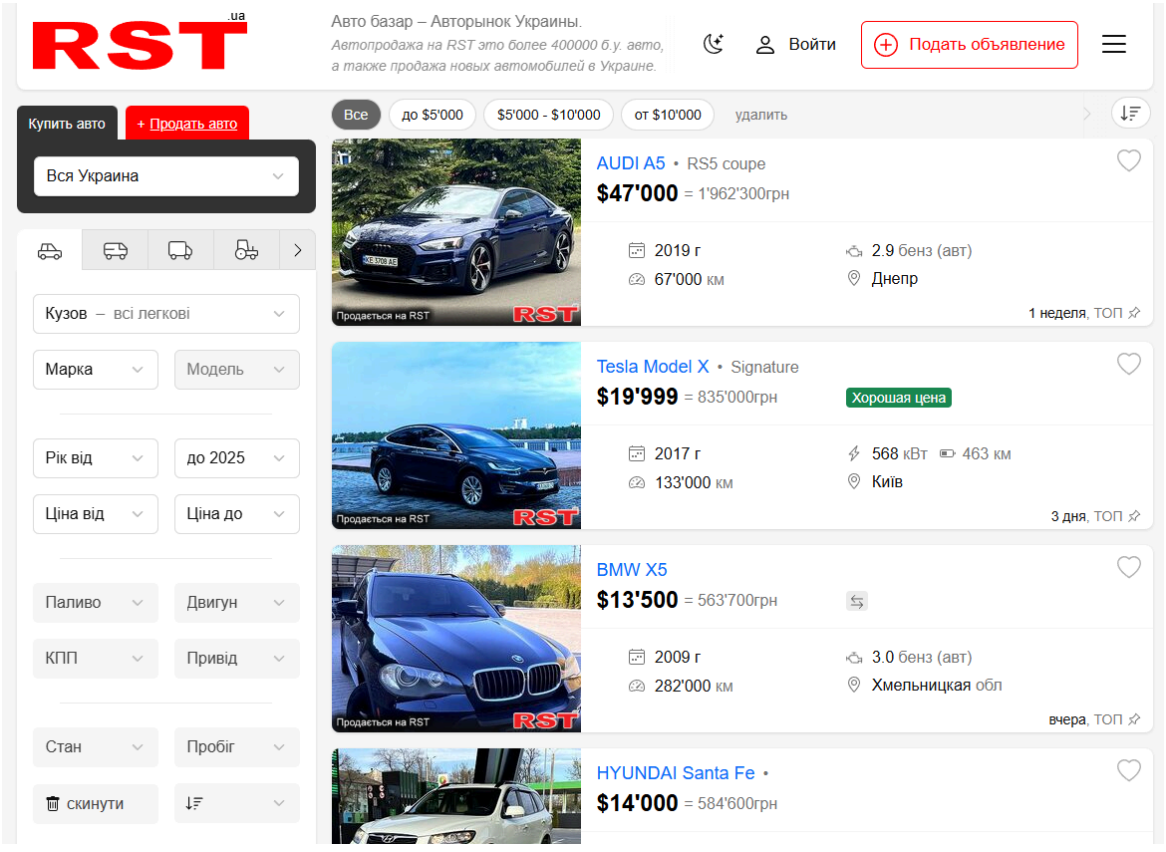


Рис. 1.2. Основна сторінка сайту з доменом rst.ua

Джерело: https://rst.ua/ukr/

Характеристика платформи:

RST.ua — популярний майданчик для продажу автомобілів, орієнтований

на простоту та зручність використання.

Особливості інтерфейсу та дизайну:

- Мінімалістичний дизайн.
- Лаконічний інтерфейс.
- Фокус на функціональності та швидкості пошуку.

Ключові функціональні можливості:

- Базові інструменти для продажу та купівлі авто.
- Гнучкий пошук за параметрами.
- Система контактів із продавцями.

---

3. OLX.ua (Фото сайту зображено на рис. 1.3).

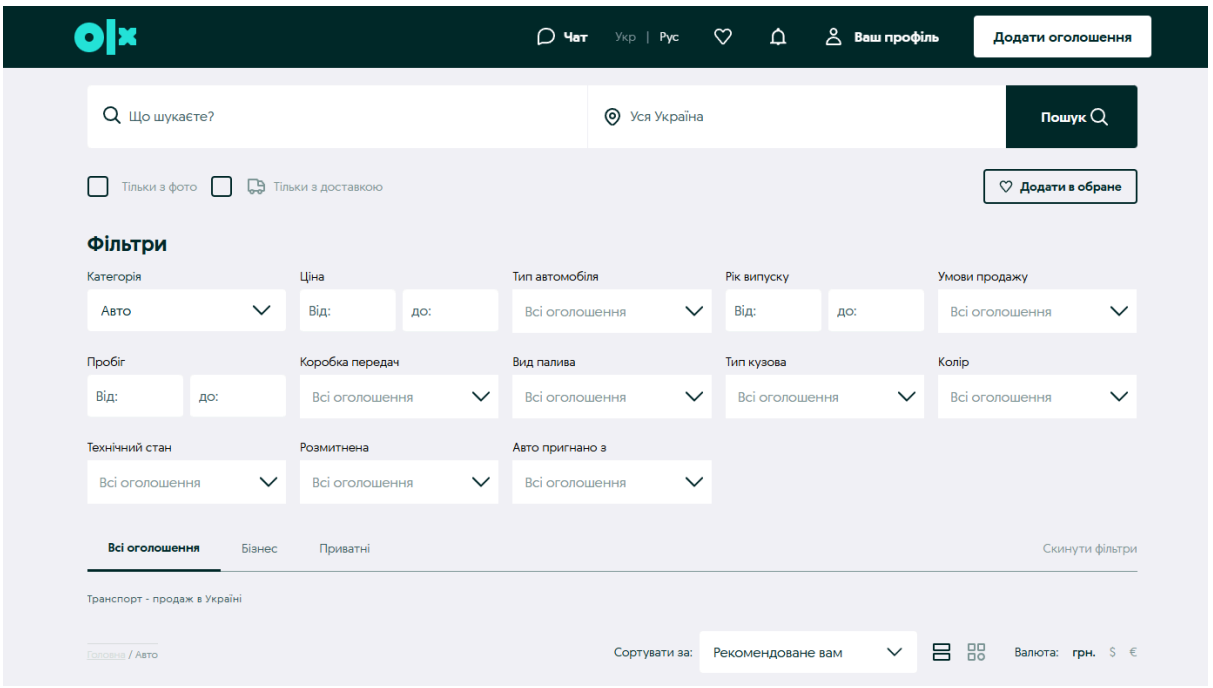


Рис. 1.3. Сторінка з основними фільтрами сайту з доменом olx.ua

Джерело:

https://www.olx.ua/uk/transport/legkovye-avtomobili/q-car/?srsltid=AfmBOoqIpyhxJQGSOOM7a8UPgo wjCeHqN1YRvqZSGd_HIys8WwaBW2WN

Характеристика платформи:

OLX.ua — універсальна платформа для продажу різних товарів, яка включає окремий розділ для автомобілів.

Особливості інтерфейсу та дизайну:

- Простий та зрозумілий дизайн.
- Гнучка система налаштування фільтрів.
- Орієнтація на зручність для користувачів.

Ключові функціональні можливості:

- Базовий функціонал продажу автомобілів.
- Внутрішній чат для комунікації.
- Можливість просування оголошень.

Загальні тенденції ринку онлайн-продажу автомобілів

Аналіз платформ дозволяє виділити кілька ключових трендів, які формують сучасний ринок:

1. **Технологічність:**

   Використання сучасних веб-технологій для забезпечення швидкої, стабільної роботи платформ.

2. **Розширені пошукові можливості:**
   - Інтуїтивна навігація.
   - Багатокритеріальні фільтри для точного підбору автомобіля.
3. **Візуальна презентація:**
   - Якісні фотографії.
   - Відеоогляди автомобілів.
   - Детальні галереї зображень.
4. **Додаткові сервіси:**
   - Перевірка історії автомобіля.
   - Онлайн-страхування.
   - Фінансові інструменти.
   - Консультаційна підтримка.

Ці платформи демонструють розвиток онлайн-ринку автомобілів в Україні, пропонуючи покупцям і продавцям дедалі зручніші та функціональніші інструменти.

1.2. Вибір технологій для реалізації клієнтської частини

Розробка сучасних клієнтських вебзастосунків, особливо таких як багатофункціональні онлайн-маркетплейси, вимагає використання ефективних інструментів, що забезпечують високу швидкість розробки, зручність підтримки та масштабованість. Серед різноманіття JavaScript/TypeScript фреймворків та бібліотек для побудови користувацьких інтерфейсів, найбільш популярними та затребуваними сьогодні є React, Vue.js, Svelte та Angular. Кожен з цих інструментів має свої архітектурні особливості, переваги та недоліки, що робить процес вибору оптимального рішення для конкретного проєкту важливим етапом розробки.

- **React:** JavaScript бібліотека для побудови користувацьких інтерфейсів, що використовує концепцію віртуального DOM. Вирізняється високою гнучкістю та великою екосистемою сторонніх рішень, проте, будучи лише бібліотекою для UI, вимагає інтеграції додаткових інструментів для повноцінної розробки SPA.
- **Vue.js:** Прогресивний фреймворк, відомий своєю простотою вивчення та зручністю використання. Надає офіційні companion-бібліотеки для ключових завдань SPA, добре підходить для проєктів різного масштабу, хоча при реалізації дуже великих систем можуть виникати питання, пов'язані зі структурою.
- **Svelte:** Інноваційний підхід до розробки інтерфейсів, що компілює код компонентів у високоефективний нативний JavaScript, забезпечуючи відмінну продуктивність без необхідності віртуального DOM. Головним обмеженням є відносно молода та менша екосистема порівняно з іншими лідерами ринку.
- **Angular:** Комплексний ("повноцінний") фреймворк, що пропонує інтегровані рішення для більшості аспектів розробки SPA (маршрутизація, HTTP, управління станом). Відрізняється чіткою, думко-орієнтованою структурою, що сприяє масштабованості та зручності підтримки великих проєктів, хоча має вищий поріг входження.

Порівняльний аналіз та обґрунтування вибору Angular:

При виборі фреймворку для розробки клієнтської частини маркетплейсу вживаних автомобілів враховувалися такі ключові критерії:

- **Масштабованість:** Маркетплейс є складною системою з великою кількістю функціональних модулів та даних. Angular, завдяки своїй модульній архітектурі (або архітектурі на основі Standalone компонентів) та суворій структурі, надає інструменти, необхідні для ефективного управління складністю та масштабування проєкту у майбутньому. На відміну від більш гнучких бібліотек, які вимагають від команди самостійного узгодження архітектурних рішень, Angular пропонує готові патерни, що спрощує онбординг нових розробників та підтримку коду в довгостроковій перспективі.
- **Екосистема та інтегровані рішення:** Angular надає "з коробки" більшість інструментів, необхідних для розробки типового SPA (маршрутизація, HTTP-клієнт, управління формами). Хоча React та Vue.js мають великі екосистеми сторонніх бібліотек, інтеграція та забезпечення їх сумісності вимагає додаткових зусиль. Інтегровані рішення Angular спрощують процес розробки та знижують ризик конфліктів між бібліотеками.
- **Надійність та підтримка:** Розробка під керівництвом Google забезпечує довгострокову підтримку фреймворку, регулярні оновлення та передбачуваний розвиток. Це є важливим фактором для великих проєктів, які плануються до тривалої експлуатації.
- **Продуктивність та оптимізація:** Хоча ранні версії Angular могли мати проблеми з продуктивністю порівняно з React чи Vue.js, сучасний Angular активно розвивається у цьому напрямку. Використання TypeScript, AOT-компіляції, а головне – гнучких стратегій Change Detection (таких як OnPush) та інтеграція з Signal API дозволяють досягти високої продуктивності навіть у складних додатках. Це підтверджується постійними зусиллями команди Angular щодо оптимізації механізмів рендерингу, що було досліджено і застосовано в даній роботі.
- **Типізація:** Використання TypeScript є значною перевагою для великих проєктів. Статична типізація допомагає виявляти помилки на ранніх етапах, покращує якість коду та полегшує роботу в команді.

Незважаючи на відносно крутішу криву навчання порівняно з Vue.js або Svelte, переваги Angular у масштабованості, наявності інтегрованих рішень, підтримці TypeScript та потенціалі для оптимізації роблять його оптимальним вибором для розробки клієнтської частини маркетплейсу вживаних автомобілів, де надійність, зручність підтримки та висока продуктивність є пріоритетними.

1.3. Принципи управління станом у клієнтських вебдодатках.

Зі зростанням складності сучасних односторінкових застосунків (SPA), особливо таких, що працюють із значним обсягом взаємопов'язаних даних та вимагають динамічного оновлення інтерфейсу, ефективне управління станом додатку стає критично важливим завданням. Стан застосунку включає в себе всі дані, які відображаються користувачеві, а також інформацію про його дії, налаштування та поточний контекст.

1.3.1. Необхідність управління станом:

У невеликих застосунках передача даних між компонентами може здійснюватися через вхідні (@Input()) та вихідні (@Output()) параметри [1], а для асинхронних операцій можуть використовуватися сервіси з суб'єктами RxJS (Subject, BehaviorSubject) [2]. Однак, у міру зростання проєкту, такий підхід призводить до виникнення складної та заплутаної мережі залежностей між компонентами (так званий "props drilling" або "callback hell"), що ускладнює:

- **Відстеження змін:** Важко зрозуміти, які компоненти отримують дані, звідки вони надходять та як зміни в одному місці впливають на інші частини додатку.
- **Масштабованість:** Додавання нового функціоналу або зміна існуючого вимагає модифікації багатьох компонентів та зв'язків між ними.
- **Тестування:** Ізольоване тестування компонентів стає складнішим через їхню тісну пов'язаність зі станом, який розподілений по всьому додатку.
- **Підтримка коду:** Збільшується ймовірність появи помилок, рефакторинг стає ризикованим.

Для вирішення цих проблем застосовуються патерни та бібліотеки для централізованого управління станом (State Management). Основна ідея полягає у створенні єдиного джерела істини (Single Source of Truth) для стану додатку, що знаходиться в одному місці (Store). Компоненти можуть читати дані зі Store та відправляти команди (дії) для зміни стану, але пряма модифікація стану компонентами зазвичай заборонена. Це забезпечує передбачуваність змін стану та спрощує налагодження.

1.3.2. Підходи до управління станом в Angular:

В екосистемі Angular існує декілька популярних рішень для управління станом, які базуються на різних принципах та патернах.

Класичний NgRx Store: Це одне з найбільш зрілих та потужних рішень, що базується на патерні Redux. NgRx Store [3] пропонує сувору односпрямовану архітектуру потоку даних, що вимагає визначення:

- **Actions:** об'єкти, що описують події, які відбулися.
- **Reducers:** чисті функції, що визначають, як змінюється стан у відповідь на дії.
- **Selectors:** функції для отримання даних зі стану.
- **Effects:** механізм для управління побічними ефектами (наприклад, HTTP-запитами).

Незважаючи на свою потужність та передбачуваність, NgRx Store може вимагати значного обсягу шаблонного коду (boilerplate), що ускладнює його вивчення та підтримку для невеликих або середніх проєктів, а також додає рівні абстракції.

NGXS: Ця бібліотека [4] позиціонується як більш проста та менш багатослівна альтернатива NgRx Store. Вона використовує класи та декоратори для визначення стану та дій, що зменшує обсяг boilerplate-коду порівняно з класичним NgRx. Однак, NGXS все ще значною мірою покладається на Observable з бібліотеки RxJS та імперативне використання методу **dispatch()** для відправки дій.

NgRx Signal State (та Signal API): З появою в Angular нового Signal API, що забезпечує так звану дрібнозернисту реактивність (fine-grained reactivity), команда

NgRx розробила нове рішення – NgRx Signal State [5] (також відоме як NgRx Signal Store). Цей підхід повністю інтегрований з нативним для Angular Signal API та дозволяє управляти станом додатку з мінімальним обсягом коду та без необхідності використання RxJS Observable для базових операцій зі станом.

NgRx Signal State поєднує переваги централізованого Store з простотою використання та високою продуктивністю, яку надають Signals. Він дозволяє:

- Визначати стан за допомогою Signals [6].
- Оновлювати стан безпосередньо через Signals або спеціальні методи.
- Реагувати на зміни стану за допомогою обчислюваних Signals або ефектів (при необхідності).
- Значно зменшити шаблонний код порівняно з класичними рішеннями.

Цей підхід ідеально узгоджується з розвитком Angular у напрямку Signal-based архітектури та добре інтегрується з оптимізаціями рендерингу, такими як ChangeDetection Strategy.OnPush [7], дозволяючи оновлювати лише ті частини інтерфейсу, які дійсно змінилися.

1.4. Механізми оновлення користувацького інтерфейсу у фронтенд-фреймворках.

Одним із фундаментальних завдань будь-якого фронтенд-фреймворку є ефективне та швидке оновлення користувацького інтерфейсу (UI) у відповідь на зміни в стані додатка. З розвитком SPA, де значна частина логіки та даних обробляється на стороні клієнта, механізми синхронізації даних та візуального представлення стають все більш складними. Неефективне оновлення UI може призводити до зниження продуктивності, "зависань" інтерфейсу та негативно впливати на користувацький досвід.

1.4.1. Суть механізмів оновлення інтерфейсу (Change Detection):

В основі динамічних фронтенд-додатків лежить процес, відомий як "виявлення змін" (Change Detection) [7]. Це механізм, який відстежує зміни в даних додатка і визначає, які частини інтерфейсу потребують оновлення для відображення

цих змін. Різні фреймворки реалізують цей процес по-різному, але загальна мета полягає в тому, щоб мінімізувати прямі маніпуляції з DOM (Document Object Model), які є відносно повільними, та оновлювати лише необхідні елементи.

1.4.2. Механізм виявлення змін в Angular:

Angular використовує власний механізм виявлення змін, який за замовчуванням покладається на бібліотеку Zone.js [8]. Zone.js перехоплює асинхронні операції браузера (такі як HTTP-запити, події DOM, таймери) і повідомляє Angular про потенційні зміни даних. Після кожної такої операції Angular запускає цикл виявлення змін, який за стандартною стратегією (ChangeDetectionStrategy.Default) перевіряє все дерево компонентів зверху вниз на наявність змін. Якщо зміни виявлено, відповідні частини DOM оновлюються.

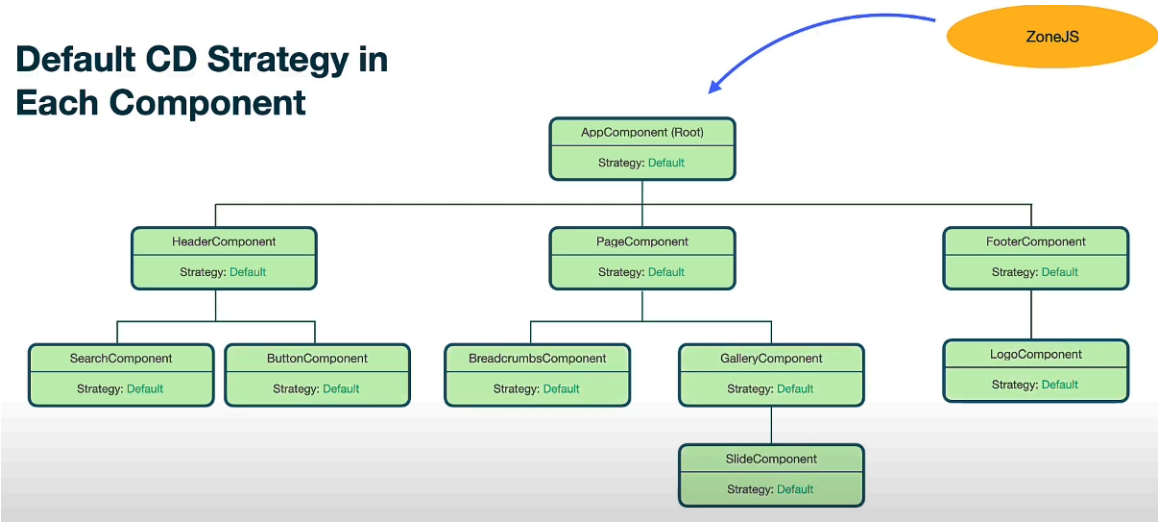


Рис. 1.4. Приклад перевірки компонентного дерева при Default налаштуванні

Джерело:

https://www.youtube.com/watch?v=WAu7omIoerM&t=825s&ab_channel=DecodedFrontend

Хоча цей механізм є надійним, перевірка всього дерева компонентів при кожній потенційній зміні може призводити до надлишкових обчислень та зниження продуктивності у великих додатках зі складною структурою компонентів та частими змінами даних.

1.4.3. Оптимізація виявлення змін: Стратегія OnPush:

Для підвищення продуктивності Angular надає альтернативну стратегію виявлення змін – ChangeDetectionStrategy.OnPush [7]. При використанні цієї стратегії Angular не перевіряє компонент на зміни при кожній асинхронній події, ініційованій Zone.js. Натомість, компонент буде перевірено та оновлено лише у таких випадках:

- Змінилися вхідні параметри (@Input()) компонента (за умовою, що вони є немутабельними об'єктами).
- Відбулася подія, ініційована самим компонентом або його дочірніми компонентами (наприклад, через EventEmitter).
- Явно викликано метод для позначення компонента як такого, що потребує перевірки (markForCheck()).
- Змінилася Observable, на яку підписано в шаблоні за допомогою async пайпа.
- Змінився Signal, прив'язаний до шаблону.

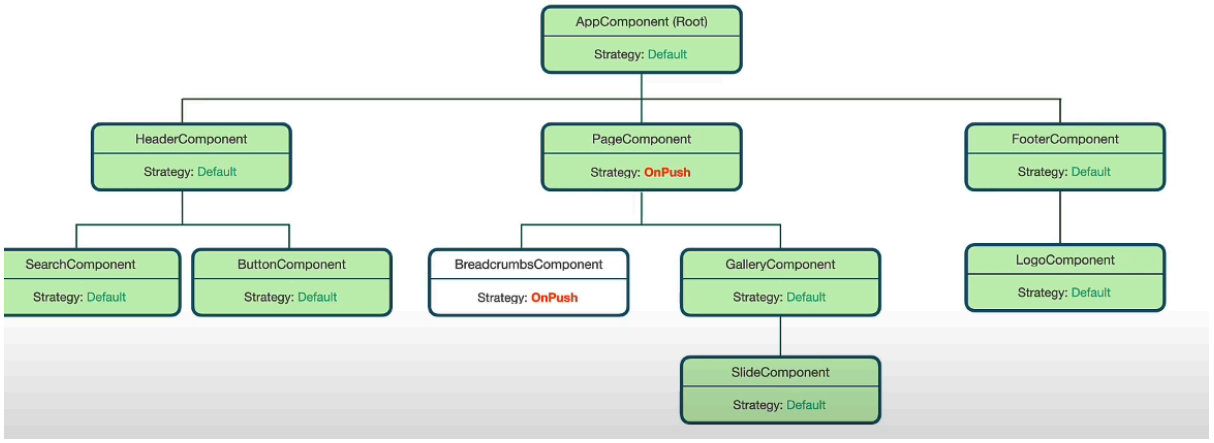


Рис. 1.5. Приклад перевірки компонентного дерева при вибірковому OnPush налаштуванні

Джерело:

https://www.youtube.com/watch?v=WAu7omIoerM&t=825s&ab_channel=DecodedFrontend

Використання OnPush стратегії дозволяє значно зменшити навантаження на систему виявлення змін, оскільки Angular пропускає перевірку піддерев компонентів, які не отримали нових даних. Це особливо ефективно у додатках з

великою кількістю "тупих" (Dumb) компонентів, які отримують всі дані через @Input().

1.4.4. Роль Signal API у реактивності та оновленні інтерфейсу:

Новітній Signal API [9] в Angular є суттєвим кроком у напрямку більш ефективного та дрібнозернистого управління реактивністю. Signals – це функції, які містять значення та відстежують залежності. Коли значення Signal змінюється, Angular автоматично знає, які обчислювані значення (computed Signals) та ефекти (effects), а головне – які частини інтерфейсу залежать від цього Signal, і потребують оновлення.

Інтеграція Signal API з механізмом виявлення змін, особливо у поєднанні зі стратегією OnPush, дозволяє досягти ще більш високого рівня оптимізації рендерингу. Замість перевірки всього компонента або його частини, Angular може оновити лише ті DOM-елементи, які безпосередньо пов'язані зі зміненим Signal. Це забезпечує "fine-grained reactivity", де оновлюється мінімально необхідний фрагмент інтерфейсу. Signal API спрощує роботу з реактивними даними порівняно з використанням RxJS [1] для простих сценаріїв, зменшуючи boilerplate-код та когнітивне навантаження на розробника. Цей підхід є частиною стратегії Angular щодо переходу до майбутнього без Zone.js [8] .

1.4.5. Інструменти для аналізу та оптимізації:

Для розуміння та оптимізації роботи механізмів виявлення змін, розробники Angular можуть використовувати спеціалізовані інструменти, такі як Angular DevTools. Ці інструменти дозволяють візуалізувати процес виявлення змін, аналізувати, які компоненти перевіряються та оновлюються, і виявляти потенційні "вузькі місця" (bottlenecks) або надлишкові перевірки, спричинені, наприклад, неправильним використанням стратегій Change Detection або особливостями роботи Zone.js.

Таким чином, ефективне управління оновленнями користувацького інтерфейсу в Angular досягається за рахунок гнучкого механізму виявлення змін,

оптимізаційних стратегій (OnPush) та сучасного Signal API, які разом дозволяють мінімізувати ресурсозатрати на рендеринг та забезпечити високу продуктивність додатків, що є критично важливим для багатофункціональних платформ, як-от онлайн-маркетплейс.

РОЗДІЛ 2.

ПРАКТИЧНА ЧАСТИНА РОЗРОБКИ САЙТУ - МАРКЕТПЛЕЙС ВЖИВАНИХ АВТОМОБІЛІВ

2.1. Розробка дизайну лендінгу (landing page) для сайту

Розробка дизайну лендінгу для сайту «Kolesko» ґрунтувалася на принципах зручності та візуальної привабливості, враховуючи необхідність швидкого та ефективного залучення користувачів. Важливим етапом було використання вже розробленої кольорової палітри, що забезпечує цілісність та гармонійність візуального стилю.

2.1.1. Проєктування хедера та Hero-секції

При розробці хедера головною метою було мінімальне наповнення при максимальній інформативності та функціональності. Всі елементи розміщувалися таким чином, щоб забезпечити легку навігацію та швидкий доступ до ключових функцій.

У верхній частині сторінки розміщено хедер з логотипом «Kolesko», який є основним елементом фірмової ідентичності та навігації. Поруч з логотипом інтегровано пошуковий рядок (search bar), що дозволяє користувачам швидко знаходити потрібні автомобілі без необхідності використання розширених фільтрів.

Праворуч у хедері розташовані дві важливі іконки: іконка профілю, що надає доступ до особистого кабінету користувача, та іконка обраних оголошень, що дозволяє швидко переглядати збережені пропозиції. Також у хедері розміщена кнопка «Продати машину», яка заохочує нових продавців додавати свої оголошення на платформу, тим самим розширюючи асортимент маркетплейсу.

Дизайн хедера враховує адаптивність для різних пристроїв. На мобільних версіях хедер може бути спрощений, наприклад, з використанням меню-бургера для приховування менш пріоритетних елементів та забезпечення оптимального використання екранного простору.

Далі розташовується Hero-секція – перший екран, який бачить користувач при відвідуванні сайту. Ця секція має привертати увагу та передавати основну цінність маркетплейсу. Вона включає зображення двох автомобілів, що асоціюються з цільовою тематикою, а також короткий, але змістовний заголовок та підзаголовок. Нижче під підзаголовком розташовані чотири іконки, що візуалізують кроки процесу купівлі-продажу: "Обери машину", "Зв'яжись з продавцем", "Назнач місце зустрічі" та "Ти платиш на місці". Ці іконки супроводжуються відповідними текстовими поясненнями, що допомагають користувачеві швидко зрозуміти етапи угоди.

Використання кольорової палітри, розробленої партнером, дозволило забезпечити візуальну єдність та гармонію між хедером, Hero-секцією та іншими елементами сторінки. Контрастні акцентний червоний колір застосовується для кнопок дії, такої як «Продати машину», щоб зробити її більш помітною та заохотити до натискання.

Нижче наведено приклад дизайну хедера та Hero-секції для десктопної та мобільної версій (див. Рис. 2.1 та див. Рис. 2.2).

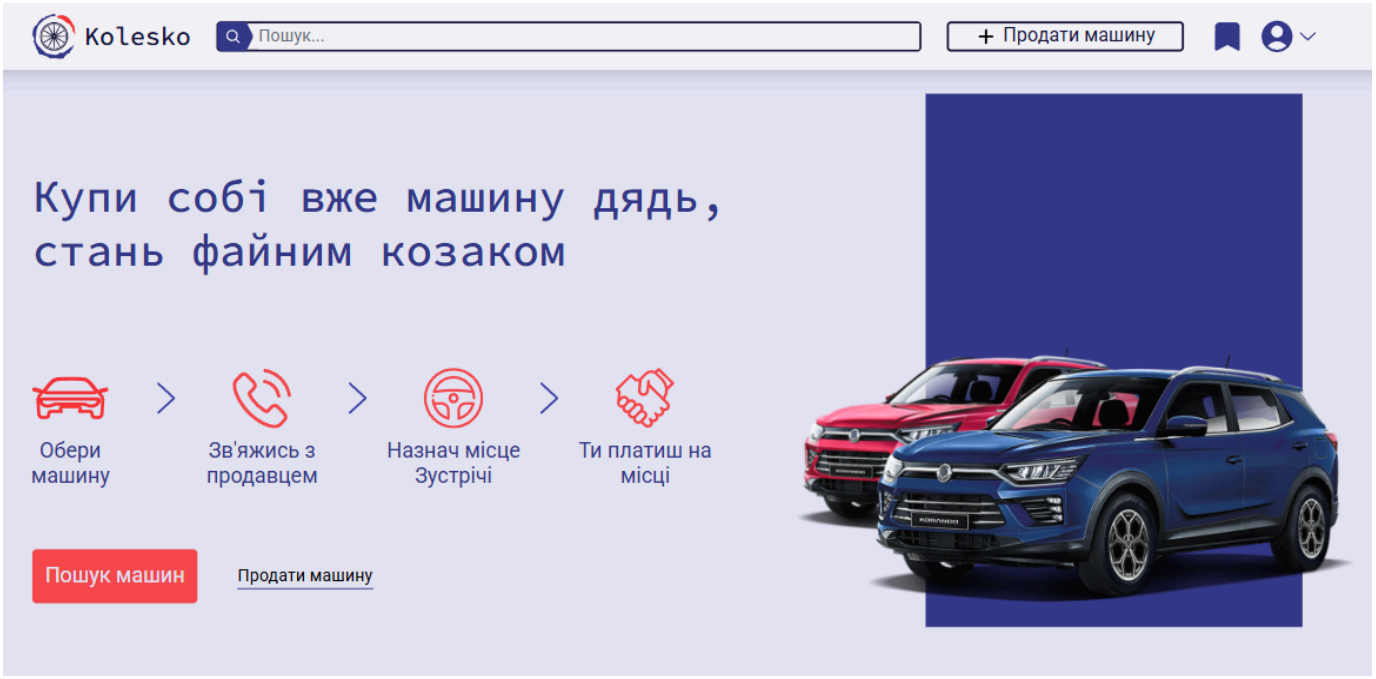


Рис. 2.1. Дизайн хедера та Hero-секції для десктопної версії

Джерело: Розроблено автором

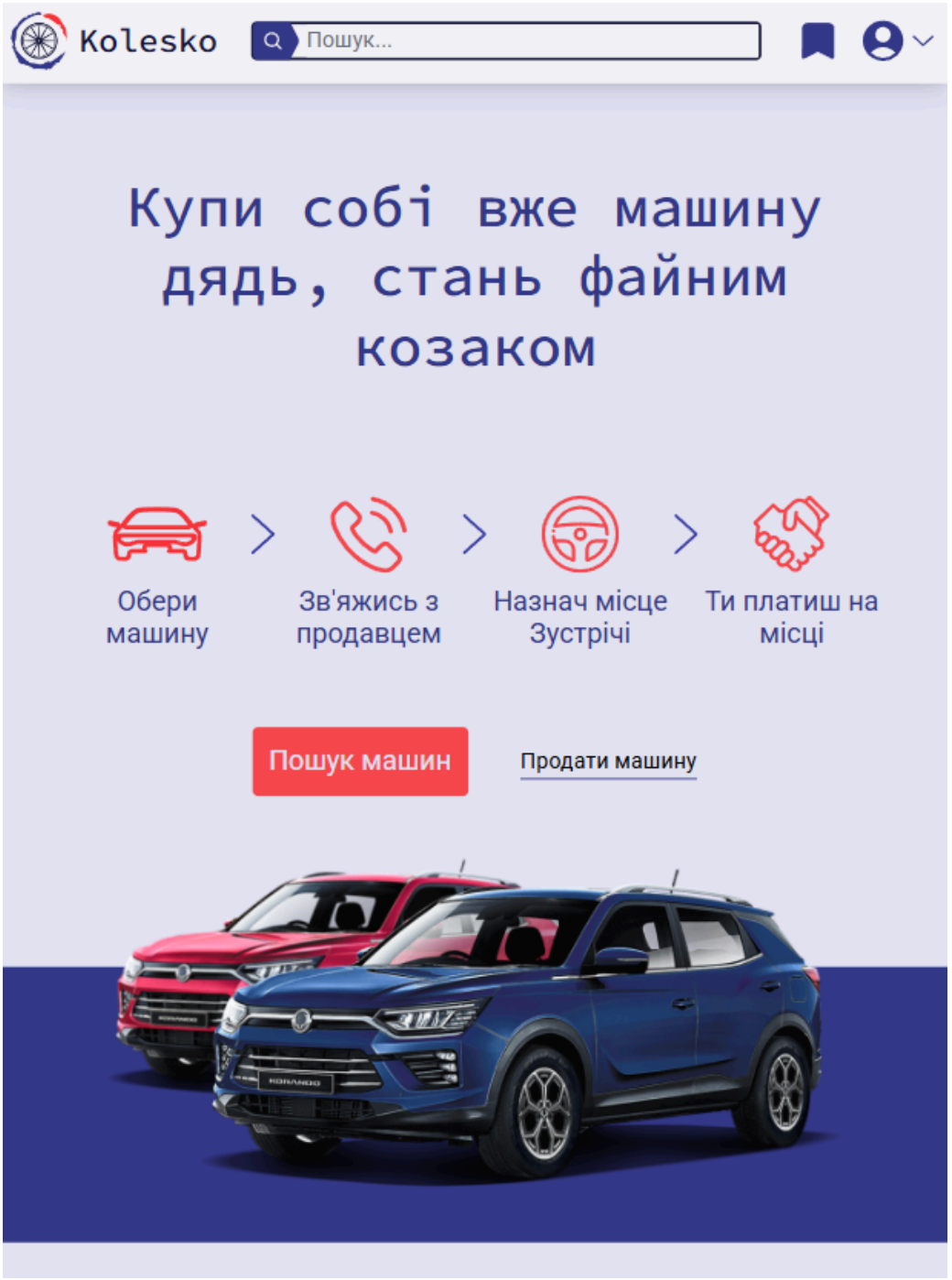


Рис. 2.2. Дизайн хедера та Hero-секції для мобільної версії

Джерело: Розроблено автором

2.1.2. Проєктування секцій "Обери свій тип" та "ТОП пропозиції"

Після Hero-секції на лендінг-сторінці розташовуються дві ключові секції: "Обери свій тип" та "ТОП пропозиції". Їхня мета – зацікавити користувача, представивши найактуальніші та найпривабливіші пропозиції, спонукаючи до подальшого дослідження каталогу та збільшення монетизації платформи.

Секція "Обери свій тип" призначена для швидкого доступу до оголошень за типом кузова автомобіля. Вона має на меті спростити пошук для користувачів, які вже визначилися з бажаним типом транспортного засобу. Візуально ця секція представлена у вигляді стрічки-сітки з інтерактивними елементами, кожен з яких візуалізує певний тип кузова (наприклад, седан, кросовер, пікап) та супроводжується зображенням, що асоціюється з цим типом. При натисканні на елемент користувач перенаправляється на сторінку каталогу з відфільтрованими оголошеннями саме для цього типу автомобіля. Такий підхід значно скорочує шлях користувача до бажаного контенту, підвищуючи ефективність взаємодії з платформою.

Секція "ТОП пропозиції" є динамічним блоком, який відображає найбільш привабливі та пріоритетні оголошення, виставлені згідно з певними параметрами, визначеними розробником сайту. Ці параметри можуть включати, наприклад, автомобілі, що мають високий попит, пропонуються за акційною ціною, або є результатом платного просування з боку продавця. Основна мета цього блоку – не лише зацікавити користувача, але й створити потенціал для монетизації платформи за рахунок платних розміщень. Кожна пропозиція представлена у вигляді компактної картки, що містить якісне зображення автомобіля, його марку та модель, а також ціну. Це дозволяє швидко оцінити привабливість оголошення без переходу на окрему сторінку деталей.

Обидві секції спроєктовані з урахуванням адаптивності для різних пристроїв. На десктопній версії вони можуть бути розташовані поруч або одна за одною, використовуючи горизонтальний скролінг або сіткове відображення для оптимального використання простору. На мобільній версії, щоб уникнути перевантаження екрана, ці секції розташовуються вертикально, одна під одною. Картки в секції "ТОП пропозиції" адаптують свій розмір та розташування, щоб забезпечити зручне відображення на менших екранах, зберігаючи при цьому читабельність інформації. Елементи в секції "Обери свій тип" також динамічно перебудовуються, зменшуючи кількість елементів у ряду або збільшуючи їх розмір для зручності взаємодії на сенсорних екранах. Використання мінімалістичного

дизайну та вже визначеної кольорової палітри забезпечує візуальну консистентність цих секцій з рештою лендінг-сторінки.

Нижче наведено приклади дизайну секцій "Обери свій тип" та "ТОП пропозиції" (див. Рис. 2.3, див. Рис. 2.4).

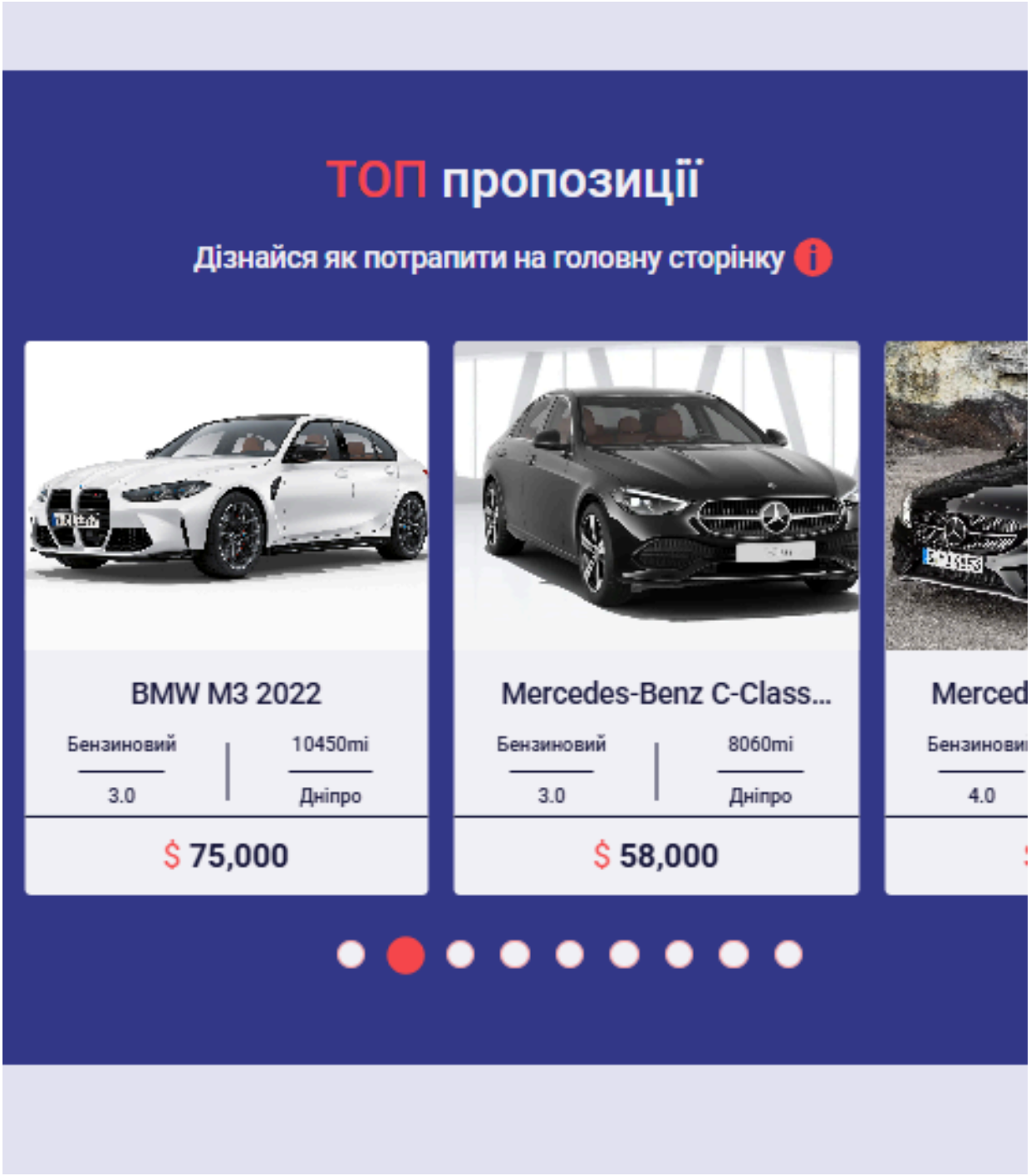


Рис. 2.3. Фото секції “Топ пропозиції” (мобільна версія)

Джерело: Розроблено автором

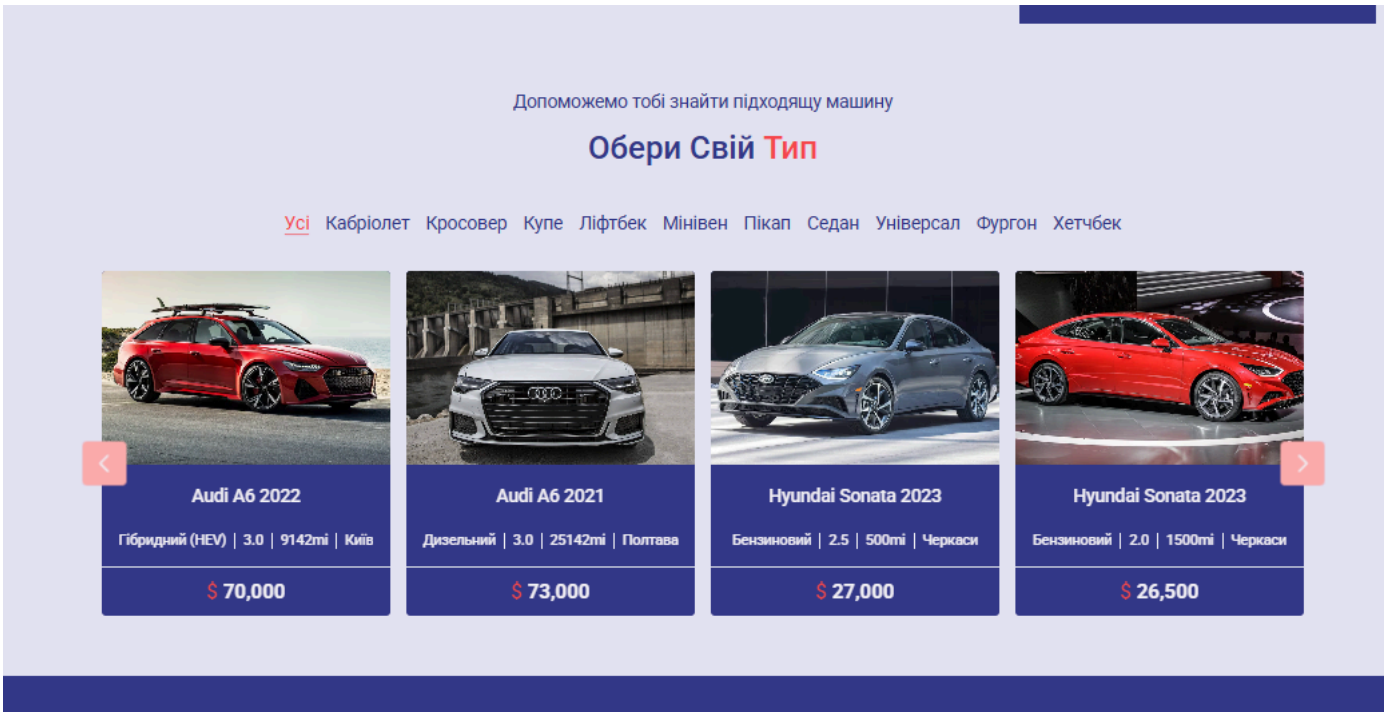


Рис. 2.4. Фото секції “Обери свій тип” (десктопна версія)

Джерело: Розроблено автором

2.1.3. Проєктування візуально-орієнтованої секції для залучення

Ця секція розроблена як ефективний візуальний елемент, що має на меті знову привернути увагу користувача або продавця, спонукаючи його до подальшої взаємодії з платформою. Її основне завдання – не стільки надавати нову інформацію, скільки естетично та інтуїтивно перенаправляти користувачів до ключових розділів сайту або до процесу додавання оголошення.

Візуально секція складається з великого, привабливого зображення автомобіля, що асоціюється з якістю, сучасністю та бажанням володіти авто. Над цим зображенням або поруч з ним (в залежності від пристрою) розміщується короткий, але спонукальний текст, який може бути звернений як до потенційних покупців ("Знайди свій ідеальний автомобіль"), так і до продавців ("Продай своє авто легко та швидко").

Основна ідея цієї секції полягає у створенні емоційного зв'язку з користувачем через якісний візуал та простий, зрозумілий меседж. Вона виступає як "міст" між попередніми блоками з пропозиціями та наступними функціональними елементами

сайту.

Щодо адаптивності, на мобільних пристроях ця секція зберігає свою візуальну привабливість, але її елементи адаптуються до меншого розміру екрана. Зображення може бути масштабоване або обрізане таким чином, щоб зберігати фокус на автомобілі, а текстові блоки розташовуються вертикально, забезпечуючи зручність читання та взаємодії на сенсорних екранах. Розмір шрифтів також може бути скоригований для оптимальної читабельності.

Нижче наведено приклад дизайну цієї секції (див. Рис. 2.5).

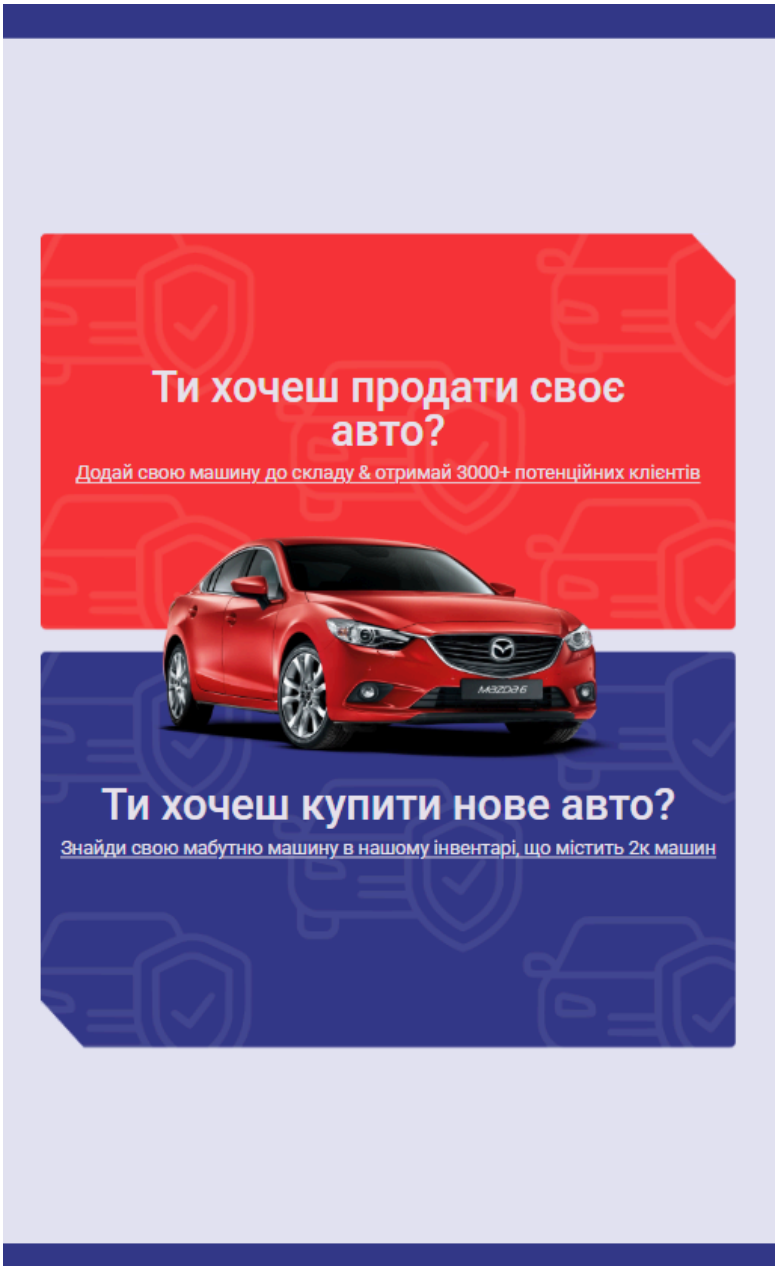


Рис. 2.5. Фото “візуально-орієнтованої” секції (мобільна версія)

Джерело: Розроблено автором

2.1.4. Проєктування секції "Швидкий пошук"

Секція "Швидкий пошук" (див. Рис. 2.6.) є одним з найважливіших функціональних блоків на лендінг-сторінці, оскільки вона безпосередньо відповідає на ключову потребу користувача – швидко знайти бажаний автомобіль. Її обґрунтування полягає у забезпеченні максимальної зручності та економії часу для відвідувачів, які не бажають переходити в повноцінний каталог або використовувати розширені фільтри одразу.

Візуально ця секція виконана у вигляді інтерактивної форми пошуку, яка містить ключові поля для введення параметрів автомобіля. Особливістю дизайну цієї секції є інтеграція візуальних ефектів для покращення користувацького досвіду. Це може бути анімація випадаючого меню, яка плавно з'являється та зникає, або ж елемент "красивого слайдера" з машинкою, що робить алюзію на дорогу. Цей слайдер або фонове зображення з динамічним ефектом покликані створити відчуття руху та швидкості, асоціюючись з комфортною подорожжю та легкістю пошуку. Вони додають естетичної привабливості та допомагають утримати увагу користувача.

Важливим аспектом є також кнопка "Пошук", розташована поруч з полями введення. Вона має бути чітко видимою та контрастною (наприклад, червоного кольору згідно з палітрою), щоб заохотити користувача до дії. Натискання на цю кнопку перенаправляє користувача на сторінку каталогу з вже відфільтрованими результатами.

Щодо адаптивності, на мобільних пристроях секція "Швидкий пошук" зберігає свою основну функціональність, але її елементи перебудовуються для оптимального відображення на менших екранах. Поля введення можуть розташовуватися вертикально, а анімації та слайдер адаптуються таким чином, щоб не перевантажувати мобільний інтерфейс і зберігати швидкість завантаження.

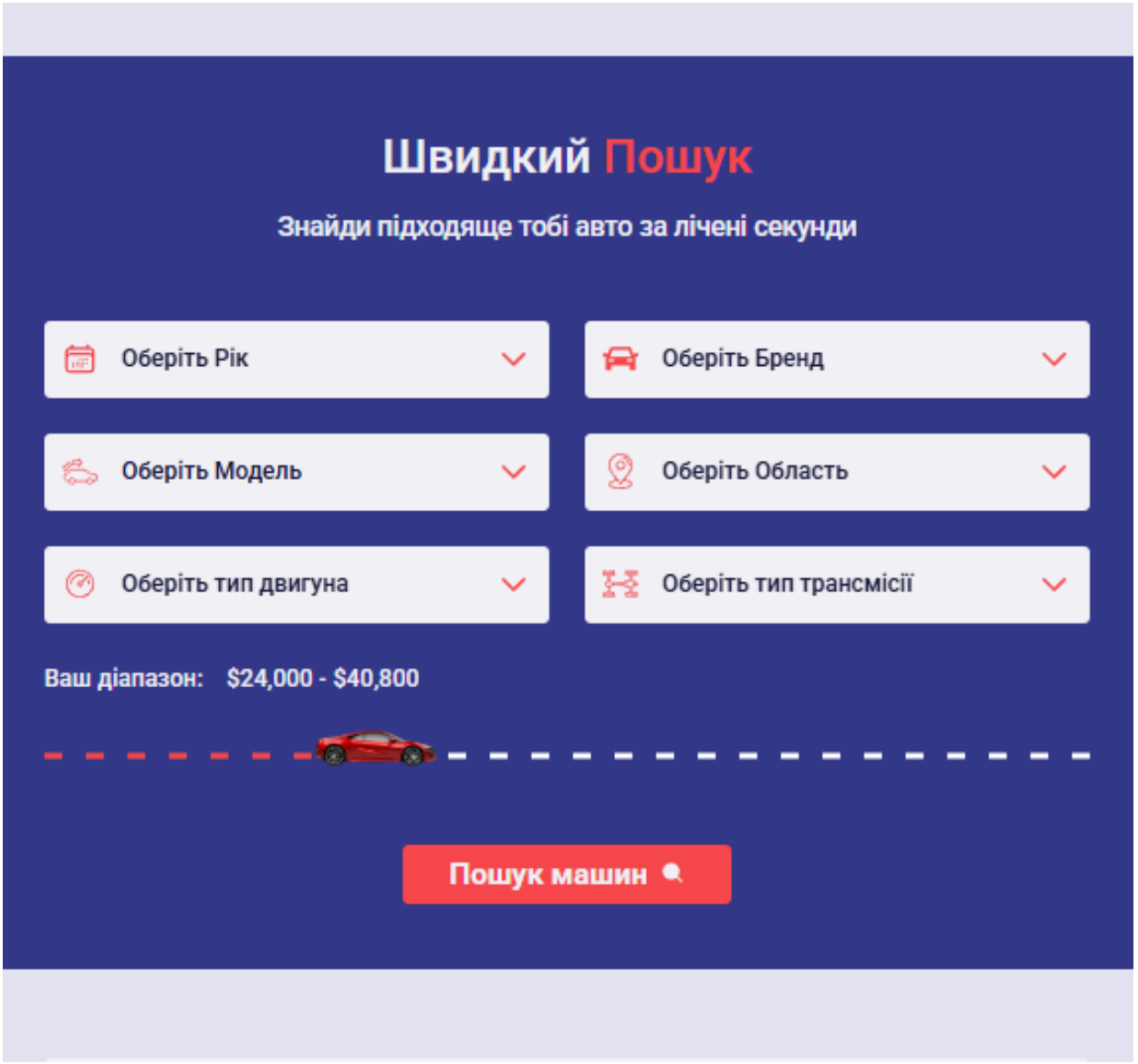


Рис. 2.6. Фото секції “Швидкий пошук” (планшетна версія)

Джерело: Розроблено автором

2.1.5. Проєктування форми зворотного зв'язку та футера

Завершальні секції лендінг-сторінки відіграють важливу роль у забезпеченні повноцінної взаємодії з користувачами, надаючи їм можливість зв'язатися з адміністрацією сайту та отримати доступ до важливої інформації.

Форма зворотного зв'язку ("Зв'яжіться з нами") Ця секція розроблена для тих користувачів, які не знайшли відповіді на свої питання в інших розділах сайту або бажають залишити відгук чи пропозицію. Її включення на лендінг-сторінку

обґрунтовується необхідністю забезпечити прямий канал комунікації, що підвищує довіру до платформи та покращує користувацький досвід.

Візуально форма зворотного зв'язку виконана у чистому, мінімалістичному стилі, що відповідає загальному дизайну маркетплейсу. Вона включає стандартні поля для введення: ім'я, електронну пошту (для відповіді) та текстове поле для повідомлення. Поля спроєктовані таким чином, щоб бути інтуїтивно зрозумілими та зручними для заповнення.

Футер (нижній колонтитул) Футер є обов'язковим елементом будь-якого веб-сайту, що надає додаткову навігацію та важливу інформацію. На лендінг-сторінці він виконує функцію підсумкового блоку, що містить посилання на ключові сторінки, контактну інформацію та відомості про права власності.

Дизайн футера виконаний у стриманих, нейтральних тонах з розробленої палітри, щоб не відволікати увагу від основного контенту сторінки, але при цьому забезпечувати його читабельність.

Адаптивність футера проявляється в тому, що на мобільних пристроях він перетворюється з багатоколонкової структури (як на десктопі) на одноколонкову. Посилання та контактна інформація розташовуються вертикально, що полегшує їхній перегляд та взаємодію на маленьких екранах. Текстовий контент футера також може адаптуватися, зменшуючи розмір шрифту, щоб забезпечити оптимальну читабельність без перевантаження простору.

Нижче наведено приклад дизайну форми зворотного зв'язку та футера (див. Рис. 2.7. та див. Рис. 2.8.)

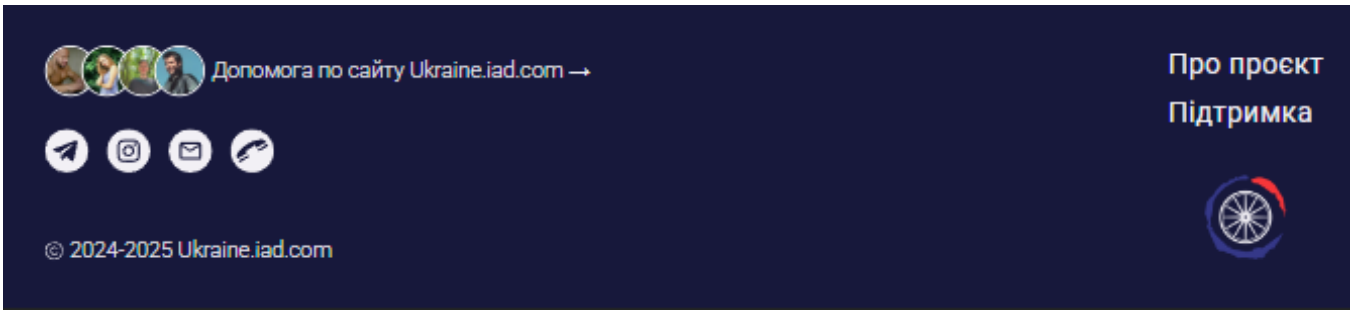


Рис. 2.7. Фото футера (планшетна версія)

Джерело: Розроблено автором

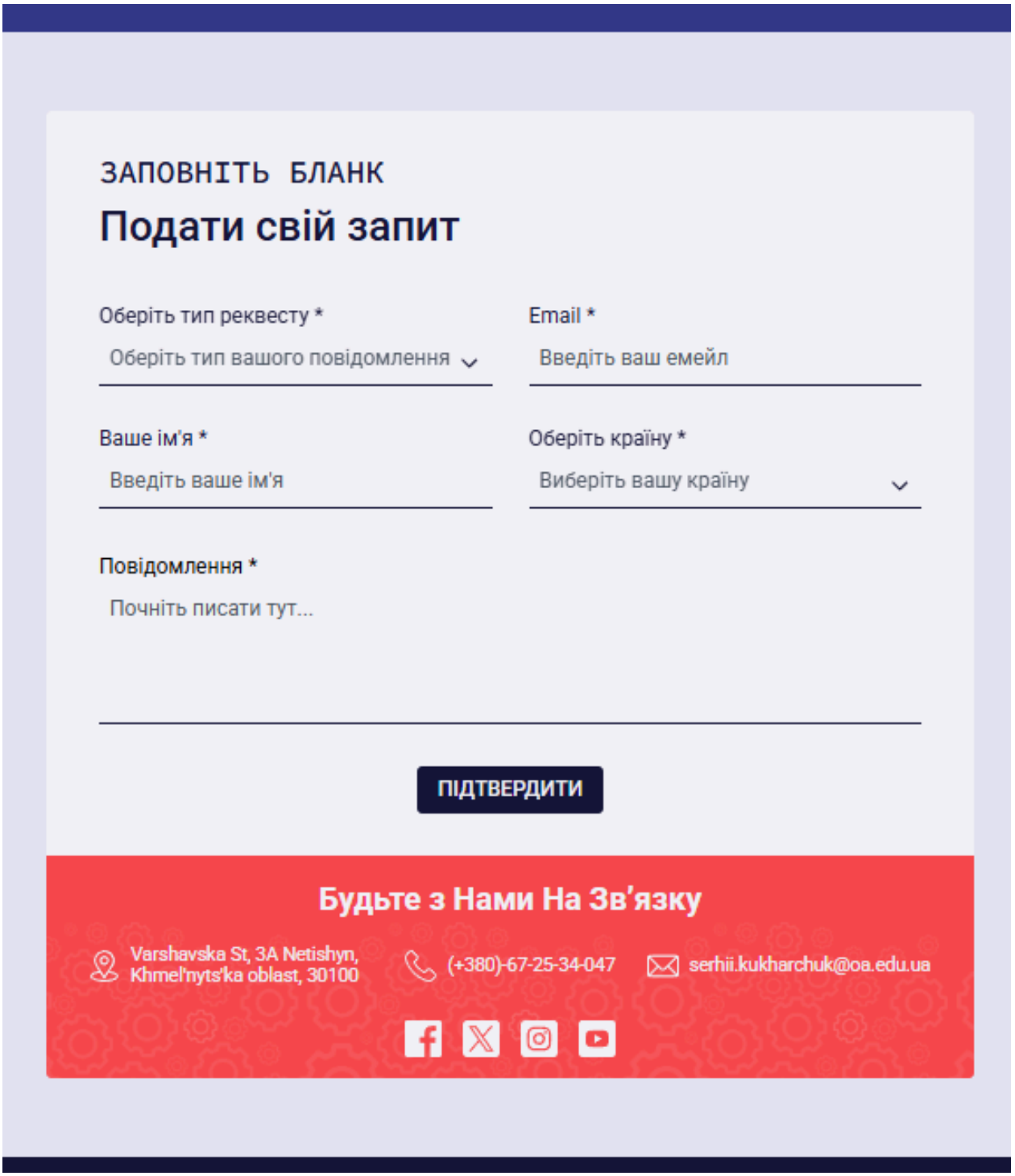


Рис. 2.8. Фото секції “Швидкий пошук” (планшетна версія)

Джерело: Розроблено автором

2.2. Організація розробки та управління кодом за допомогою Git і GitHub

У процесі розробки клієнтської частини маркетплейсу було організовано ефективне управління кодом за допомогою системи контролю версій Git [10] та

платформи GitHub [11], що є стандартом у сучасній командній розробці програмного забезпечення.

Git — це розподілена система керування версіями, яка забезпечує збереження повної історії змін у проєкті, дозволяє ефективно відстежувати внесені правки та, у разі потреби, повертатися до попередніх станів коду. Крім того, Git дозволяє працювати над проєктом одночасно декільком розробникам, розподіляючи завдання між гілками (branches), що мінімізує конфлікти при інтеграції змін.

Серед основних переваг використання Git у проєкті можна виділити:

- збереження повної історії змін;
- можливість створення окремих гілок для нових функцій або експериментів;
- швидке переключення між версіями коду;
- контроль над інтеграцією змін за допомогою механізмів злиття (merge) та повторного базування (rebase).

Для хостингу репозиторію та організації командної роботи було обрано **GitHub** — популярну хмарну платформу, що забезпечує зручні інструменти для колаборації. GitHub не лише зберігає код у централізованому сховищі, але й підтримує створення *pull requests*, що дозволяють ініціювати рев'ю змін перед їхнім злиттям до основної гілки. Це сприяє підвищенню якості коду та прозорості командної взаємодії.

Серед функцій GitHub, які активно використовувалися у проєкті:

- створення окремих гілок для розробки нових компонентів;
- використання pull requests для перевірки та обговорення змін;
- коментування коду безпосередньо в межах запитів на злиття;
- ведення журналу змін (issue tracking) та інтеграція з іншими сервісами (наприклад, CI/CD).

Вибір **GitHub** як основної платформи зумовлений його широкою підтримкою серед розробників, зручним інтерфейсом, а також розвиненою екосистемою інтеграцій з інструментами автоматизації, аналізу коду, тестування та розгортання. У порівнянні з GitLab чи Bitbucket, GitHub продемонстрував більшу гнучкість у підтримці сторонніх сервісів, зокрема при налаштуванні конвеєрів CI/CD за

допомогою GitHub Actions.

Таким чином, застосування Git у поєднанні з GitHub забезпечило надійне середовище для керування кодом, сприяло організованій співпраці над проєктом та дозволило мінімізувати ризики, пов'язані з конфліктами при інтеграції змін, що є критично важливим для успішної реалізації сучасних вебдодатків.

➢ Організація роботи у проєкті

Уся робота над клієнтською частиною сайту зберігалася у репозиторії GitHub. У цьому проєкті було організовано панель керування, де ми використовували три основні стани для завдань (див. Рис 2.9):

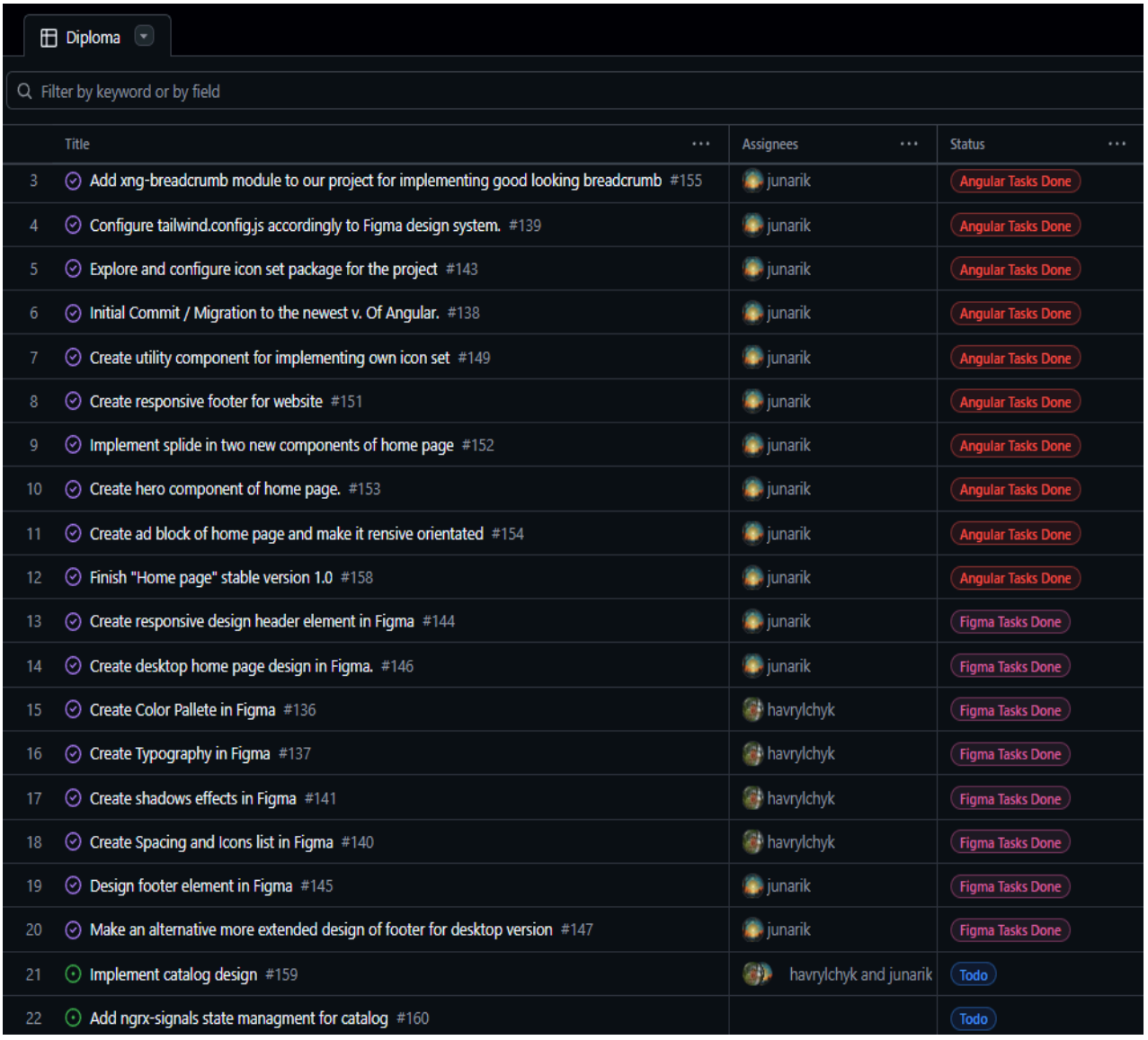


Рис. 2.9. Приклад-фото тасків на панелі керування

Джерело: Розроблено автором

Пояснення до рисунка 2.9.

- **To do:** завдання, які потрібно виконати.
- **In process:** завдання, що перебувають у процесі розробки.
- **Done:** завершені завдання.

Кожна функціональність чи компонент розроблялися в окремій гілці. Внесені зміни регулярно фіксувалися через коміти для збереження історії роботи. Після завершення роботи над гілкою створювався **pull request** для злиття змін з основною гілкою ***master***.

Pull request переглядала команда, і для його затвердження був потрібен **approve** від хоча б одного члена команди. Після схвалення гілка змерджувалася в ***master***, і внесені зміни ставали доступними в основній версії проєкту.

Такий підхід забезпечує прозорість, контроль за кодом і легкість у впровадженні нових функціональностей.

2.3. Проектування модульної архітектури

Модульна архітектура [12] є основою для створення масштабованих, багаторазових і зручних у підтримці застосунків в Angular. Вона забезпечує структуроване розділення функціональності, інкапсулюючи компоненти, сервіси, директиви, пайпи (канали) та інші залежності у так званих **"NgModules"**.

Angular побудований навколо концепції модулів, що робить їх невід'ємною частиною кожного застосунку. Навіть найпростіший Angular-застосунок має хоча б один модуль — **AppModule**, який є кореневим модулем. Зі зростанням складності проекту модульна архітектура дозволяє організовувати код, розділяючи застосунок на логічні блоки.

2.3.1. Переваги модульної архітектури

У таблиці 2.3.1 узагальнено основні переваги використання модульної архітектури у клієнтській частині інформаційної системи.

Таблиця 2.3.1

Переваги модульної архітектури в Angular-застосунках

Перевага	**Опис**
Поділ відповідальності	Кожен модуль має чітке призначення:• SharedModule — для спільних компонентів, директив, пайпів.• FeatureModule — для конкретної функціональності (наприклад, товари, профілі). Це зменшує складність і підвищує зрозумілість.
Масштабованість	Можна легко додавати нові або змінювати існуючі модулі без шкоди для решти застосунку.
Багаторазове використання	Універсальні модулі (з кнопками, формами, логуванням) можна застосовувати повторно в різних частинах застосунку або в інших проєктах.
Зручність обслуговування	Код кожного модуля ізольований, що полегшує тестування, оновлення та рефакторинг. Логіка легко зрозуміла завдяки вузькій спеціалізації.
Швидкий розвиток	Команди можуть працювати паралельно над різними модулями (наприклад, аутентифікація та замовлення).
Ліниве завантаження (Lazy Loading)	Angular дозволяє завантажувати модулі лише за потреби, що зменшує початковий час завантаження. Особливо корисно у великих проєктах.

2.3.2. Типи модулів, які реалізовано в проекті

1. **Кореневий модуль (AppModule)**

   Містить базову конфігурацію застосунку. У ньому реєструються глобальні сервіси, кореневий компонент (***AppComponent***) і модулі, які необхідні для роботи застосунку.

2. **Функціональні модулі (Feature Modules)**

   Використовуються для розподілу логіки застосунку на логічні блоки. Наприклад, модулі для управління товарами, користувачами або

замовленнями.

4. **Ядро (Core Module)**

   Містить сервіси та функціонал, які мають бути доступними в усьому застосунку, наприклад, глобальний сервіс логування або авторизації.

5. **Модулі для маршрутизації (Routing Modules)**

   Використовуються для організації маршрутизації між модулями та компонентами. Наприклад, кожен функціональний модуль може мати свій власний модуль маршрутизації.

2.3.3. Додаткова оптимізація залежностей за допомогою Standalone Features

У межах цього проєкту було прийнято рішення використовувати standalone-підхід [13] виключно у структурі папки shared, що містить спільні компоненти. Це пояснюється тим, що багато з таких компонентів (як-от кнопки, індикатори завантаження, сповіщення) мають точкове використання на сторінках і не потребують централізованого імпорту через спільний модуль. У попередніх версіях Angular типовим підходом було створення SharedModule**,** який містив усі загальні компоненти, директиви та пайпи, що використовувалися в різних частинах застосунку. Така структура передбачала імпорт усього модуля навіть у випадках, коли насправді була потрібна лише одна з його складових. Це збільшувало кількість імпортованого коду та ускладнювало контроль над залежностями.

Використовуючи standalone-компоненти в папці shared, ми досягаємо більшої гнучкості: кожен компонент підключається рівно там, де він потрібен, без додаткових обгорток. Це не лише зменшує розмір бандлу, а й дозволяє точніше контролювати залежності всередині проєкту. Крім того, standalone-компоненти краще ізолюються один від одного, що значно спрощує їх тестування, повторне використання і підтримку.

Водночас важливо зазначити, що повний перехід на standalone-підхід для всієї архітектури — з повним уникненням модулів core та інших features — був би неефективним. Модулі продовжують відігравати важливу роль у структуризації

застосунку. Наприклад, ProductModule дозволяє логічно об’єднати пов’язані компоненти, сервіси та маршрути в один контекст і зручно організувати ліниве завантаження. Аналогічно, CoreModule забезпечує єдине джерело для глобальних сервісів і конфігурації застосунку, що є критично важливим для запобігання дублікації логіки.

Таким чином, у проєкті було свідомо поєднано обидва підходи: standalone-компоненти використовувалися для локалізованих, повторно застосовуваних частин інтерфейсу в межах папки shared, а традиційні модулі застосовувалися для побудови загальної логіки, структури та навігації. Такий баланс забезпечив оптимальну архітектуру, що забезпечила гнучкість, логічну організацію структури та зручність у масштабуванні й підтримці.

Тому з вищенаведеного формується така структура папок, зображена на рис. 2.10.

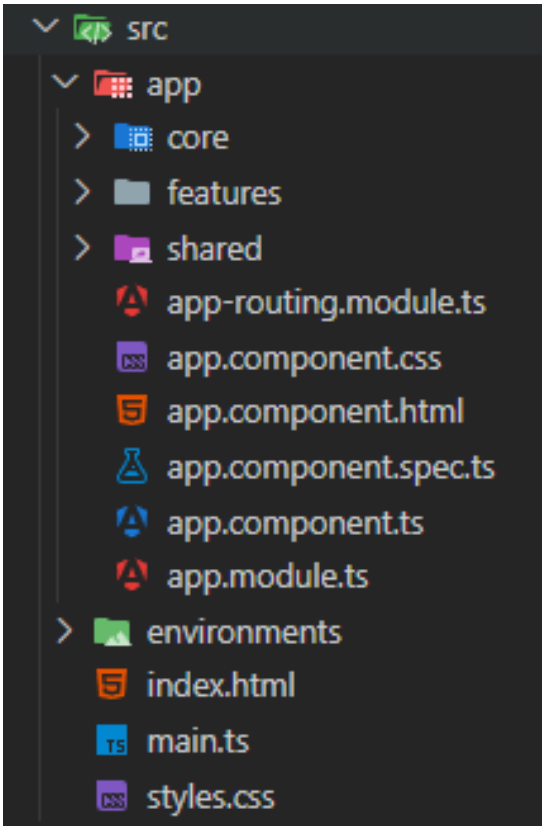


Рис. 2.10. Фото-приклад основи модульної структури власного проекту

Джерело: Розроблено автором

2.4. Підхід Dumb/Smart Components

Під час розробки проекту було введено додаткові вдосконалення до стандартних рекомендацій з організації модульного менеджменту коду — зокрема, реалізовано розділення компонентів за принципом Dumb/Smart [14]. Такий підхід дозволяє відокремити просту презентаційну логіку від складного управління станом і сервісної взаємодії, що підвищує читабельність, спрощує юніт-тестування та пришвидшує масштабування.

У рамках цієї методики “тупі” (Dumb) компоненти відповідають виключно за відображення даних. Вони отримують усі необхідні значення через ***@Input()***, емінують події через ***@Output()*** і не містять жодної бізнес-логіки або взаємодії з сервісами. Яскравим прикладом є компонент FastSearchSectionComponent (див. Лістинг 2.4.1), який лише рендерить форму пошуку і передає назовні натискання кнопок і введені значення, не знаючи про те, звідки надходять дані і куди вони передаються.

Лістинг 2.4.1. — “Тупий” компонент “FastSearchSectionComponent”

```
export class FastSearchSectionComponent {
  @Input() InfoObjectDataOptionated?: {
    ApiCallOption: string;
    DropDownElementUlInfo: Array<{ id: string; name: string }>;
  };
  @Input() priceRange$!: Observable<PriceRange>;
  @Output() getInfoForUlEvent = new EventEmitter<DropdownElementData>();

  handleGetInfoForUlEvent(ApiCallOption: DropdownElementData) {
    this.getInfoForUlEvent.emit(ApiCallOption);
  }
  @Output() changeRouterLinkEvent: EventEmitter<SearchAdvancedParams> =
    new EventEmitter<SearchAdvancedParams>();
  handleChangeRouterLink(searchAdvanced: SearchAdvancedParams): void {
    this.changeRouterLinkEvent.emit(searchAdvanced);  }}
```

І навпаки, “розумні” (Smart) компоненти концентрують у собі всю бізнес-логіку. Вони відповідають за завантаження та обробку даних із сервісів чи API, формують контекст для дочірніх Dumb-компонентів і координують їхню взаємодію. Прикладом

такого підходу є модульний компонент **HomeComponent** (див. Додаток А), що не лише запитує необхідні дані, але й передає їх у свої презентаційні підкомпоненти, забезпечує обробку помилок та реакцію на події користувача.

Така організація полегшує навігацію в проекті, оскільки основний smart-компонент завжди можна знайти за назвою модуля. Це створює чітку ієрархію, де кожен модуль має свій центральний компонент для управління всією функціональністю, пов'язаною з цим модулем.

Також, у кожному модулі проекту (core, features, shared) було створено папку “ui”, яка відповідає за компоненти відображення (dumb-компоненти). Така структура забезпечує чітке розділення відповідальностей та спрощує навігацію в проекті (див. Додаток В).

Додавання папки “ui” до модулів є вдосконаленням модульного менеджменту в Angular. Це не лише організує проект за принципом Dumb/Smart компонентів, але й робить його більш масштабованим, зрозумілим та зручним у підтримці. Такий підхід є особливо корисним для великих застосунків, де чітка структура та розподіл відповідальності є ключовими для успішної розробки.

Також варто зазначити, що важливою складовою будь-якого швидкодіючого сайту є ліниве завантаження. В **Angular** його реалізація завдяки модульній системі є досить зручною та ефективною. Наприклад, у нашому проекті є основний файл маршрутизації (***app-routing.module.ts***, див. Лістинг 2.4.2), де ми визначаємо базові маршрути для застосунку. До цього основного роутингу ми можемо підключати інші модулі з тим же лінивим завантаженням, що дозволяє оптимізувати роботу застосунку. Це досягається шляхом завантаження лише тих модулів або ж компонентів, які потрібні для конкретного маршруту, а не всього функціоналу одразу.

Лістинг 2.4.2. — код-файл основного роутинга проекту

```
const routes: Routes = [
{
path: '',
title: 'Home',
component: HomeComponent,
```

```
data: { breadcrumb: { label: 'Колеско', info: 'home_xng'} } },
{
path: 'catalog',
title: 'catalog',
loadChildren: () => import('./features/catalog/catalog.module').then(m =>
m.CatalogModule),
data: { breadcrumb: { label: 'Каталог'} }
},
{ path: '**', component: NotFoundComponent },
];
@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule],
})
export class AppRoutingModule { }
```

2.5. Switch проекту до ChangeDetectionStrategy.OnPush для зменшення зайвих перерендерів компонентів.

У рамках оптимізації продуктивності клієнтської частини додатку було прийнято рішення перейти до точкового рендерингу компонентів, з відмовою від повної перевірки всього дерева при кожній зміні. Цей підхід дозволяє оновлювати лише ті компоненти, стан яких справді зазнав змін, що суттєво зменшує навантаження на механізм відображення.

Для реалізації цього підходу я використав можливості, які надає Angular у поєднанні з zone.js [8], що дозволяє виявляти зміни у стані додатку. Тепер рендеринг компонента відбувається лише тоді, коли спрацьовує один із двох основних тригерів, які ініціюють перевірку змін у компонентному дереві. Їх візуальне представлення наведено на рис. 2.11.

Рис. 2.11. Тригери, які змушують **zone.js** ініціювати перевірку стану компонентного дерева

Джерело: розроблено автором

Щоб помітити компонент, як такий, що повинен оновити свій HTML відповідно до актуального стану залежностей, я використовую один із п’яти доступних механізмів, представлених на рис. 2.12.

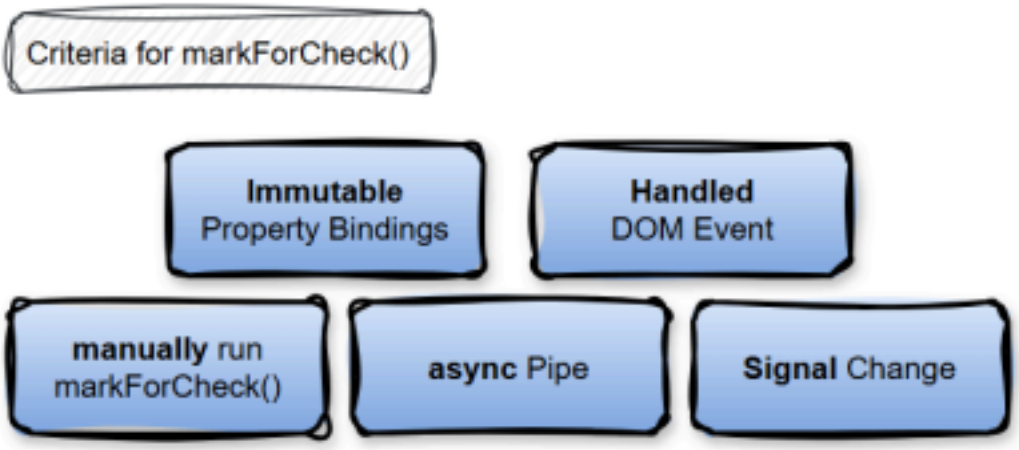


Рис. 2.12. Механізми, які дозволяють Angular позначити компонент як такий, що потребує оновлення

Джерело: розроблено автором

Застосування цих підходів дозволяє Angular ефективно оновлювати лише змінені компоненти, уникаючи зайвих обчислень та підвищуючи загальну продуктивність додатку.

2.6. Інтеграція NgRx Signal Store та застосування Signal API у модулі Catalog

Модуль Catalog у межах розробки клієнтської частини став одним із найбільш функціонально насичених усього застосунку. Він охоплює відображення списку транспортних засобів, реалізацію фільтрації, пагінації, взаємодію з API, керування великою кількістю залежних станів, а також інтеграцію з маршрутизацією (routing), яка дозволяє зберігати поточний стан фільтрів, сторінки та пошукових критеріїв у URL. Завдяки цьому користувач має можливість поділитися посиланням на конкретний запит або повернутися до нього пізніше без втрати контексту.

Саме через таку складність і взаємозалежність функціональних частин модуля було прийнято рішення впровадити NgRx Signal Store [5] у поєднанні з Signal API [6], що дозволило суттєво спростити управління даними, підтримати синхронізацію з маршрутом і забезпечити високу продуктивність.

Signal Store виявився ідеальним рішенням для цього модуля з кількох причин:

- **Зменшення шаблонного коду:** Завдяки Signal API відпала потреба у створенні складних ефектів, селекторів та ред’юсерів, характерних для класичного NgRx. Це дозволило сфокусуватись на бізнес-логіці, а не на інфраструктурному коді.
- **Чистота архітектури:** Усі дані, пов’язані з транспортними засобами та фільтрами, були винесені у спеціалізовані Signal Stores. Це дало змогу централізувати логіку та досягти високого рівня модульності, що особливо важливо для великого модуля, як-от Catalog.
- **Покращене керування життєвим циклом:** Завдяки методу withHooks вдалося автоматизувати процес завантаження та очищення стану при ініціалізації та знищенні стору. Такий підхід знижує ризик помилок, пов'язаних із ручним викликом методів у компонентах, та забезпечує стабільну синхронізацію з API.
- **Реактивність без RxJS:** Застосування Signal API дозволяє працювати з реактивністю на рівні шаблону без потреби підписок, async pipe чи додаткової

обробки стрімів. Дані зберігаються у вигляді signal, які автоматично оновлюють інтерфейс при зміні. Це зробило код у компонентах значно чистішим і легшим для супроводу.

- **Оптимізація рендерингу:** Поєднання Signal API з Change Detection стратегією ***OnPush*** дозволяє оновлювати лише ті частини інтерфейсу, стан яких дійсно змінився. Це особливо важливо в умовах великого списку транспортних засобів і частих змін фільтрів чи критеріїв пошуку.

І також додаткову гнучкість у побудові архітектури надали ***signalStoreFeatures***, які можна порівняти з гнучким конструктором на кшталт “лего”. Вони дозволили розділити окремі частини бізнес-логіки та стану на незалежні блоки, які згодом були легко об’єднані в єдину цілісну систему.

У результаті впровадження Signal Store у модулі Catalog дало змогу побудувати ефективну, масштабовану та легко підтримувану архітектуру. Управління станом стало передбачуваним, а інтеграція з API — простішою і менш залежною від компонентів. Це дозволило зосередитися на розробці функціональності, не витрачаючи зайві зусилля на підтримку складної логіки керування станом.

Приклад кінцевої структури файлів модулю Catalog після інтеграції з NGRX signal store [3] можна переглянути знизу на рисунку 2.13.

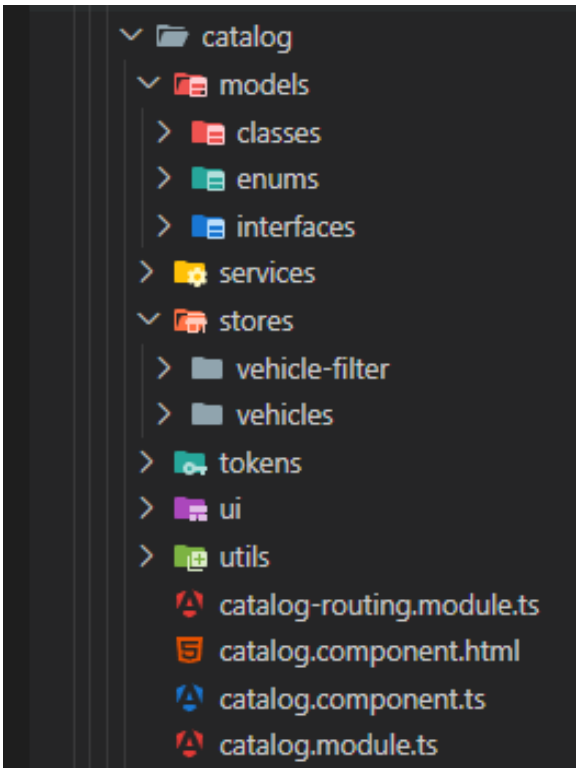


Рис. 2.13. Структура файлів Catalog

Джерело: розроблено автором

2.7. Імплементація дизайн системи у проекті

Для впровадження кольорової палітри та загального дизайну було обрано **Tailwind CSS** — фреймворк для стилізації, що надає значні переваги завдяки своїй зручності у використанні та можливості писати класи безпосередньо у HTML. Багатий набір вбудованих класів Tailwind CSS значно спрощує і прискорює процес створення адаптивного дизайну [15].

Однією з ключових переваг Tailwind CSS є можливість кастомізації через конфігураційний файл. Це дозволяє легко створювати унікальну кольорову палітру, додавати власні відступи, шрифти, розміри тексту та інші параметри, враховуючи вимоги дизайну (див. Додаток С).

Для забезпечення крос-браузерної сумісності та адаптивного дизайну використовуються підходи responsive design, які дозволяють створювати сайти, що правильно відображаються на різних пристроях та розмірах екранів. У цьому процесі важливу роль відіграє Tailwind CSS, який надає зручні інструменти для

налаштування стилів під різні розміри екранів завдяки системі *breakpoints*.

Breakpoints позначаються префіксами, такими як *sm:, md:, lg:, xl:, 2xl:* і застосовуються перед класами, які потрібно змінити. Нижче наведено приклад створення кнопки з використанням цього підходу у лістингу 2.7.1.

Лістинг 2.7.1 — частина коду у tailwind.config.ts файлі

```
theme: {
   container: {
     center: true,
     padding: {
       DEFAULT: "16px",
       md: "32px",
       lg: "32px",
       2xl: "120px",
     },
     screens: {
       sm: "640px",
       md: "768px",
       lg: "1024px",
       xl: "1280px",
       2xl: "1440px",
     },
   },

   colors: {
     ...COLORS,
     content: {
       primary: COLORS.portGore[950],
       secondary: COLORS.portGore[700],
       tertiary: COLORS.portGore[400],
       disable: COLORS.portGore[200],
       informative: COLORS.kimberly[500],
       positive: COLORS.lima[500],
       error: COLORS.red[600],
     },
   },

   fontFamily: {
     code: ["Source Code Pro", "monospace"],
     ubuntu: ["Ubuntu", "sans-serif"],
     roboto: ["Roboto", "sans-serif"],
   },
```

2.8. Підключення та інтеграція додаткових бібліотек компонентів

У процесі розробки складних веб-застосунків часто виникає потреба в використанні готових рішень для вирішення типових задач. Компонентні бібліотеки надають розробникам набір готових UI-елементів, що дозволяє значно прискорити процес розробки та покращити якість коду.

2.8.1. Splide

Splide - це легка, гнучка та доступна бібліотека JavaScript для створення слайдерів. Вона надає широкий спектр можливостей для налаштування та кастомізації, дозволяючи створювати унікальні слайдери, що відповідають потребам будь-якого проєкту. [16]

Переваги бібліотеки Splide для реалізації слайдерів наведено у таблиці 2.8.1.

Таблиця 2.8.1

Переваги Splide

Перевага	**Опис**
Легкість та швидкість	Splide має невеликий розмір файлу та високу продуктивність, що забезпечує швидке завантаження та плавну роботу слайдерів.
Доступність	Відповідає стандартам WCAG, що робить слайдери зручними для користувачів з обмеженими можливостями.
Простота використання	Має зрозумілий API, що полегшує інтеграцію у проєкти будь-якої складності.
Кросбраузерність	Працює стабільно в усіх сучасних браузерах: Chrome, Firefox, Safari, Edge тощо.

Slider JS Web Component

Slider JS Web Component — це бібліотека для створення слайдерів, яка

базується на веб-компонентах.

Чому обрано Splide, а не Slider JS Web Component?

У моєму проєкті було обрано **Splide** через його численні переваги в порівнянні з Slider JS Web Component, з яким я працював раніше:

1. **Гнучкість і налаштування**

   Splide пропонує більше можливостей для кастомізації зовнішнього вигляду та поведінки слайдерів, дозволяючи легко адаптувати їх до потреб проєкту.

2. **Легкість**

   Splide має менший розмір файлу та забезпечує кращу продуктивність, що позитивно впливає на швидкість завантаження сторінки.

3. **Інтеграція з Angular**

   Splide значно краще інтегрується з Angular завдяки простішій структурі API та сумісності з Angular-компонентами. Це спрощує процес імплементації та дозволяє ефективно використовувати Angular-екосистему.

4. **Активна спільнота**

   Splide має велику та активну спільноту користувачів і розробників, що сприяє постійному розвитку бібліотеки. Це також надає доступ до великої кількості ресурсів і підтримки.

Завдяки цим перевагам Splide дозволила створити слайдер, який повністю відповідає технічним вимогам проєкту. Саму ж інформацію щодо імплементації я черпав з блогу про Splide та на офіційному сайті, а також із власного досвіду.

Самі ж приклади готовогих рішеннь імплементованих за допомогою цієї бібліотеки можна переглянути на фото нижче: (див. Рис. 2.14, див. Рис. 2.15, див. Рис. 2.16, див. Рис. 2.17).

Рис. 2.14. Темно-синій дизайн слайдера, UI/UX якого реалізовано за допомогою splide бібліотеки

Джерело: Розроблено автором

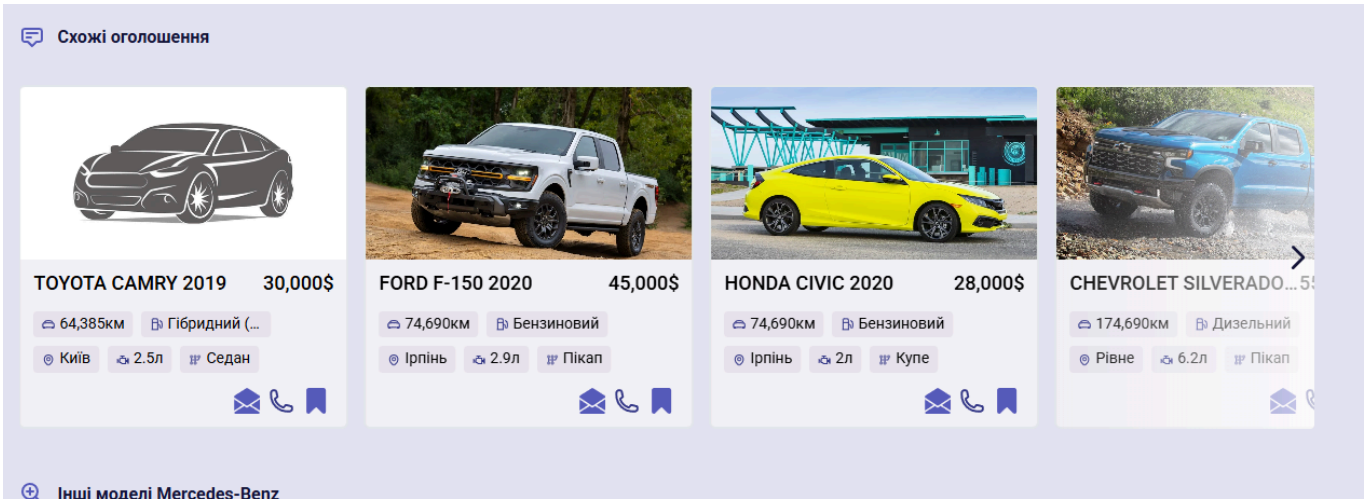


Рис. 2.15. Slider вигляд на сторінці оголошення продукту

Джерело: Розроблено автором

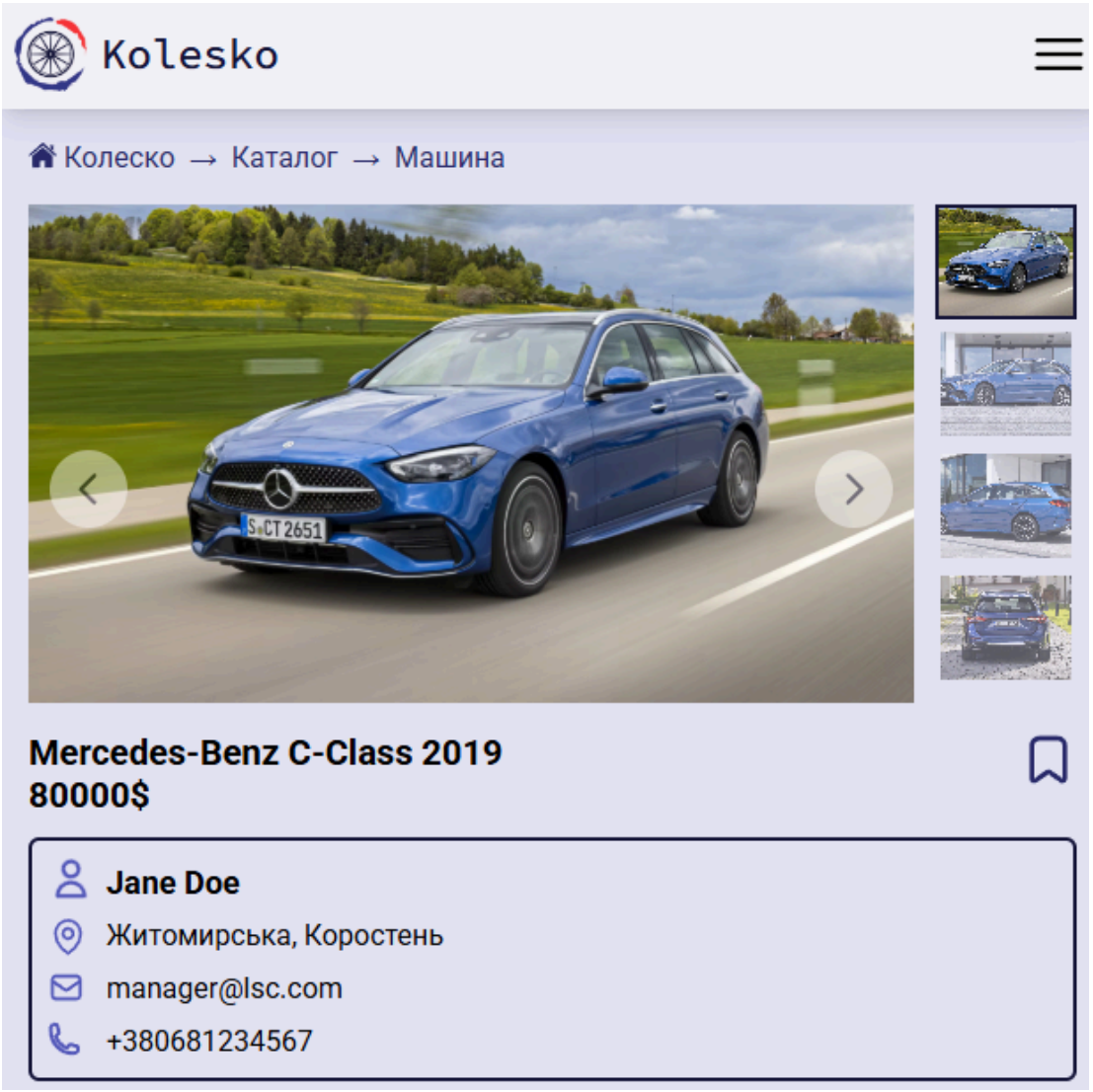


Рис. 2.16. Carousel-slider на мобілці

Джерело: Розроблено автором

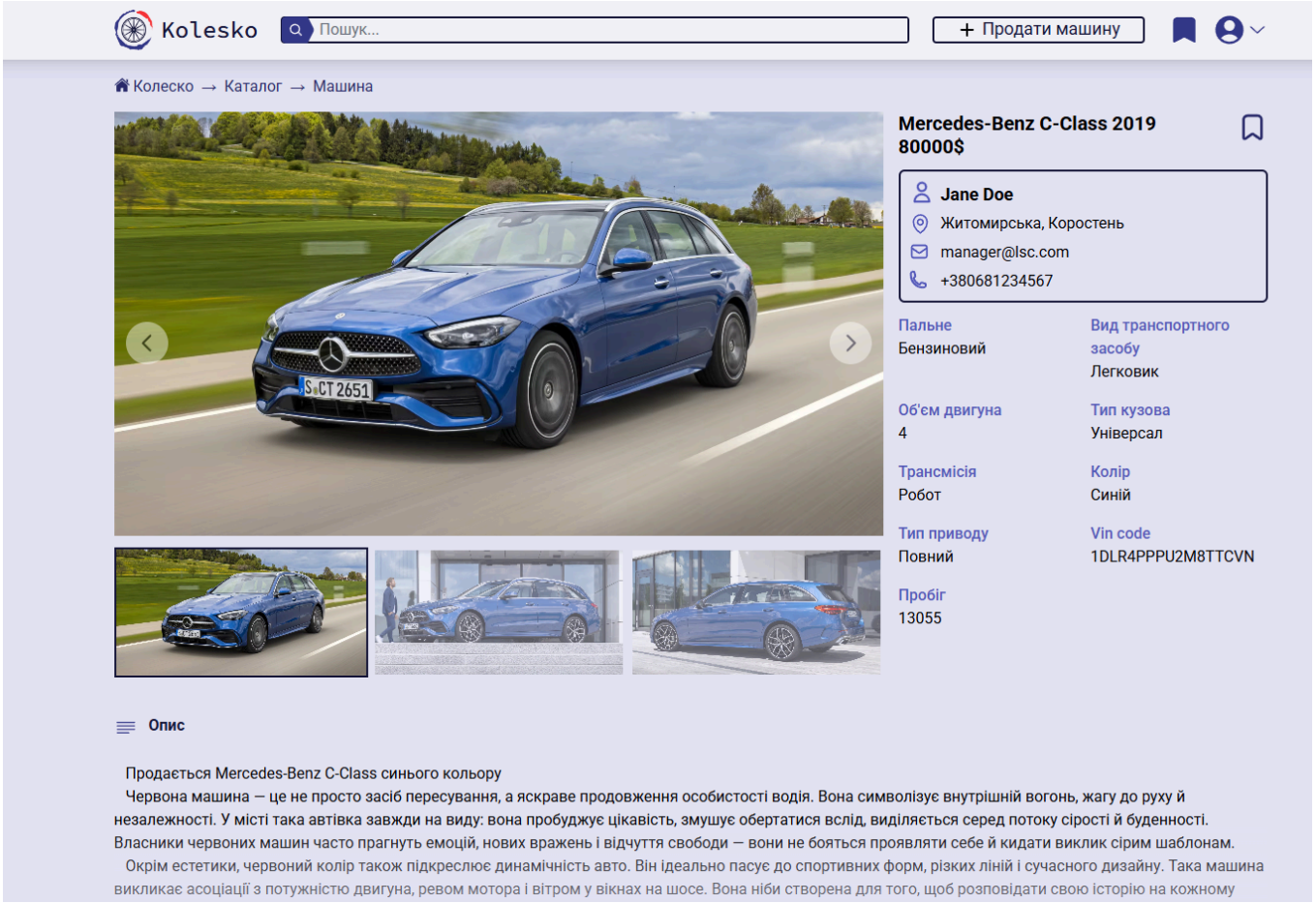


Рис. 2.17. Carousel slider із мініатюрами на desktop resolution

Джерело: Розроблено автором

2.8.2. Xng-breadcrumb

xng-breadcrumb — це бібліотека для Angular, яка використовується для створення навігаційних елементів. Вона дозволяє легко відобразити поточний шлях користувача у вигляді ієрархічної структури, надаючи чіткий орієнтир для користувача в межах веб-додатку.

Чому обрано xng-breadcrumb?

У моєму проєкті я обрав xng-breadcrumb [17] через його переваги порівняно з іншими бібліотеками:

1. ***Проста інтеграція з Angular***

Бібліотека створена спеціально для Angular, тому її легко інтегрувати з Angular-маршрутизатором. Налаштування передбачає використання декораторів, що спрощує процес додавання «хлібних крихт» до кожного маршруту.

2. ***Мінімалістичний дизайн і легкість***

xng-breadcrumb має невеликий розмір і не містить зайвих залежностей. Це забезпечує кращу продуктивність і не перевантажує проєкт.

3. ***Динамічність***

Можливість динамічно оновлювати назви маршрутів і змінювати параметри, що особливо важливо для роботи з великими проєктами.

4. ***Гнучкість***

Підтримка налаштувань для кастомізації стилів і поведінки. Вона дозволяє використовувати власні шаблони для виведення «хлібних крихт», адаптуючи їх до дизайну проєкту.

5. ***Документація і підтримка***

Бібліотека має зрозумілу і структуровану документацію, яка дозволяє швидко розібратися в основних функціях і можливостях.

Раніше я працював із іншими бібліотеками, подібні до ngx-breadcrumbs, але вони мали деякі недоліки, зокрема складність налаштування або недостатню підтримку Angular. xng-breadcrumb виявився більш простим і інтуїтивним для роботи, особливо коли йдеться про веб-сайти зі складними маршрутами. Його API та гарно розписана документація дозволяє адаптовувати компонент під різні сценарії, що значно полегшило роботу з інтерфейсом під час написання. Лістинг коду імплементації в основному модулі проекту та інтеграцію у routing-module можна переглянути нижче: (див. Лістинг 2.8.2.1, див. Лістинг 2.8.2.2).

Лістинг 2.8.2.1 — код основного app.ts компонента з інтегрованим xng-breadcrumb тегом

```
<div class="wrapper">
  <app-header></app-header>
  <div class="pt-sm pb-[14px]">
    <app-breadcrumb-holder
[isMainPage]="isMainPage"></app-breadcrumb-holder>
  </div>

  <router-outlet></router-outlet>
  <app-footer></app-footer>
```

```
</div>
```

Лістинг 2.8.2.2 — код app-routing.module.ts з інтегрованим breadcrumb взятої зі стороньої бібліотеки xng-breadcrumb

```
const routes: Routes = [
{
path: '',
title: 'Home',
component: HomeComponent,
data: { breadcrumb: { label: 'Колеско', info: 'home_xng'} } },
{
path: 'catalog',
title: 'catalog',
loadChildren: () => import('./features/catalog/catalog.module').then(m =>
m.CatalogModule),
data: { breadcrumb: { label: 'Каталог'} }
},
{ path: '**', component: NotFoundComponent },
];

@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule],
})

export class AppRoutingModule { }
```

2.7.3. Ngx-toastr

ngx-toastr — це бібліотека для Angular [18], призначена для виведення повідомлень типу *toast* (спливаючих повідомлень), яка забезпечує зручний спосіб інформування користувача про події в додатку: успішні дії, попередження або помилки.

Чому обрано ngx-toastr?

У моєму проєкті я обрав ngx-toastr через наступні переваги:

1. **Зручність і простота використання:** ngx-toastr має інтуїтивно зрозумілий API, що дозволяє легко викликати повідомлення про помилки або інші події. Наприклад: ***this.toastr.error('Помилка під час збереження даних', 'Помилка');***

2. **Швидка інтеграція з Angular**

   Бібліотека легко підключається до Angular-застосунку через модуль ToastrModule, який імпортується у AppModule. Налаштування займає мінімум часу.

3. **Гнучкість налаштування**

   Можна задавати тривалість показу, позицію на екрані, стилі, а також використовувати власні шаблони для toast-повідомлень. Це дозволяє гармонійно інтегрувати їх в існуючий дизайн сайту.

4. **Кросбраузерність і продуктивність**

   Повідомлення відображаються стабільно у всіх популярних браузерах, не створюючи навантаження на продуктивність застосунку.

5. **Візуальна привабливість і кастомізація**

   Повідомлення виглядають сучасно й акуратно, при цьому можуть бути додатково стилізовані під дизайн-політику проєкту за допомогою CSS.

ВИСНОВОК

У результаті виконання кваліфікаційної роботи було успішно розроблено клієнтську частину для маркетплейсу вживаних автомобілів «Kolesko». Цей проект охопив увесь цикл створення сучасного фронтенд-застосунку, приділяючи особливу увагу вимогам до продуктивності, архітектурної гнучкості та забезпечення позитивного користувацького досвіду.

На початковому етапі було проведено аналіз предметної області та конкурентних рішень на українському ринку, що дозволило сформувати чіткі вимоги до функціональності та дизайну платформи. Для реалізації клієнтської частини було обрано фреймворк Angular, який забезпечив потужний інструментарій для створення масштабованої та добре структурованої архітектури. Це включало застосування таких підходів, як модульність, ліниве завантаження (Lazy Loading) та оптимізаційні стратегії Change Detection, включно з OnPush, а також інтеграцію сучасного Signal API для ефективного управління станом та рендерингом.

Було спроєктовано та реалізовано інтуїтивно зрозумілий та візуально привабливий дизайн лендінгу, що включає ключові секції для взаємодії з користувачем та презентації автомобілів. Інтерфейс розроблявся з урахуванням адаптивності для різних пристроїв та базувався на попередньо розробленій дизайн-системі.

З метою забезпечення високої продуктивності та ефективного управління даними, особливо в таких складних розділах, як каталог з фільтрацією та пагінацією, було інтегровано NgRx Signal Store. Архітектурні рішення, зокрема поділ на "розумні" та "презентаційні" компоненти (Smart/Dumb components) та використання standalone-компонентів для спільних елементів, сприяли покращенню читабельності коду, його повторному використанню та легкості підтримки.

Процес розробки супроводжувався налагодженою системою контролю версій Git та управлінням проектом на платформі GitHub, що забезпечило прозорість, командну взаємодію та контроль якості коду.

Таким чином, розроблена клієнтська частина маркетплейсу «Kolesko» є сучасним, технологічно обґрунтованим рішенням, що відповідає актуальним стандартам фронтенд-розробки. Проект демонструє успішне застосування передових підходів Angular для створення продуктивного, масштабованого та зручного для користувача веб-додатку, готового до подальшого розвитку та інтеграції.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. StackBlogger. Sharing Data Between Components using RxJS BehaviourSubject in Angular. URL: https://stackblogger.com/sharing-data-between-components-rxjs-angular/ (дата звернення: 30.04.2025).
2. Kumar R. Simplifying Component Communication in Angular: A Step-by-Step Guide. CodezUp. 2023. URL: https://codezup.com/simplifying-component-communication-angular-step-by-step-guide/ (дата звернення: 10.05.2025).
3. NgRx. Store: Introduction. URL: https://ngrx.io/guide/store (дата звернення: 01.05.2025).
4. NGXS. NGXS: State Management for Angular. URL: https://www.ngxs.io/ (дата звернення: 15.05.2025).
5. NgRx. SignalStore. URL: https://ngrx.io/guide/signals/signal-store (дата звернення: 25.04.2025).
6. Angular.dev. Signals. URL: https://angular.dev/guide/signals (дата звернення: 20.05.2025).
7. Jahollari E. Deep dive into the OnPush change detection strategy in Angular. Angular.Love. URL: https://angular.love/deep-dive-into-the-onpush-change-detection-strategy-in-angular (дата звернення: 28.04.2025).
8. Alam S. What is Zone.js in Angular?. Medium. 2022. URL: https://medium.com/@sehban.alam/what-is-zone-js-in-angular-e0029c21c32f (дата звернення: 02.05.2025).
9. Angular.dev. signal Function. URL: https://angular.dev/api/core/signal (дата звернення: 14.05.2025).
10. Git. Git Documentation. URL: https://git-scm.com/ (дата звернення: 30.03.2025).
11. GitHub. GitHub. URL: https://github.com/ (дата звернення: 01.04.2025).
12. Alam S. Mastering Modular Architecture in Angular. Medium. 2023. URL: https://medium.com/@sehban.alam/mastering-modular-architecture-in-angular-4cc2632fc964 (дата звернення: 18.05.2025).
13. Angular Docs (v17). Standalone components. URL: https://v17.angular.io/guide/standalone-components (дата звернення: 11.04.2025).
14. Aung J. Dumb Components and Smart Components. Medium. 2018. URL: https://medium.com/@thejasonfile/dumb-components-and-smart-components-e7b33a698d43 (дата звернення: 09.05.2025).
15. Tailwind CSS. Tailwind CSS Documentation. URL: https://tailwindcss.com/ (дата звернення: 04.05.2025).

16. SplideJS. Splide: JavaScript Slider & Carousel. URL: https://splidejs.com/ (дата звернення: 12.05.2025).
17. npm. xng-breadcrumb. URL: https://www.npmjs.com/package/xng-breadcrumb (дата звернення: 05.05.2025).
18. npm. ngx-toastr. URL: https://www.npmjs.com/package/ngx-toastr (дата звернення: 07.05.2025).

ДОДАТКИ

ДОДАТОК А

```
export class HomeComponent implements OnInit {
  InfoObjectDataOptionated: {
    ApiCallOption: string;
    DropDownElementUlInfo: Array<{ id: string; name: string }>;
  } = { ApiCallOption: '', DropDownElementUlInfo: [] };

  handleGetInfoForUlEvent(ApiCallOption: DropdownElementData) {
    this.homeservice
      .getDropDownData(ApiCallOption)
      .pipe(
        filter(
          (
            response:
              | ApiResponse<
                  Brand | Model | LocationRegion | TransmissionType |
EngineType
                >
              | undefined
          ) => {
            return response !== undefined;
          }
        ),
        map(
          (
            response: ApiResponse<
              Brand | Model | LocationRegion | TransmissionType |
EngineType
            >
          ) =>
            response.items.map((item) => {
              return {
                id: item.id,
                name: item.name,
              };
            })
        ) ).subscribe((info: { id: string; name: string }[]) =>
{
this.InfoObjectDataOptionated = {
```

Продовження додатку А

```
          ApiCallOption: ApiCallOption,
          DropDownElementUlInfo: info,
        };
      });
  }

  vehicleCount: number = 20;
  vehicles: Vehicle[] = [];
  bodyTypes: BodyType[] = [];

  filteredBodyTypes: BodyType[] = [];

  topPriceVehicles: Vehicle[] = [];
  priceRange$!: Observable<PriceRange>;

  constructor(private homeservice: HomeService, private router: Router) {}
  ngOnInit(): void {
    this.getVehicles();
    this.getTopPriceVehicles(50000);
    this.getBodyTypes();

    this.priceRange$ = this.homeservice.getPriceRanges();
  }

  getVehicles(vehicleCount?: number, bodyTypeId?: string) {
    this.homeservice.getVehicles(vehicleCount, bodyTypeId).subscribe({
      next: (response: { items: Vehicle[] }) => {
        this.vehicles = response.items;
      },
      error: (error) => {
        console.error(error);
      },
    }); }

  getTopPriceVehicles(LowerPriceLimit: number) {
    this.homeservice
      .getVehicles(undefined, undefined, LowerPriceLimit)
      .subscribe({
        next: (response: { items: Vehicle[] }) => {
          this.topPriceVehicles = response.items; }, error: (error) => {
console.error(error);
```

```
        },
      });
  }

  getBodyTypes() {
    const vehicleObservables: Observable<ApiResponse<Vehicle>>[] = [];

    this.homeservice.getBodyTypes().subscribe({
      next: (response: { items: BodyType[] }) => {
        this.bodyTypes = response.items;

        // group observables
        response.items.forEach((bodyType) => {
          vehicleObservables.push(
            this.homeservice.getVehicles(undefined, bodyType.id)
          );
        });

        // forkJoin to wait before all the observables will be executed
asyncroniously
        // and after that gives us all the results in the same moment.
        forkJoin(vehicleObservables).subscribe({
          next: (vehicleResponses: ApiResponse<Vehicle>[]) => {
            const filteredBodyTypesDataHolder: BodyType[] = [];
            vehicleResponses.forEach((response, index) => {
              if (response.items.length >= 2) {
                filteredBodyTypesDataHolder.push(this.bodyTypes[index]);
              }
            });

            this.filteredBodyTypes = [
              ...this.filteredBodyTypes,
              ...filteredBodyTypesDataHolder,
            ];
          },
          error: (error) => {
            console.error(error);
          },
        });
      },
      error: (error) => {
        console.error(error);  }, });
```

Продовження додатку А

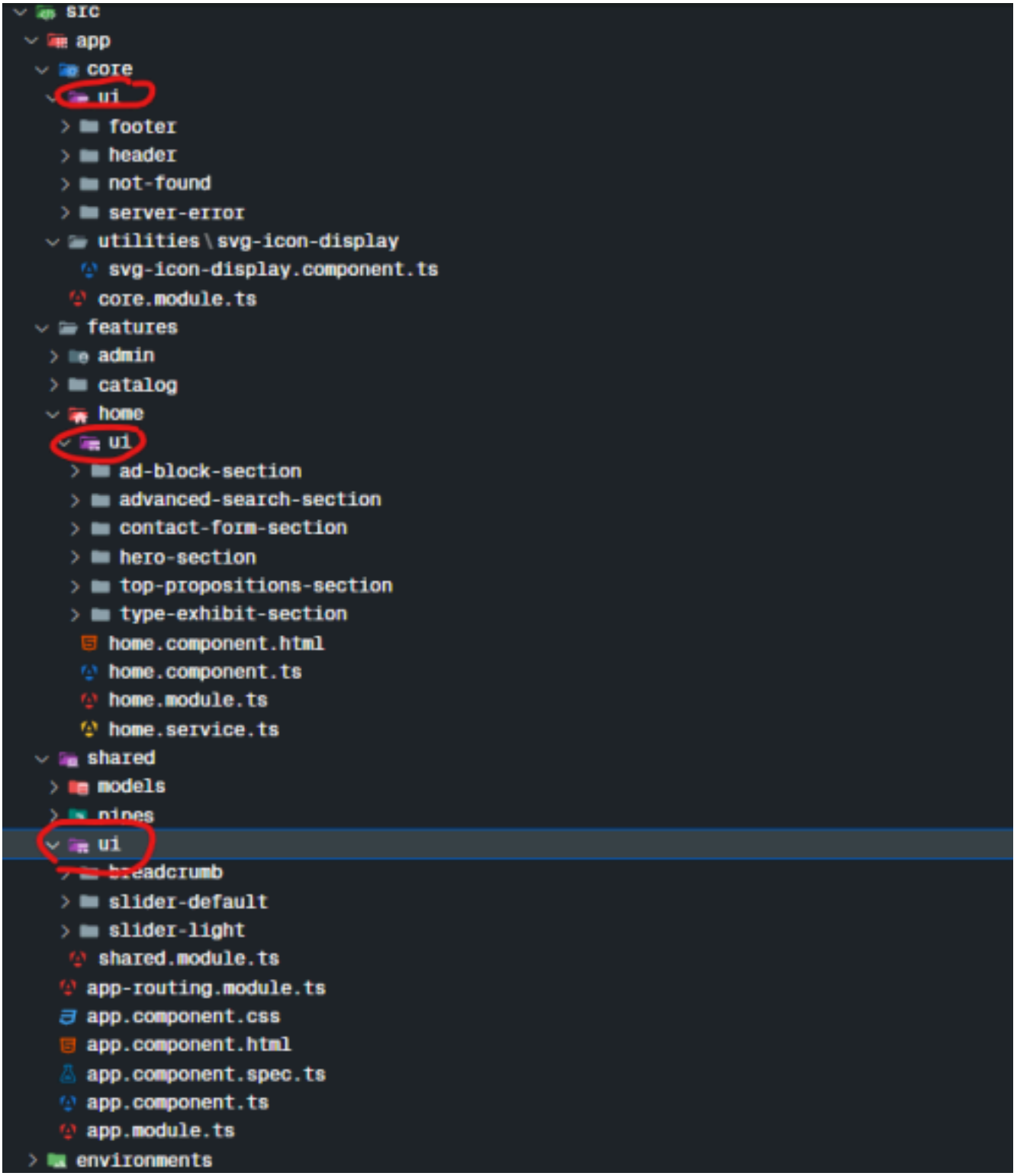
```
  }

  handleVehicleCardClick(vehicle: Vehicle) {
    this.router.navigate(['/catalog', vehicle.id]);
  }

  handleClickedBodyTypeEvent(bodyType: BodyType) {
    if (bodyType.id !== '0') {
      this.getVehicles(undefined, bodyType.id);
    } else {
      this.getVehicles();
    }
  }

  handleChangeRouterLink(searchAdvanced: SearchAdvancedParams): void {
    this.router.navigate(['/catalog'], { queryParams: { ...searchAdvanced }
});
  }
}
```

ДОДАТОК В



Додаток В. Розширене фото-приклад основи модульної структури власного проекту

ДОДАТОК С

```
/** @type {import('tailwindcss').Config} */
const COLORS = require("./colors.js");

module.exports = {
  content: ["./src/**/*.{html,ts}", "./node_modules/flowbite/**/*.js"],
  theme: {
    container: {
      center: true,
      padding: {
        DEFAULT: "16px",
        md: "32px",
        lg: "32px",
        "2xl": "120px",
      },
      screens: {
        sm: "640px",
        md: "768px",
        lg: "1024px",
        xl: "1280px",
        "2xl": "1440px",
      },
    },

    colors: {
      ...COLORS,
      content: {
        primary: COLORS.portGore[950],
        secondary: COLORS.portGore[700],
        tertiary: COLORS.portGore[400],
        disable: COLORS.portGore[200],
        informative: COLORS.kimberly[500],
        positive: COLORS.lima[500],
        error: COLORS.red[600],
      },
    },

    fontFamily: {
      code: ["Source Code Pro", "monospace"],
      ubuntu: ["Ubuntu", "sans-serif"],
      roboto: ["Roboto", "sans-serif"],
    },
```

Продовження додатку С

```
    fontSize: {
      // HEADING
      h1: ["42px", { lineHeight: "1.2", fontWeight: "500" }],
      h2: ["36px", { lineHeight: "1.2", fontWeight: "500" }],
      h3: ["32px", { lineHeight: "1.2", fontWeight: "500" }],
      h4: ["28px", { lineHeight: "1.2", fontWeight: "500" }],
      h5: ["24px", { lineHeight: "1.2", fontWeight: "500" }],
      h6: ["20px", { lineHeight: "1.2", fontWeight: "500" }],
      // BODY
      xl: ["20px", { lineHeight: "1.2" }],
      lg: ["18px", { lineHeight: "1.2" }],
      md: ["16px", { lineHeight: "1.2" }],
      sm: ["14px", { lineHeight: "1.2" }],
      xs: ["12px", { lineHeight: "1.2" }],
    },

    fontWeight: {
      bold: 700,
      medium: 500,
      light: 300,
    },

    lineHeight: {
      none: "1",
      normal: "1.2",
    },

    spacing: {
      "0": "0px", // 0px
      xs: "0.25rem", // 4px
      sm: "0.5rem", // 8px
      base: "1rem", // 16px
      md: "1.5rem", // 24px
      lg: "2rem", //32px
      xl: "2.5rem", // 40px
      "2xl": "4rem", // 64px
    },
    borderRadius: {
      none: "0px",
      sm: "2px",
      DEFAULT: "4px", md: "6px",
```

Продовження додатку С

```
      lg: "8px",
      xl: "12px",
      "2xl": "16px",
      "3xl": "24px",
      full: "9999px",
    },
  },
  plugins: [require("flowbite/plugin")],
};
```