Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**Розробка системи електронної комерції за допомогою Next.js (TypeScript) та Axum (Rust)**»

Виконав: студент 4-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Маслов Нікіта Євгенович

Керівник Корнійчук М.А.

спеціаліст компютерних наук, асистент

Рецензент ____________________

Кваліфікаційна робота захищена з оцінкою ____________________

Секретар ЕК _________________

«11» травня 2023 р.

Київ – 2023

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“10” жовтня 2022 року

ЗАВДАННЯ

ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Маслову Нікіті

1. Тема роботи «**Розробка системи електронної комерції за допомогою Next.js (TypeScript) та Axum (Rust)**», керівник роботи Корнійчук Максим Анатолійович, спеціаліст компютерних наук, асистент

2. Строк подання студентом роботи 20 травня 2023

3. План роботи

- Анотація
- Вступ
- Розділ 1. Дослідження та аналіз предметної області
    - 1.1. Інтерактивні застосунки та SPA
    - 1.2. Адаптивна, відгукова та гібридна верстка

    - 1.3. Підхід "mobile first"
    - 1.4. Особливості використання бібліотек компонентів
    - 1.5. Оптимізація зображень
    - 1.6. SEO в контексті SPA
    - 1.7. Локалізація
    - 1.8. REST API
    - 1.9. MVC
    - 1.10. Слабке зв’язування
    - 1.11. Документоорієнтовані бази даних
- Розділ 2. Проєктування та розробка системи
    - 2.1. Складові системи та використані технології
    - 2.2. Створення дизайну застосунку
    - 2.3. Створення моделей даних
    - 2.4. Визначення REST API інтерфейсу взаємодії клієнта та сервера
    - 2.5. Розбиття фронтенду на сторінки та компоненти
    - 2.6. Використання TypeScript для покращення найдійності застосунку
    - 2.7. Верстка та трюки вирівнювання елементів
    - 2.8. Інтеграція компонентів MUI
    - 2.9. Оптимізація зображень за допомогою компонента Image

    - 2.10. Використання різних видів рендерінгу для покращення SEO та швидкодії застосунку
    - 2.11. Використання local storage для збереження товарів у кошику між браузерними сесіями
    - 2.12. Реалізація віртуальної прокрутки для відображення великої кількості елементів різної висоти у списку
    - 2.13. Реалізація локалізації за допомогою бібліотеки next-i18next
    - 2.14. Реалізація REST API за допомогою фреймворку Axum (Rust)
    - 2.15. Відправка повідомлень про нове замовлення через Телеграм-бот
- Висновки
- Список використаних джерел

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	жовтень			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	жовтень — листопад			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	листопад			
4.	Написання розділів роботи	листопад — березень			
5.	Проміжний контроль виконання роботи	лютий			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	січень — березень			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)				
	Розділ 2 (аналітично-дослідницька частина)				
	Розділ 3 (проектно-рекомендаційна частина)				
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	квітень — початок травня			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	середина травня			
9.	Подання на зовнішню рецензію	середина травня			
10.	Підготовка до захисту кваліфікаційної роботи на засіданні кафедри: написання доповіді тавиготовлення ілюстративногоматеріалу	до ___ травня			
11.	Попередній захист кваліфікаційної роботи на засіданні кафедри	до ___ травня			
12.	Подання кваліфікаційної роботи на кафедру з усіма супроводжувальними документами	до ___ травня			
13..	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 10 жовтня 2022 р.

Науковий керівник Корнійчук Максим Анатолійович

Виконавець кваліфікаційної роботи Маслов Нікіта Євгенович

ЗМІСТ

Анотація **8**
Вступ **8**
Розділ 1. Дослідження та аналіз предметної області **9**
1.1. Інтерактивні застосунки та SPA 9
1.2. Адаптивна, відгукова та гібридна верстка 9
1.3. Підхід "mobile first" 10
1.4. Особливості використання бібліотек компонентів 12
1.5. Оптимізація зображень 13
1.6. SEO в контексті SPA 15
1.7. Локалізація 18
1.8. REST API 18
1.9. MVC 20
1.10. Слабке зв’язування 21
1.11. Документоорієнтовані бази даних 23
Розділ 2. Проєктування та розробка системи **25**
2.1. Складові системи та використані технології 25
2.2. Етапи розробки 27
2.3. Створення дизайну застосунку 28
2.4. Створення моделей даних 29
2.5. Визначення REST API інтерфейсу взаємодії клієнта та сервера 33
2.6 Структура фронтенд частини 35
2.6. Особливості верстки 36
Mobile First 36
Гібридна верстка 37
Міксин для медіа запитів 37
Трюки вирівнювання елементів 39
2.8. Інтеграція бібліотечних компонентів 41
2.9. Оптимізація зображень за допомогою компонента Image 42
2.10. Використання різних видів рендерингу для покращення SEO та швидкодії застосунку 44
2.11. Використання local storage для збереження товарів у кошику між браузерними сесіями 45

2.12. Реалізація віртуальної прокрутки для відображення великої кількості елементів різної висоти у списку 46
2.13. Реалізація локалізації за допомогою бібліотеки next-i18next 47
2.14. Реалізація REST API за допомогою фреймворку Axum (Rust) 55
2.15. Відправка повідомлень про нове замовлення через Телеграм-бот 59
Висновки **61**
Список використаних джерел **63**

Анотація

У данній роботі розглядаються особливості створення сучасного SPA на прикладі e-commerce застосунку. Розробка SPA ведеться за допомогою фреймворків Next.js (TypeScript) та Axum (Rust). Досліджено технології та підходи для вирішення таких задач: створення багатого користувацького досвіду; адаптованість під будь-які пристрої; SEO-оптимізація SPA; локалізація; збереження даних.

Вступ

Була поставлена задача розробити якісний та сучасний інтернет-магазин для поширення та продажу приладів альтернативної медицини.

Велика частина цільової аудиторії інтернет-магазину — дорослі та літні люди. Через це було важливо створити максимально інтуітивний інтерфейс з великою кількістю допоміжних елементів, які спрощують користувацькі дії на сайті, наприклад, автодоповнення, розумний пошук, текстові підказки і т.д.

Було важливо створити високу інтерактивність та плавність під час користування застосунком, тому було вирішено використовувати SPA (single-page application).

Щоб було легше натрапити на сайт у пошукових меражах, було вирішено зробити внутрішню SEO-оптимізацію.

Також була поставлена задача зробити інтернет-магазин інтернаціональним з можливістю зробити замовлення з-за кордону. Щоб люди з різних країн могли зрозуміти контент застосунку, було вирішено реалізувати переклад різними мовами (локалізацію).

Розділ 1. Дослідження та аналіз предметної області

1.1. Інтерактивні застосунки та SPA

Інтерактивний веб-застосунок — це застосунок, який здатний взаємодіяти з користувачем без необхідності постійного оновлення сторінки. Такі застосунки зазвичай використовують AJAX (Asynchronous JavaScript And XML) для виконання асинхронних запитів на сервер, що дозволяє отримувати та відправляти дані без необхідності повного оновлення сторінки. Це значно підвищує швидкість роботи застосунку та забезпечує кращий користувацький досвід.

Односторінковий застосунок (SPA) — це конкретний тип інтерактивних веб-застосунків, який працює в браузері користувача і не вимагає повного перезавантаження сторінки під час навігації. SPA використовує єдиний HTML-документ як каркас, а динамічне оновлення вмісту здійснюється за допомогою JavaScript. Основна мета SPA — забезпечити більш плавний та інтерактивний користувацький досвід, подібний до робочого середовища настільних застосунків.

1.2. Адаптивна, відгукова та гібридна верстка

В контексті веб-дизайну сучасні технології надають можливість розробляти сайти та веб-застосунки, які забезпечують оптимальний користувацький досвід, незалежно від пристрою, на якому вони відображаються. Для досягнення цієї мети використовуються різні методи верстки: адаптивна, відгукова та гібридна.

Адаптивна верстка (Adaptive Design) передбачає створення декількох макетів для різних розмірів екранів. Зазвичай визначаються шість варіантів макетів залежно від ширини екрану: 320, 480, 760, 960, 1200, 1600.

Веб-застосунок визначає тип пристрою користувача та відображає відповідний макет. Цей підхід забезпечує високий рівень контролю над дизайном на різних пристроях, але може вимагати більше розробницьких ресурсів, оскільки кожен макет потребує індивідуальної розробки та підтримки.

Відгукова верстка (Responsive Design) використовує гнучкі (або "резинові") сітки, зображення та CSS-стилі, які автоматично підлаштовуються під розмір екрану користувача. Гнучкість досягається завдяки комбінації використання відносних одиниць, наприклад, відсотків та технологій flexbox та grid. При використанні відгокового підходу одним макетом можна охопити широкий спектр розмірів екранів, що робить цей підхід більш гнучким і економічним з точки зору розробки. Однак, це також може призвести до складностей у контролі дизайну, оскільки елементи автоматично підлаштовуються під різні пристрої, що може призвести до непередбачуваних візуальних результатів.

Гібридна верстка представляє собою поєднання адаптивного та відгукового підходів. Таке міксування дозволяє використовувати корисні риси обох методів верстки для створення найкращого результату. Прикладом гібридного підходу може бути ситуація, коли створються 3 основні макети: для мобільних, планшетних та десктопних розмірів екрану — а для підлаштовування вигляду всіх проміжних розмірів та окремих елементів використовуються медіа запити, технології flexbox, grid та відносні одиниці.

1.3. Підхід "mobile first"

Підхід "Mobile First" — це стратегія проектування та розробки веб-сайтів, при якій спочатку створюється версія для мобільних пристроїв, а потім робиться адаптація для більших екранів. Ця концепція є революційною

зміною парадигми, оскільки традиційно спочатку створювали десктопні версії веб-сайтів, а потім адаптували їх для мобільних пристроїв.

Підхід "Mobile First" виник у відповідь на швидке зростання кількості користувачів мобільних пристроїв. Згідно з даними статистики, більшість користувачів інтернету зараз використовують мобільні пристрої для доступу до вебу. Виходячи з цього, підхід "Mobile First" був призначений для покращення користувацького досвіду на мобільних пристроях.

Цей підхід має ряд переваг. По-перше, оскільки мобільні пристрої мають обмежені технічні характеристики, наприклад, розмір екрану, швидкість процесора, "Mobile First" стимулює дизайнерів та розробників до створення більш ефективних, простих та швидких сайтів. По-друге, концептуально та морально дизайнерам та розробникам легше почати с мінімальної версії для мобільних пристроїв та додавати функції та елементи по мірі зростання розміру екрану, аніж почати с максимальної версії та з болем на серці думати, що прибирати та урізати.

Часто підхід "Mobile First" реалізується за допомогою CSS media queries, коли спочатку створюється мобільна версія сайту, а потім за допомогою таких властивостей, як `@media screen and (min-width: [value])` виконується адаптація під пристрої з більшим розміром екрану.

Підхід "Mobile First" також має значний вплив на SEO (Search Engine Optimization). Google, який є найбільшою пошуковою системою в світі, ввів індексування "Mobile First" у 2018 році. Це означає, що Google спочатку оцінює мобільну версію веб-сайту при визначенні його рангу в пошукових видачах.

Таким чином, якщо веб-сайт надає відмінний користувацький досвід на мобільних пристроях, це може позитивно вплинути на його SEO

ранжування. На відміну від цього, якщо веб-сайт працює погано на мобільних пристроях, це може негативно вплинути на його видимість у пошукових видачах. Відтак, підхід "Mobile First" є не лише стратегією дизайну та розробки, але й важливим фактором SEO.

1.4. Особливості використання бібліотек компонентів

Часто, якщо немає серйозних вимог до дизайну або є обмежений час на створення веб-сайту, розробники прибігають до використання бібліотек готових компонентів, таких як Bootstrap, MUI, ChakraUI і т.д.

Такі бібліотеки й справді спрощують та пришвидшують розробку, проте важливо пам'ятати й про певні недоліки таких інструментів. Попри те, що бібліотеки компонентів надають непогану можливість кастомізації, часто буває досить важко та часозатратно адаптувати ці компоненти до своїх потреб. Причина тому — складність влаштування готових компонентів.

Розробники бібліотек компонентів намагаються створити максимально універсальні компоненти, тому майже завжди ці компоненти мають складну структуру та багато складових. Тому для кастомізації доводиться спочатку витратити немало часу, щоб розібратися з внутрішнім влаштуванням цих компонентами, а потім змінити певні частини так, щоб не зламалися інші.

Інша проблема також слідує зі спроби зробити готові компоненти максимально універсальними. Окрім функцій, які потрібні для певного веб-сайту, бібліотечні компоненти можуть мати ще купу всього, що ніколи не буде використано цим сайтом. Така комора додаткових функцій часто призводить до того, що компоненти стають дуже важкими та повільними. Як результат — погіршення користувацького досвіду.

1.5. Оптимізація зображень

Оптимізація зображень — це процес зменшення розміру файлу зображення без втрати якості до прийнятного рівня. Ця процедура є критично важливою в контексті веб-розробки, оскільки великі файли зображень можуть суттєво сповільнити завантаження сторінки, негативно впливаючи на користувацький досвід і позиціонування в пошукових системах.

Один із способів оптимізації зображень — це стиснення зображення за допомогою спеціальних сервісів, таких як TinyPNG, TinyJPG, ImageOptim, які зменшують обсяг файлу, не впливаючи на фізичні розміри зображення. Тут слід бути обережним, оскільки надмірне стиснення може призвести до втрати якості зображення.

Вибір правильного формату зображення також є важливим аспектом оптимізації зображень, оскільки різні формати мають різні характеристики і можуть краще або гірше підходити для конкретних сценаріїв використання.

1. *JPEG (Joint Photographic Experts Group):* Цей формат зазвичай використовується для фотографій або ілюстрацій зі складними кольорами. JPEG підтримує мільйони кольорів, що робить його вибором номер один для високоякісних, деталізованих зображень. Однак JPEG використовує втратне стиснення, що означає, що деякі дані втрачаються при стисненні, що може призвести до погіршення якості при високому ступені стиснення.

2. *PNG (Portable Network Graphics):* PNG — це формат, який використовує безвтратне стиснення, тому якість зображення не постраждає від стиснення. PNG також підтримує прозорість, що

робить його відмінним вибором для логотипів, значків та інших зображень, яким потрібна прозора основа. Однак PNG-файли мають тенденцію бути більшими за розміром в порівнянні з JPEG.

3. *GIF (Graphics Interchange Format):* GIF формат, як правило, використовується для анімацій, але він підтримує лише 256 кольорів, тому він не підходить для високоякісних зображень. GIF також підтримує прозорість.

4. *WebP:* WebP — це сучасний формат зображень, розроблений Google, який забезпечує краще стиснення і якість, ніж JPEG і PNG. Він підтримує як втратне, так і безвтратне стиснення, а також прозорість і анімацію. Однак, не всі браузери є сумісними з форматом WebP, що може обмежувати його загальне використання.

5. *SVG (Scalable Vector Graphics):* SVG є форматом векторних зображень, що робить його ідеальним для логотипів, іконок, діаграм і будь-яких інших зображень, які мають бути чіткими при будь-якому масштабуванні. SVG-файли, як правило, мають невеликий розмір файлу і підтримують прозорість і анімацію.

У зв'язку з тим, що кожен формат зображення має свої власні характеристики та переваги, вибір найбільш підходящого формату вимагає врахування типу зображення, яке планується використовувати, а також того, як планується використовувати це зображення.

Наприклад, якщо потрібне зображення високої якості для головної сторінки веб-сайту, можливо, буде краще використовувати формат JPEG, щоб отримати найкращу можливу якість при прийнятному рівні стиснення. Якщо потрібен прозорий логотип для веб-сайту, формат PNG або SVG

може бути найкращим вибором. Якщо потрібна анімація, вибір GIF або WebP може бути найкращим рішенням.

Важливо зазначити, що під час вибору формату зображення потрібно також враховувати підтримку браузерів. Хоча більшість сучасних браузерів сумісні з усіма цими формати, деякі старіші версії браузерів можуть не підтримувати новіші формати, такі як WebP.

Використання оптимізованих зображень не тільки поліпшує час завантаження сторінки, але і покращує загальний користувацький досвід, що в свою чергу позитивно впливає на SEO. Пошукові системи, такі як Google, враховують швидкість завантаження сторінки при ранжуванні сайтів, тому оптимізація зображень може допомогти покращити SEO-рейтинг.

1.6. SEO в контексті SPA

Пошукова оптимізація (SEO) є критично важливою складовою веб-розробки, оскільки вона забезпечує видимість веб-сайту в пошукових системах, таких як Google. Вона включає в себе ряд методів і практик, спрямованих на покращення ранжування сайту в результатах пошуку, що в свою чергу збільшує органічний (неоплачуваний) трафік на сайт.

Однак, коли йдеться мова про односторінкові додатки (SPA), SEO стає особливим викликом. Структура SPA вимагає відмінних підходів до SEO, порівняно з традиційними веб-сайтами.

SPA використовують JavaScript для рендерингу контенту на клієнтській стороні, що може створити проблеми для пошукових систем. Історично, багато пошукових роботів мали проблеми з індексуванням веб-сайтів, які важко залежали від JavaScript. Це призводило до того, що вміст SPA не був відображений або індексований правильно, що погіршувало рейтинг SEO.

Однак, з часом пошукові системи, зокрема Google, значно покращили свою здатність обробляти JavaScript. Проте навіть із цими покращеннями, на даний момент все ще важливо розробляти SPA з урахуванням незрілості підтримки JavaScript при індексації сайтів.

Один з підходів для оптимізації SPA полягає в використанні техніки, відомої як Server Side Rendering (SSR). Суть SSR полягає в тому, що вміст SPA генерується на сервері перед відправкою на клієнтський браузер. Це означає, що пошукові роботи можуть індексувати вміст, як це відбувається з традиціційними веб-сайтами, що забезпечує більшу видимість для SPA.

Окрім SSR, існує техніка, відома як Pre-rendering, що передбачає генерацію статичних HTML-сторінок для кожного маршруту або шляху у SPA. Такі статичні сторінки можуть бути заздалегідь згенеровані і відправлені до пошукових роботів, коли ті сканують сайт. Як і з SSR, такий підхід дозволяє пошуковим роботам правильно індексувати вміст SPA. У порівнянні з SSR при використанні Pre-rendering сторінки відображются швидше, бо коли користувач відкриває сторінку, сервер відразу відправляє заздалегідь згенерований HTML.

Важливо зауважити, що крім технічних аспектів SEO, таких як SSR та Pre-rendering, існують і інші фактори, які важливі для пошукової оптимізації SPA. До них входять:

- *Структура сайту і навігація.* Важливо, щоб структура SPA була логічною і зрозумілою для користувачів і пошукових роботів. Пошукові роботи використовують структуру сайту для розуміння його контенту і контексту.
- *Метатеги.* Метатеги, такі як title, description, alt, keywords, важливі для пошукових систем. Вони допомагають пошуковим роботам

розуміти контент на сторінці і пов'язувати його з відповідними запитами пошуку.

- *Schema markup.* Schema markup є формою мікророзмітки, яка допомагає пошуковим роботам краще розуміти контент на сторінці. Вона може включати деталі про автора, дату публікації, відгуки та іншу інформацію.
- *Завантаження сторінки та продуктивність.* Швидкість завантаження сторінки важлива не тільки для користувацького досвіду, але й для пошукової оптимізації. Пошукові системи враховують швидкість завантаження сторінки при визначенні ранжування сайту. Чим швидше завантажується сторінка, тим краще для SEO.

Технічні інструменти, які можуть допомогти в оптимізації SEO для SPA, включають:

- *Node.js.* Це середовище JavaScript для сервера, яке часто використовується для SSR в SPA.
- *Next.js і Nuxt.js.* Це фреймворки, базовані на React і Vue відповідно, які вбудовують підтримку SSR та Pre-rendering.
- *Prerender.io і Rendertron.* Це сервіси, які використовуються для Pre-rendering SPA.
- *Google Search Console.* Цей інструмент від Google дозволяє перевіряти індексацію сайту, переглядати статистику пошукового трафіку і розробляти SEO.

Використання цих інструментів та методів, разом з розумінням особливостей SPA, може допомогти розробникам створити SPA, які не

тільки привабливі для користувачів, але й оптимізовані для пошукових систем.

1.7. Локалізація

Локалізація — це процес адаптації продукту до мови та культури окремого географічного регіону або ринку. Переважно ця адаптація досягається шляхом перекладу контенту застосунку на різні мови.

У контексті SPA локалізацію зручно реалізовувати за допомогою спеціальних бібліотек, таких як i18n. Ці бібліотеки дозволяють в окремому місці визначати текстові блоки та переклад для них різними мовами. А потім, коли треба у верстці вставити текст, то вставляється не конкретний переклад, а виклик спеціальної бібліотечної функції з ідентифікатором текстового блоку. У результаті виклику цієї функції на цьому ж місці з'являється конкретний варіант перекладу в залежності від поточної вибраної локалі.

1.8. REST API

REST (Representational State Transfer) — це архітектурний стиль розробки мережевих додатків, який впроваджується у HTTP протоколі. Він є домінуючим стандартом для створення веб-API (Application Programming Interface), що дозволяє веб-додаткам взаємодіяти один з одним.

REST API використовує HTTP методи GET, POST, PUT, DELETE та інші, щоб дозволити взаємодію між клієнтською та серверною частиною веб-додатків. Наприклад, GET метод використовується для отримання даних, POST — для створення нових даних, PUT — для оновлення існуючих даних, а DELETE - для видалення даних.

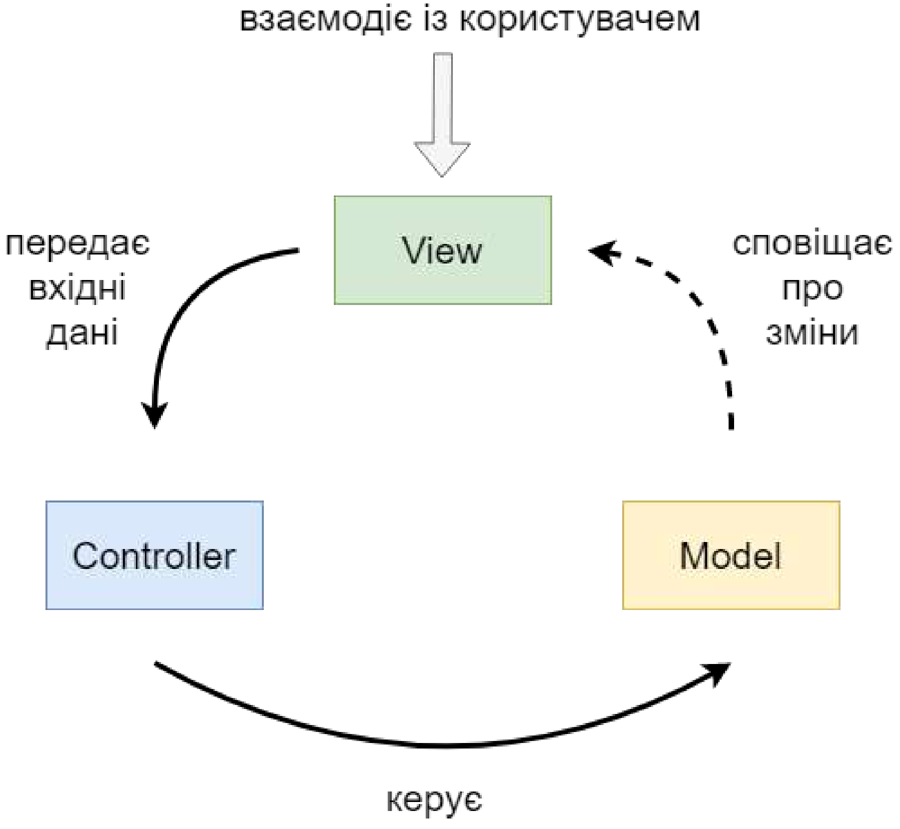
Важливим принципом REST є безстановість (statelessness). Тобто кожен запит від клієнта до сервера містить всю необхідну інформацію для

обробки запиту. Сервер не зберігає жодного стану клієнта між запитами, що забезпечує високу масштабованість та надійність системи.

Ще одним принципом REST є ресурсно-орієнтований дизайн. В REST API, "ресурс" — це будь-який об'єкт або сутність, з якою може взаємодіяти клієнт. Кожен ресурс має унікальний ідентифікатор (URI), за допомогою якого клієнт може досягти його.

RESTful API — це API, що повністю дотримується принципів REST. Воно використовуються в багатьох веб-додатках, включаючи односторінкові додатки (SPA), для забезпечення простого, зрозумілого та стандартизованого способу взаємодії між клієнтською та серверною сторонами. Проте варто зазначити, що на практиці часто при створенні API беруть до уваги тільки основні принципи REST, а іншими нехтують або реалізують лише частково.

1.9. MVC



MVC — це архітектурний шаблон проектування, який ділить застосунок на три основні компоненти: модель (Model), вид (View) та контролер (Controller).

Модель відповідає за представлення даних та бізнес-логіки. Вона є відображенням даних, які працюють у застосунку, та містить функції для доступу, обробки та зберігання цих даних. В контексті веб-додатку, модель зазвичай відповідає за взаємодію з базою даних.

Вид відповідає за візуальне представлення даних, тобто те, що бачить користувач. Він відображає дані, отримані з моделі, та генерує відповідний вихідний формат, наприклад, HTML.

Контролер служить для зв'язку між моделлю та видом. Він обробляє вхідні запити від користувача,, змінює модель за потреби, та оновлює вигляд.

Мета шаблону — гнучкий дизайн програмного забезпечення, який повинен полегшувати подальші зміни чи розширення програм, а також надавати можливість повторного використання окремих компонентів програми. Крім того використання цього шаблону у великих системах сприяє впорядкованості їхньої структури і робить їх більш зрозумілими за рахунок зменшення складності.

MVC може бути застосований і при створенні REST API, проте в контексті REST API буде відсутня частина V (вид) на серверній стороні, оскільки представлення даних відбувається на клієнті. Тому у цій ситуації шаблон MVC скорочується і може бути названий MC. При цьому модель та контролер виконують наступні функції:

- *Модель (Model).* Модель представляє доменні дані та бізнес-логіку, яка керує цими даними. У контексті REST API, модель відповідає за взаємодію з базою даних, забезпечуючи методи для збереження, видалення, оновлення та отримання даних.

- *Контролер (Controller).* Контролер у ролі "директора" керує потоком даних між моделлю та клієнтом. Він обробляє вхідні запити, здійснює відповідні дії в моделі та формує відповідь.

1.10. Слабке зв’язування

Слабке зв'язування (Loose Coupling) — це підхід до проєктування програмних систем, який максимізує незалежність і гнучкість компонентів системи. Ідея слабкого зв'язування полягає в тому, що різні частини системи повинні взаємодіяти одна з одною мінімально необхідно, так щоб зміни в одному компоненті мали мінімальний вплив на інші.

Одним із головних інструментів досягення слабкого зв’язування є інтерфейси. Інтерфейс визначає контракт між двома компонентами, який

не накладає жодних обмежень на внутрішню реалізацію цих компонентів. Тобто можливо замість указання конретної реалізації просто казати, що певний елемент повинен бути таким, що реалізовує визначений інтерфейс. Такий підхід дозволяє звести майже до нуля кількість місць, де у залежностях якогось елементу системи вказується конкретна реалізація іншого елементу системи. Це й створює слабку зв’язаність. Проте, як вже було сказано раніше, використання інтерфейсів зводить **майже** до нуля кількість конкретнизованих залежностей, бо все ж на якомусь етапі потрібно вказати реальну реалізацію інтерфейсу. Але це частіше всього робиться на верхніх рівнях програмної системи і тільки один раз, що не створює такого тісного зв’язку.

У процесі реалізації слабкого зв’язування отримуються значні бонуси:

- *Сприяння модульності.* Слабке зв'язування сприяє розбиттю програми на окремі модулі, які можна розробляти, тестувати, оновлювати та виправляти незалежно один від одного.
- *Легкість тестування.* У компонентах зі слабким зв’язуванням для всіх залежностей при тестуванні можна просто вказати фейкові реалізації інтерфейсів (mocks). Це дозволяє тестувати компоненти системи ізольовано один від одного. У свою чергу ізольоване тестування дозволяє швидше виявити джерело проблеми, бо при виконанні тестів завжди видно, в якому конкретному компоненті щось працює не так, як повинно. Якби при тестуванні компонентів використовувалися справжні залежності, то було б вже набагато складніше визначити джерело проблеми. Коли падають тести в одному компоненті, тести падають і в усіх інших залежних від нього компонентах. У результаті замість конретного проблемного місця відображається багато місць, де виявлена проблема.

- *Підтримка розширюваності.* Коли компоненти слабко пов'язані, можна легко змінювати реалізації окремих комопнентів та додавати нові функції, не вносячи значних змін в існуючий код.

1.11. Документоорієнтовані бази даних

Документоорієнтовані бази даних є одним із підтипів NoSQL (Not Only SQL) баз даних, що призначені для зберігання, отримання та обробки даних у вигляді неструктурованих документів. Найбільш відомі представники таких баз даних – це MongoDB та CouchDB.

У центрі документоорієнтованої моделі даних знаходиться концепція "документу". В контексті цих баз даних, документ — це більш гнучка і самодостатня структура, ніж традиційний рядок в таблиці реляційної бази даних. Документи містять в собі всю необхідну інформацію про об'єкт, і зберігають дані у форматах, які легко читаються людиною, таких як JSON або XML.

Однією з ключових переваг документоорієнтованих баз даних є гнучкість моделі даних. Вони не вимагають попереднього визначення схеми та дозволяють змінювати структуру даних "на льоту", в залежності від потреб додатку. Це робить їх особливо привабливими для використання в ситуаціях, коли вимоги до даних можуть часто змінюватися.

Другою важливою перевагою є горизонтальна масштабованість, що дозволяє легко розширювати ємність бази даних, додаючи нові вузли до кластера.

Проте важливо зазначити, що документоорієнтовані бази даних не є панацеєю і мають свої обмеження. Зокрема, вони можуть бути менш ефективними, ніж реляційні бази даних для складних зв'язків між даними або для виконання складних запитів.

Також не можна нехтувати CAP теоремою, яка стверджує, що в будь-якій розподіленій системі, такій як база даних, можна досягти тільки двох з трьох властивостей: консистентності, доступності та стійкості до розподілу.

Документоорієнтовані бази даних зазвичай оптимізуються для доступності та стійкості до розподілу, жертвуючи консистентністю. Це означає, що вони дозволяють тимчасові невідповідності між різними вузлами в розподіленій системі. Прикладом може бути ситуація, коли два користувача одночасно звертаються до одного й того ж запису в базі даних. Користувач A оновлює запис, і оновлення відображається в одному вузлі. Однак через певну затримку в мережі це оновлення може не встигнути пройти до вузла, до якого звертається користувач B. У результаті користувач B бачить стару версію запису, що порушує принцип консистентності.

Для деяких систем жертва консистентністю може бути прийнятним компромісом, а для інших — ні. Тому, як завжди, приймати рішення про використання певного типу бази даних треба спираючись на вимоги конкретної системи, яка розробляється.

Розділ 2. Проєктування та розробка системи

2.1. Складові системи та використані технології

Для реалізації e-commerce застосунку було використано наступні технології:

- **Дизайн:**
  - Figma для створення макетів;
  - Photoshop для роботи з фотографіями;
- **Фронтенд:**
  - Next.js. Вибрано з декількох причин: через потужні можливості SEO-оптимізації за рахунок використання SSR та SSG; вбудовані можливості автоматичної оптимізації зображень; зручний механізм маршрутизації на основі файлової системи;
  - TypeScript. Використання більш строгої типізації дозволяє покращити надійність та підтримуваність застосунку;
  - SASS modules. Система модулів чудово підходить до компонентної React архітектури, бо кожен компонент буде мати у рамках свого модуля свої стилі, які не можуть перетинатися зі стилями інших компонентів. Також препроцесор SASS може бути корисний для створення міксинів та за рахунок зручних вбудованих функцій, наприклад, darken та lighten;
  - MUI. Бібліотека React компонентів для спрощення життя в ситуаціях з більш складними компонентами;
- **Бекенд:**
  - Rust. Була використана з декількох причин: особисте вподобання; статичну типізацію, що забезпечує кращу

надійність та підтримуваність; ефективність використання ресурсів, що дозволить знизити вартість серверів;
   - MongoDB. Була використана документоорієнтована NoSQL база даних, бо у поточному застосунку використовується невелика кількість сутностей з невеликою кількістю зв’язків. Також сутності містять вкладені структури даних, які можуть мати різні типи у різних екземлярів;
   - MVC. Було використано цей архітектурних шаблон, щоб краще структурувати серверну частину застосунку та розподіли обов’язки між різними частинами системи;
- **Комунікація.** REST API;
- **Авторизація.** На поточному етапі було вирішено поки що не реалізовувати авторизацію, бо наразі її єдина користь для користувача — пришвидшення процесу оформлення замовлення, бо система запам’ятовує дані про користувача, які треба заповнювати. Користь цієї економії часу зводиться майже до нуля в контексті поточного інтернет-магазину приладів альтернативної медицини, бо продукт позіціонується як “один прилад на все життя”. Тобто повторні замовлення від однієї й тієї ж людини досить рідке явище, саме тому користь від автозаповнення даних при оформленні замовлення невелика.
- **Локалізація.** Бібліотека next-i18next;
- **Зберігання зображень.** Усі зображення було вирішено зберігати на стороні Next.js на фронтенд, бо кількість зображень невелика, не планується якесь значне збільшення та кількість зображень не залежить від користувачів;
- **Контроль версій.** Git, Github.

2.2. Етапи розробки

1. Створення дизайну застосунку.
2. Визначення моделей даних.
3. Створення REST API інтерфейсу взаємодії клієнта та сервера.
4. Реалізація фронтенд частини.
5. Реалізація бекенд частини.
6. Поєднання фронтенд та бекенд.
7. Тестування та зневадження.

2.3. Створення дизайну застосунку

Етап розробки дизайну застосунку включив в себе наступні задачі:

- створення структури сайту,
- підготовка текстових матеріалів,
- фотозйомка приладів,
- робота над фотографіями у Photoshop,
- розробка макетів сторінок у Figma.

У результаті цього етапу було створено детальні макети сторінок застосунку у трьох версіях: мобільна, планшетна та десктопна:

2.4. Створення моделей даних

```rust
pub type ProductName = String;
pub type ProductPrice = f64;

pub type LocalizedProductDescr = String;

#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "camelCase")]
pub struct ProductDescr {
    pub uk: LocalizedProductDescr,
    pub en: LocalizedProductDescr,
    pub ru: LocalizedProductDescr,
}

pub type ImgPath = String;
pub type ImgWidth = u64;
pub type ImgHeight = u64;

#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "camelCase")]
pub struct ProductImg {
    pub path: ImgPath,
    pub width: ImgWidth,
    pub height: ImgHeight,
}

pub type LocalizedDiscountInfo = String;

#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "camelCase")]
pub struct DiscountInfo {
    pub uk: LocalizedDiscountInfo,
    pub en: LocalizedDiscountInfo,
    pub ru: LocalizedDiscountInfo,
}

#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "camelCase")]
pub struct Product {
    #[serde(rename = "_id", skip_serializing_if = "Option::is_none")]
    pub id: Option<ObjectId>,
    pub name: ProductName,
    pub descr: ProductDescr,
    pub img: ProductImg,
    pub price: ProductPrice,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub discount_info: Option<DiscountInfo>,
}
```

```rust
pub type Country = String;
pub type City = String;
pub type Address = String;
pub type Zip = String;
pub type Warehouse = String;

#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct NovaposhtaDeliveryKind {
    pub city: City,
    pub warehouse: Warehouse,
}

#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct CourierDeliveryKing {
    pub city: City,
    pub address: Address,
    pub zip: Zip,
}

#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct WorldwideDeliveryKing {
    pub country: Country,
    pub city: City,
    pub address: Address,
    pub zip: Zip,
}

#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "camelCase", tag = "kind", content = "data")]
pub enum Delivery {
    Novaposhta(NovaposhtaDeliveryKind),
    Courier(CourierDeliveryKing),
    Worldwide(WorldwideDeliveryKing),
}

pub type OrderItemQuantity = u64;

#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "camelCase")]
pub struct OrderItem {
    pub product_id: String,
    pub quantity: OrderItemQuantity,
}
```

```rust
#[derive(Serialize, Deserialize, Debug, Clone)]
#[serde(rename_all = "camelCase")]
pub struct Order {
    #[serde(rename = "_id", skip_serializing_if = "Option::is_none")]
    pub id: Option<ObjectId>,
    #[serde(skip_serializing_if = "Option::is_none")]
    pub creation_time: Option<DateTime<Utc>>,
    pub first_name: String,
    pub last_name: String,
    pub email: String,
    pub phone_number: String,
    pub delivery: Delivery,
    pub ordered_products: Vec<OrderItem>,
}
```

У результаті цього етапу було створено моделі даних для двох сутностей: Product (продукт) та Order (замовлення).

Модель продукту (Product) містить наступні дані:

- id (унікальний ідентифікатор продукту);
- name (назва продукту);
- descr (опис продукту);
- img (зображення). Складається з декількох полів. Поля width та height необхідно задавати окремо явно, бо цього вимагає Next.js для оптимізації зображень:
  - path (відносний шлях до файлу зображення);
  - width (ширина зображення);
  - height (висота зображення);
- price (вартість продукту);
- discount_info (поточна можлива знижка на продукт).

Модель замовлення (Order) містить такі дані:

- id (унікальний ідентифікатор замовлення);
- creation_time (час створення замовлення);
- first_name (ім’я замовника);
- last_name (прізвище замовника);
- email (електронна пошта замовника);
- phone_number (номер телефона замовника);
- delivery (варіант доставки). Складений атрибут різних типів:
    - novaposhta. Варіант доставки Україною компанією “Нова пошта”. Має такі поля:
        - city (місто, куди доставляється товар);
        - warehouse (відділення цього міста);
    - courier. Варіант кур’єрської доставки Україною за вказаною адресою. Має наступні поля:
        - city (місто, куди доставляється товар);
        - address (адреса доставки);
        - zip (поштовий індекс);
    - worldwide. Варінт доставки світом. Має такі поля:
        - country (країна, куди доставляється товар);
        - city (місто, куди доставляється товар);
        - address (адреса доставки);
        - zip (поштовий індекс);
- ordered_products (список замовлених товарів). Це масив елементів типу OrderItem, які мають наступну структуру:
    - product_id (посилання на замовлений товар);
    - quantity (кількість замовлених товарів цього типу).

2.5. Визначення REST API інтерфейсу взаємодії клієнта та сервера

Для взаємодії клієнтської та серверної частин було визначено такий інтерфейс:

- Створення замовлення:
  - POST /orders
  - Приклад тіла запиту у форматі JSON:


```json
{
    "firstName": "Нікіта",
    "lastName": "Маслов",
    "email": "maslov.n.e@gmail.com",
    "phoneNumber": "+380 (67) 538 47 98",
    "delivery": {
        "kind": "worldwide",
        "data": {
            "country": "Poland",
            "city": "Варшава",
            "address": "варшавская 97",
            "zip": "98789"
        }
    },
    "orderedProducts": [
        {
            "productId": "645f6f60acd5793eddfa8ae9",
            "quantity": 5
        },
        {
            "productId": "645f705facd5793eddfa8aec",
            "quantity": 3
        }
    ]
}
```


- Отримання інформації про продукт за унікальним ідентифікатором (id):
  - GET /products/:product_id

- Приклад тіла відповіді у форматі JSON:


```
{
    "_id": {
        "$oid": "645f6f60acd5793eddfa8ae9"
    },
    "name": "KGS",
    "descr": {
        "uk": "Середній за вартістю варіант.",
        "en": "An average cost option.",
        "ru": "Средний по стоимости вариант."
    },
    "img": {
        "path": "/images/kgs-kit.jpg",
        "width": 3436,
        "height": 1730
    },
    "price": 4900.0
},
```


- Отримання всіх існуючих продуктів:
  - GET /products
  - Приклад тіла відповіді у форматі JSON:


```
{
    "products": [
        {
            "_id": {
                "$oid": "645f6f60acd5793eddfa8ae9"
            },
            "name": "KGS",
            "descr": {
                "uk": "Середній за вартістю варіант.",
                "en": "An average cost option.",
                "ru": "Средний по стоимости вариант."
            },
            "img": {
                "path": "/images/kgs-kit.jpg",
                "width": 3436,
                "height": 1730
            },
            "price": 4900.0
        },
        {
            "_id": {
                "$oid": "645f705facd5793eddfa8aec"
            },
            "name": "KGS-MINI",
```

2.6 Структура фронтенд частини

Увесь фронтенд було розбито на частини (папки), кожна з яких відповідає за збереження елементів певної логічної групи:

- public — папка для збереження усіх статичних файлів, про яку знає та яку обслуговує Next.js
    - images — усі зображення застосунку
    - locales — усі файли з контентом для різних локалей
        - en — файли з контентом англійською
            - common.json — контент англійською, спільний для всіх сторінок
            - home.json — контент англійською для головної сторінки
        - uk — файли з контентом українською
            - common.json — контент українською, спільний для всіх сторінок
            - home.json — контент українською для головної сторінки
        - ru — файли з контентом російською
            - common.json — контент російською, спільний для всіх сторінок
            - home.json — контент російською для головної сторінки
- src — увесь вихідний код застосунку
    - api — увесь код для доступу до сторонніх API
    - components — усі компоненти застосунку
        - elements — усі базові будівельні блоки, такі як, кнопки, поля вводу і т.д.

    - modules — усі компоненти, які більше, ніж базові блоки, наприклад секції header або footer. Ці компоненти часто складаються з різних елементів (elements)
    - templates — усі шаблони сторінок, які повинні бути викликані з файлів-маршрутів у Next.js папці pages
    - layouts — усі макети, які використовуються, щоб обгортати шаблони та додавати до них певні компоненти, які будуть присутні за замовчуванням. Наприклад, у DefaultLayout можна включити компоненти Header та Footer
  - constants — місце для усіх глобальних констант
  - context — усі файли для керування контекстами при використанні React Context API
  - hooks — усі кастомні хуки
  - styles — усі глобальні стилі, які є в застосунку, наприклад, типографіка, змінні, кольори, міксини й т.д.
  - types — усі TypeScript типи, які не належать до якогось конкретного компонента
  - utils — усі допоміжні функції

2.6. Особливості верстки

Mobile First

Усі сторінки застосунку були зверстані з використанням підходу “Mobile First”. Концептуально це розумне рішення, бо морально всім легше починати з мінімальної версії для мобільних пристроїв та додавати нові елементи та нові функції зі збільшенням розміру девайсу, аніж починати з максимальної версії та думати, що видаляти, зменшувати та урізати. До

того ж підхід “Mobile First” дає кращі результати для SEO, як вже було сказано раніше.

Гібридна верстка

У поточному застосунку було використано методологію гібридної верстки, яка поєднує елементи адаптивної (adaptive) та відгукової (responsive) верстки. З адаптивної був взят підхід, який заключається в створенні декількох макетів для певних розмірів екрану. Було створено 3 макети: для мобільної версії, планшетної та десктопної. Для кожної з цих версій було визначено точку перелому, коли відбувається серйозна перебудова елементів. А для всіх проміжних розмірів було реалізовано автоматичне підлаштування зовнішнього вигляду завдяки використанню технологій гнучких сіток, таких як flexbox та grid, та відносних одиниць вимірювання, таких як %, vw, vh. Автоматичне підлаштування компонентів під проміжні розміри — це елемент відгукової верстки.

Міксин для медіа запитів

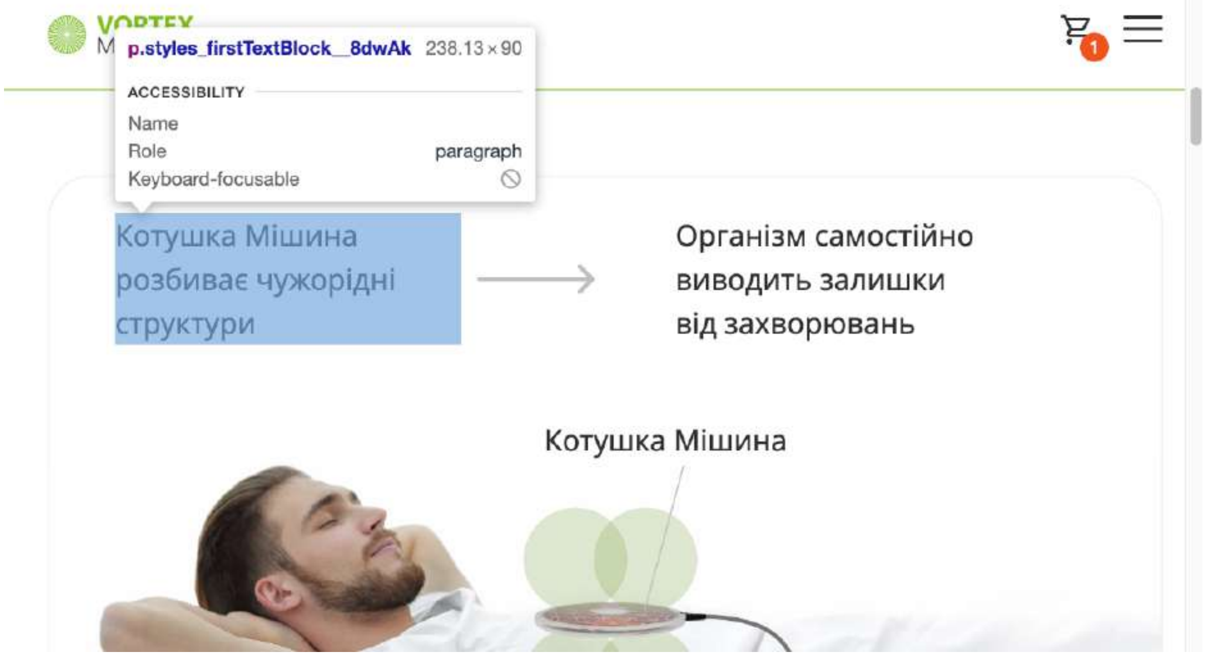
Для спрощення написання медіа запитів для основних контрольних точок було реалізовано спеціальний міксин:

```scss
@mixin for-size($size) {
  @if $size == content-min-width {
    @media screen and (min-width: $content-min-width) {
      @content;
    }
  } @else if $size == tablet {
    @media screen and (min-width: 700px) {
      @content;
    }
  } @else if $size == desktop {
    @media screen and (min-width: 1200px) {
      @content;
    }
  }
}
```

А ось приклад використання цього міксину:

```scss
@include for-size(tablet) {
  padding: 0 40px;
}

@include for-size(desktop) {
  padding: 0 50px;
}
```

Трюки вирівнювання елементів



Цікаво, що в ситуації з картинки вище за допомогою засобів CSS неможливо вирівняти стрілку по центру між двома текстовими блоками так, щоб вона сравді завжди була чітко по центру при зміні розміру вікна. Причина тому — автоматичних перенос слів.

При автоматичному переносі слів ширина текстового блока майже ніколи не відповідає ширині фактивного тексту та може постійно змінюватись. При цьому стрілка вирівнюється відносно текстового блоку, а не фактичного тексту, тому вона ніколи візуальноне розташовується по центру між текстовими блоками у цьому випадку.

Щоб вирівнювати стрілку завжди чітко по центру, ширина текстового блоку завжди повиннна відповідати ширіну контенту. Цього можна досягнути за рахунок display: inline. Проте просто при використанні display: inline вже не вийде зберігти таке позиціонування.

Цю проблему було обговорено у великих спілках фронтенд-розробників, і ніхто не зміг знайти просте рішення. Більш того, не дужу очевидно знайти й більш складне рішення, проте воно все ж таки було знайдено.

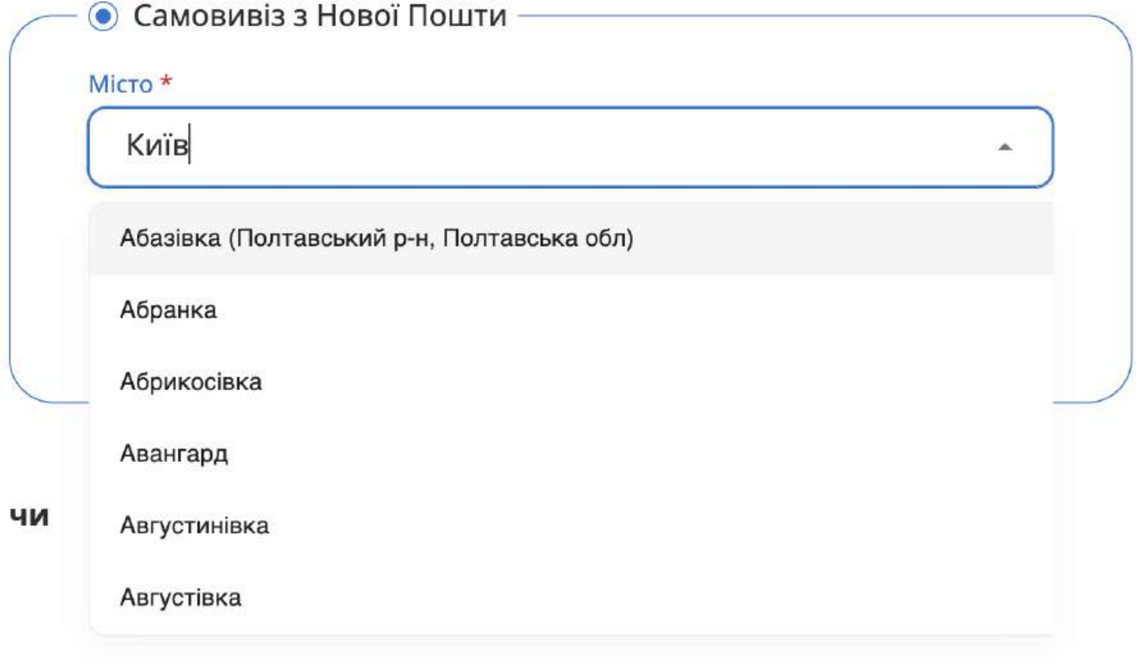
У цій ситуації доведеться створити додатковий HTML-елемент та написати певну кількість JS коду. По-перше, в HTML коді всередині текстового блоку треба обгорнути текстовий літерал у inline-елемент, частіше всього в таких випадках це span. Як видно на картинці нижче, тепер є елемент, який завжди відповідає фактичній ширині контенту навіть при автоматичному переносі слів:

Тепер доступні два блоки, між якими є простір для розміщення стрілки чітко по центру. Далі залишається через JS отримати значення ширини кожного з цих блоків та їх спільного батьківського блока та за допомогою простої геометрії порахувати відступ абсолютно спозиціонованної стрілки від лівого краю батьківського блока.

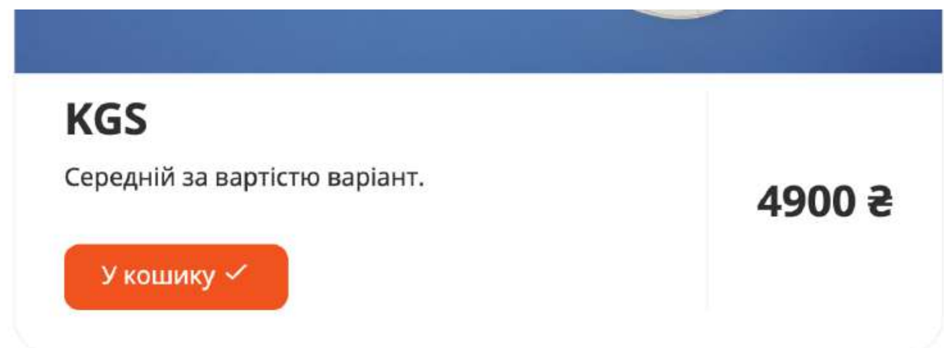
Щоб при зміні розміра вікна стрілка динамічно центрувалася, можна підписатися на resize подію та викликати функцію для перерахунку позиції стрілки.

2.8. Інтеграція бібліотечних компонентів

Майже всі UI компоненти було розроблено з нуля згідно зі створенним дизайном. Проте для деяких елементів було вирішено взяти за основу бібліотечну реалізацію, а потів адаптувати під свої потреби. Проте варто зазначити, що після завершення адаптації деяких компонентів було очевидно, що розробити ці компоненти з нуля було б швидше, аніж знаходи шляхи кастомізації.



Для реалізації випадаючого списку міст з можливісю пошуку за основу було взято MUI компонент Autocomplete. Для покращення алгоритму пошуку було використано бібліотеку match-sorter. Зовнішній вигляд Autocomplete компонента було повністю адаптовано до власного дизайну.



Для реалізації спливаючого повідомлення про успішно доданий до кошика товар за основу було взято компонент бібіотеки notistack. Дизайн також було повністю адаптовано до власних потреб.

2.9. Оптимізація зображень за допомогою компонента Image

Для автоматичної оптимізації всіх зображень було використано компонент Image з фреймворку Next.js. Цей компонент розширює елемент HTML <img> такими функціями:

- Оптимізація розміру: автоматичне відображення зображення правильного розміру для кожного пристрою, використовуючи сучасні формати зображень, такі як WebP і AVIF.
- Візуальна стабільність: автоматичне запобігання зсуву макета під час завантаження зображень.
- Швидше завантаження сторінок: зображення завантажуються лише тоді, коли вони потрапляють у вікно перегляду за допомогою

вбудованого відкладеного завантаження веб-переглядача з необов’язковими заповнювачами з розмиттям (blur-up placeholders).

```
<Image
  src="/images/plug-into.jpg"
  alt="plug in the sinus generator"
  width={681}
  height={681}
  sizes="100vw,
         (min-width: 700px) 50vw,
         (min-width: 1200px) 417px"
  quality={100}
/>
```

```
<img alt="plug in the sinus generator" loading="lazy" width="681" height="681"
decoding="async" data-nimg="1" sizes="100vw, (min-width: 700px) 50vw, (min-widt
h: 1200px) 417px" srcset="/_next/image?url=%2Fimages%2Fplug-into.jpg&w=384&q=100
384w, /_next/image?url=%2Fimages%2Fplug-into.jpg&w=640&q=100 640w, /_next/image?
url=%2Fimages%2Fplug-into.jpg&w=750&q=100 750w, /_next/image?url=%2Fimages%2Fplu
g-into.jpg&w=828&q=100 828w, /_next/image?url=%2Fimages%2Fplug-into.jpg&w=1080&q
=100 1080w, /_next/image?url=%2Fimages%2Fplug-into.jpg&w=1200&q=100 1200w, /_nex
t/image?url=%2Fimages%2Fplug-into.jpg&w=1920&q=100 1920w, /_next/image?url=%2Fim
ages%2Fplug-into.jpg&w=2048&q=100 2048w, /_next/image?url=%2Fimages%2Fplug-into.
jpg&w=3840&q=100 3840w" src="/_next/image?url=%2Fimages%2Fplug-into.jpg&w=3840&q
=100" style="color: transparent;">
```

На прикладі вище створюється елемент зображення з використанням компоненту Next.js Image. З цікавих моментів можна виділити властивість sizes, яка визначає підказки для браузера, для яких розмірів дисплею який

кращий розмір для зображення. На основі цих підказок Next.js генерує комплект зображень різного розміру та вказує ці зображення в атрибуті srcset. Далі браузер самостійно на основі підказок з атрибуту sizes вибирає найбільш підходяще зображення зі списку в атрибуті srcset.

2.10. Використання різних видів рендерингу для покращення SEO та швидкодії застосунку

Для поточного застосунку було використано 2 з 3 варіантів рендерингу: SSG (Static Site Generation) та CSR (Client-side Rendering).

На сторінці зі списком усіх продуктів можна було б використати SSR (Server-side Rendering), бо всі продукти зберігаються на сервері в базі даних и можуть змінюватися. Проте було вирішено зупинитись на отриманні списку продуктів на етапі зборки проєкту (SSG), бо нові продукти додаються дуже рідко й інформація про продукти маже ніколи не змінюється. При цьому, навіть якщо інколи доведеться щось змінювати, можливо просто перезібрати проєкт.

Сторінка оформлення замовлення не буде індексуватися пошуковими системами, тому було вирішено використовувати CSR, бо в загальному випадку це найгнучкіший варіант, який приємно використовувати, коли є така можливість. За допомогою CSR під час завантаження сторінки оформлення замовлення робиться http-запит на отримання всіх українських міст, де є доступні відділення “Нової Пошти”. До того ж для Києва, який є вибраним містом за замовчуванням, запитується список з усіх доступних на поточний момент відділень “Нової Пошти”.

Цікавим ще є елемент кошика, бо він рендериться на всіх сторінках. Цьому компоненту необхідна інформації про всі продукти, яку можна отримати

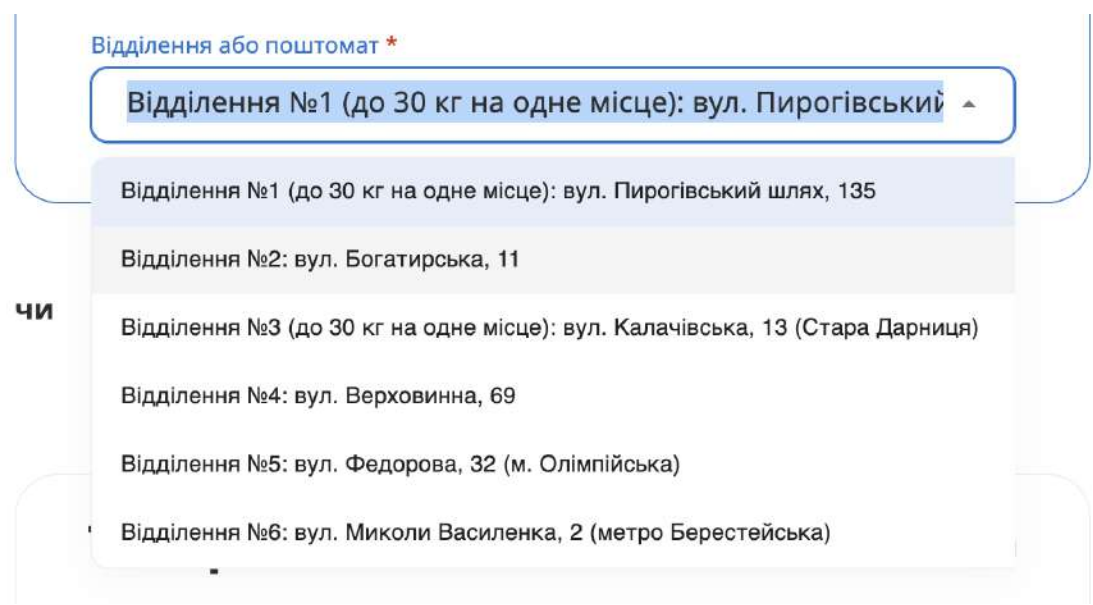
від сервера. На жаль, наразі в арсеналі Next.js немає можливості зробити глобальний getStaticProps (SSG) виклик, щоб при зборці отримати всі продукти та мати до них доступ автоматично на всіх сторінках застосунку. Тому було вирішено у випадку з кошиком використати CSR, бо вміст кошика різний для різних користувачів, і немає необхідності його індексувати пошуковими системами. При використанні CSR є можливість розмістити компонент кошика у спільному для всіх сторінок файлі _app.tsx, потім створити React контекст та провайдера цього контексту, який буде робити запит на отримання всіх продуктів при першому зверненні до застосунку, та який збереже всі продукти у своєму стані для подальшого доступу до них на будь-якій сторінці сайту.

Для всіх інших сторінок використовується SSG, бо на них розміщується статичний контент.

2.11. Використання local storage для збереження товарів у кошику між браузерними сесіями

Усі додані до кошика товари зберігаються на клієнті в local storage в браузері кожного користувача, щоб між браузерними сесіями ці дані не втрачалися.

2.12. Реалізація віртуальної прокрутки для відображення великої кількості елементів різної висоти у списку



Кількість відділень та поштоматів у такому великому місті, як Київ, може сягати тисячі елементів. Якщо всі ці елементи просто додати у випадаючий список, то цей компонент почне сильно глючити.

При цьому досить зручно мати можливість досить швидко доскролити до, наприклад, трьохсотого або чотирьохсотого елемента у декілька рухів, щоб не починати вводити номер відділення вручну.

Щоб мати можливість скролити серед тисяч елементів, але при цьому не мати глюків, було вирішено скористатися технікою віртуального скролінгу. При цьому підході з усього списка рендериться тількa та частина елементів, яка потрапляє у видимо область. При скролінгу нові елементи рендеряться, а ті, які не потрапляють у видиму область, видаляються.

В результаті була отримана можливість відображати тисячи відділень та поштоматів, які швидко скроляться без будь-яких підвисань.

2.13. Реалізація локалізації за допомогою бібліотеки next-i18next

Для спрощення реалізації локалізації було використано бібліотеку next-i18next. Вона забезпечує такі можливості: управління перекладом контенту, надання функцій та хуків для для перекладу React компонентів, SSR/SSG, простори імен для розбиття контенту на групи.

Для початку необхідно створити в корені проєкту файл next-i18next.config.js та визначити локалі, які буде підтримувати застосунок:

```
module.exports = {
  i18n: {
    defaultLocale: "en",
    locales: ["uk", "en", "ru"],
  },
};
```

next-i18next.config.js

Для поточного застосунку було визначено такі локалі: українська (uk), англійська (en), російська (ru). У якості локалі за замовчуванням було вибрано англійську. Ця локаль буде використовуватися, коли немає можливості визначити локаль користувача або ж користувач використовує локаль, якої немає у списку визначених локалей застосунку.

Тепер у файлі /pages/_app.tsx необхідно обгорнути компонент App у функцію вищого порядку appWithTranslation, яка надасть застосунку провайдера I18nextProvider для доступу до контексту перекладу:

```
no usages   Nikita Maslov
export default appWithTranslation(App);
```

Далі необхідно створити структуру контенту з перекладом, про яку буде знати next-i18next. За замовчуванням бібліотека відслідковує файли, які знаходяться за таким шляхом: /public/locales:

- public — папка для збереження усіх статичних файлів, про яку знає та яку обслуговує Next.js
    - locales — папка з усіма файлами з контентом для різних локалей, про яку знає та яку обслуговує next-i18next
        - en — файли з контентом для англійської локалі
            - common.json — контент англійською, спільний для всіх сторінок
            - home.json — контент англійською для головної сторінки
        - uk — файли з контентом для української локалі
            - common.json — контент українською, спільний для всіх сторінок
            - home.json — контент українською для головної сторінки
        - ru — файли з контентом для російської локалі
            - common.json — контент російською, спільний для всіх сторінок

- home.json — контент російською для головної сторінки

У папці для кожної локалі у прикладі вище створено по 2 файли: common.json та home.json. Ці 2 файли визначають простори імен для контенту. Для кожної сторінки застосунку прийнято створювати свій простір імен, щоб у компонент сторінки передавався тільки той контент, який буде використовуватися на цій сторінці. Для контенту, який спільний для багатьох сторінок, використовується окремий простір імен у файлі common.json. Варто зазначити, що для кожної папки локалі повинна зберігатися однакова структура файлів та структура контенту всередині цих файлів, щоб next-i18next могла чітко знати відповідні елементи для всіх локалей застосунку.

У якості прикладу було реалізовано переклад одного з модулів на головній сторінці. Наразі необхідно визначити json файл з перекладом усіх частин модуля. Можна реалізовувати вкладеність будь-якого рівня:

```
{
  "intro": {
    "heading": "Котушка Мішина\u00A0— ваш домашній лікар",
    "subheading": "Прилад для\u00A0лікування більшості захворювань у\u00A0домашніх умовах",
    "textBlock1": "Котушка Мішина допомагає позбутися гіпертонії, хвороб нирок, аритмії, ВСД, ангіни та\u00
    "textBlock2": "Технологія має 6 основних переваг: немає побічних ефектів; лікує більшість захворювань;
    "textBlock3": {
      "mainText": "Ми розповімо про\u00A0те, як\u00A0працює котушка Мішина, не\u00A0залазячи глибоко у\u00A
      "linkText": "7-годинний вебінар з\u00A0вихрової медицини"
    }
  }
}
```

/uk/home.json

Щоб переклад формувався на стороні сервера чи при збірці проєкту, необхідно всередині getServerSideProps або getStaticProps відповідно у props передати результат виклику асинхронної функції

serverSideTranslation. Ця функція приймає на вхід значення локалі, яке отримується з об'єкту аргументу функцій getServerSideProps або getStaticProps, та масив просторів імен для поточної сторінки:

```
import DefaultLayout from "@/components/layouts/Default";
import HomePage from "@/components/templates/HomePage";
import { serverSideTranslations } from "next-i18next/serverSideTranslations";

no usages  Nikita Maslov
export default function Home(): JSX.Element {
  return (
    <DefaultLayout>
      <HomePage />
    </DefaultLayout>
  );
}

no usages  Nikita Maslov
export async function getStaticProps({ locale }: { locale: string }) : Promise<{...}>  {
  return {
    props: {
      ...(await serverSideTranslations(locale,  namespacesRequired: ["common", "home"])),
    },
  };
}
```

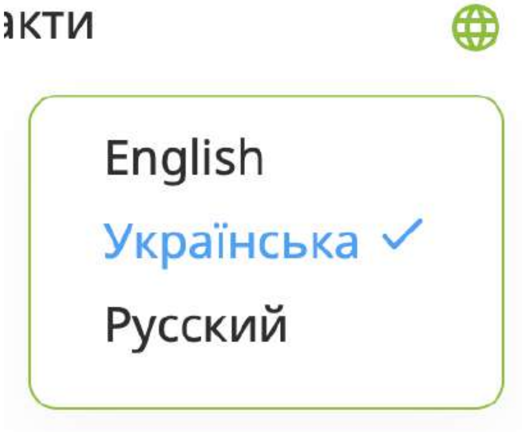
Компонент головної сторінки (home page)

Щоб скористатися можливістю перекладу всередині React-компонента, необхідно спочатку отримати функцію t з хука useTranlation, а потім викликати цю функцію у місці, де треба підставити конретний текст:

```
function Intro() : JSX.Element  {
  const { t : TFunction<"translation", undef...  } = useTranslation();

  return (
    <section className={styles.intro}>
      <IntroImage />
      <Container>
        <h1 className={styles.heading}>{t( key: "home:intro.heading")}</h1>
        <p className={styles.subheading}>{t( key: "home:intro.subheading")}</p>
        <div className={styles.mainContentWrapper}>
          <div className={styles.textBlock}>
            <p>{t( key: "home:intro.textBlock1")}</p>
            <p>{t( key: "home:intro.textBlock2")}</p>
            <p>
              {t( key: "home:intro.textBlock3.mainText")}{" "}
```

Функція t приймає у якості аргументу стрічку за певним шаблоном: спочатку вказується назва простору імен з двокрапкою, а потім через крапку йде доступ до конретного вкладеного елементу з json-файлу.

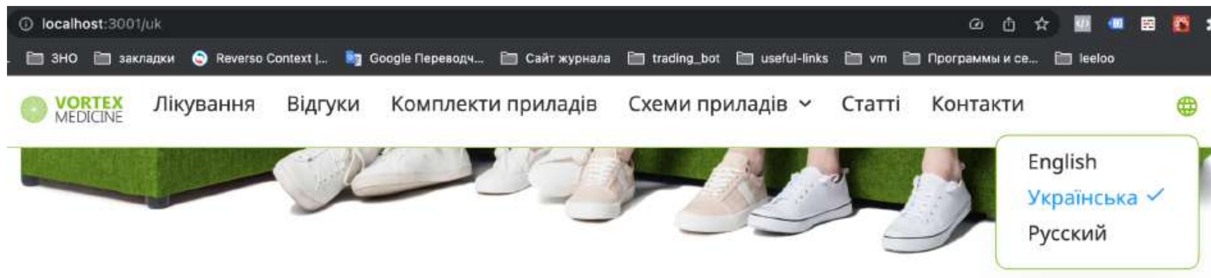
Для ручного переходу між мовами було реалізовано спеціальний перемикач:



При виборі іншої мови викликається метод Next.js маршрутизатора push, у який передається значення нової локалі. Взагалі метод push використовується для перенаправлення на іншу сторінку, проте у цьому випадку сторінка залишається тією ж самою, але з іншою локаллю. Щоб не було повернення на початок сторінки, передається додаткова опція scroll: false:

```
const router : NextRouter = useRouter();

const switchToLocale = useCallback(
  callback: (locale: string) => {
    const path : string = router.asPath;

    return router.push(path, path, options: { locale, scroll: false });
  },
  deps: [router]
);

const languageChanged = useCallback(
  callback: async (option: LanguageOption) : Promise<void> => {
    setValue(option);

    const locale : string = option.value;

    await switchToLocale(locale);
  },
  deps: [switchToLocale]
);
```

У результаті було отримано багатомовний модуль на головній сторінці:



Котушка Мішина — ваш домашній лікар

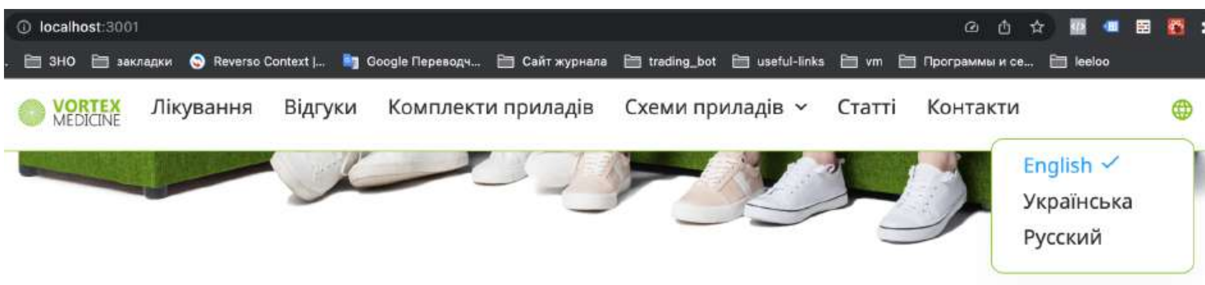
Прилад для лікування більшості захворювань у домашніх умовах

Котушка Мішина допомагає позбутися гіпертонії, хвороб нирок, аритмії, ВСД, ангіни та інших захворювань.

Технологія має 6 основних переваг: немає побічних ефектів; лікує більшість захворювань; достатньо одного приладу на все життя; лікує в домашніх умовах; компактний, тому можна взяти в подорож; легко користуватися.

Ми розповімо про те, як працює котушка Мішина, не залазячи глибоко у фізику процесів. Якщо добре знаєтеся на фізиці, подивіться 7-годинний вебінар з вихрової медицини.

150 рокі
техноло



Mishin's coil is your home doctor

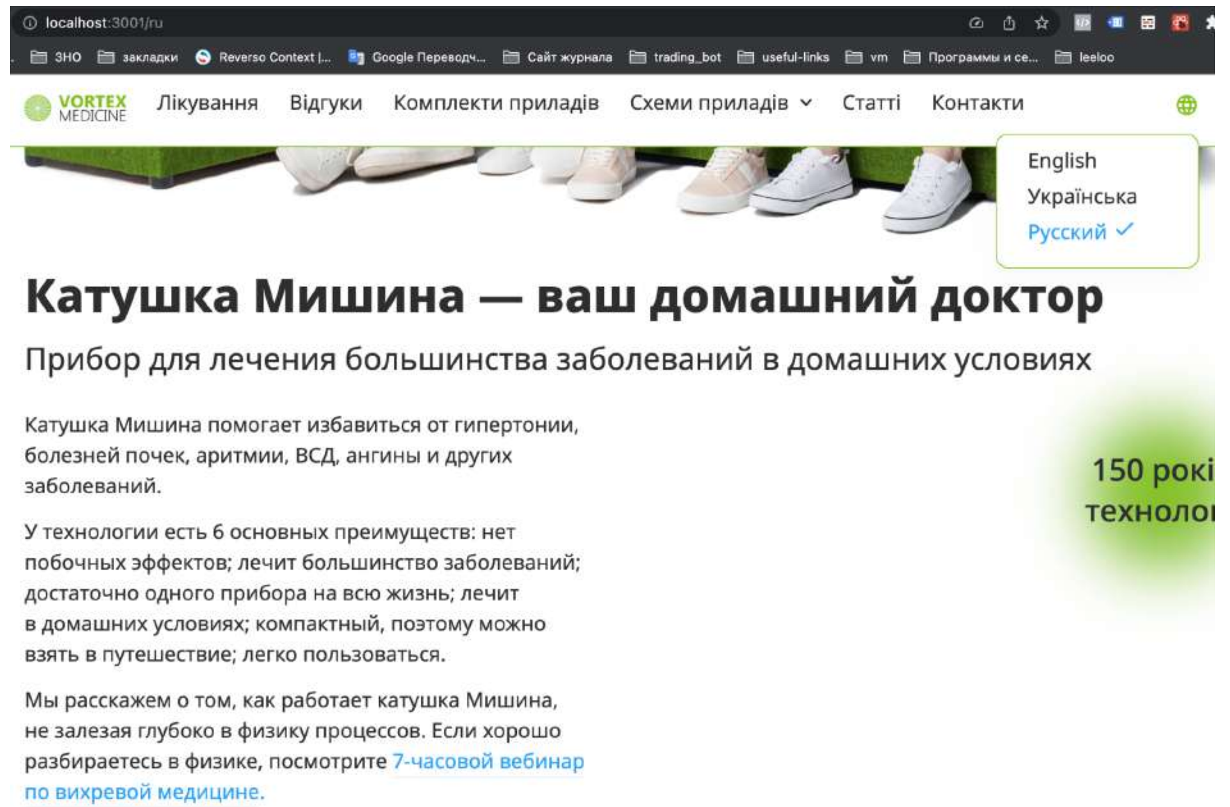
Device for the treatment of most diseases at home

Mishin's coil helps to get rid of hypertension, kidney diseases, arrhythmia, VVD, tonsillitis and other illnesses.

The technology has 6 main advantages: no side effects; treats most diseases; one device for the whole life; treats at home; compact, so it can be taken on a trip; easy to use.

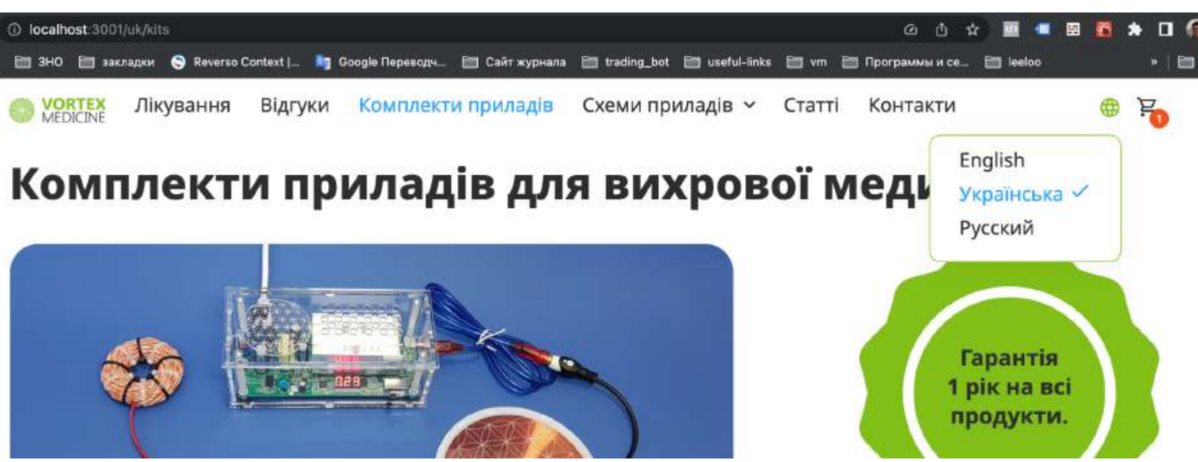
We will tell you about how Mishin's coil works without delving deeply into the physics of the processes. If you are well versed in physics, watch the 7-hour webinar on vortex medicine.

150 рокі
технолоі



Варто звернути увагу, що у стрічці браузера завжди відображається поточна локаль. Тільки для локалі за замовчуванням (англійської) нічого не відображається.

При переході між сторінками застосунку, локаль зберігається та продовжує відображатися у стірці браузера:



2.14. Реалізація REST API за допомогою фреймворку Axum (Rust)

Для реалізації бекенду було використано мову програмування Rust та фреймворк Axum. Серверна частина реалізована на основі структури, схожої на MVC, та принципу слабкого зв'язування.

Структура бекенду виглядає наступним чином:

- src — вихідний код бекенду
  - controllers — усі обробники запитів
  - models — усі моделі даних
  - repositories — інтерфейси та конкретні реалізації взаємодії з базами даних
  - utils — допоміжні функції
  - constants — усі глобальні константи

У якості прикладу реалізації слабкої зв'язанності можна виділити підхід до взаємодії з базою даних. Для моделей даних визначаються інтерфейси, які відповідають за CRUD операції з цими даними:

```
#[async_trait]
pub trait OrderRepository {
    type Order;
    type CreateOrderResult;

    async fn create_order(&self, order: Self::Order) -> Result<Self::CreateOrderResult>;
}
```

```
#[async_trait]
pub trait ProductRepository {
    type Product;
    async fn get_all_products(&self) -> Result<Vec<Self::Product>>;
    async fn get_product_by_id(&self, product_id: &str) -> Result<Option<Self::Product>>;
}
```

Мається на увазі, що всередині конкретні реалізації цих інтерфейсних функцій повинні взаємодіяти з конкретними базами даних.

Далі для цих інтерфейсів можливо створити нескінченну кількість реалізацій, в яких, наприклад, можуть використовуватися різні бази даних. Для поточного застосунку ці інтерфейси були реалізовані в контексті взаємодії з MongoDB:

```
#[derive(Debug, Clone)]
pub struct MongoRepository {
    orders: Collection<Order>,
    products: Collection<Product>,
}

impl MongoRepository {
    pub async fn init() -> Self {
        let uri : String = dotenv::var( key: MONGOURI).unwrap();

        let client : Client = Client::with_uri_str(uri).await.unwrap();
        let db : Database = client.database( name: DB_NAME);
        let orders : Collection<Order> = db.collection::<Order>( name: ORDERS_COLLECTION);
        let products : Collection<Product> = db.collection::<Product>( name: PRODUCTS_COLLECTION);

        Self { orders, products }
    }
}

#[async_trait]
impl OrderRepository for MongoRepository {
    type Order = Order;
    type CreateOrderResult = FullOrder;
    async fn create_order(&self, order: Self::Order) -> Result<Self::CreateOrderResult> {...}
}

#[async_trait]
impl ProductRepository for MongoRepository {
    type Product = Product;
    async fn get_all_products(&self) -> Result<Vec<Self::Product>> {...}
    async fn get_product_by_id(&self, product_id: &str) -> Result<Option<Self::Product>> {...}
}
```

Далі у застосунку скрізь, де можна, замість конкретної MongoDB реалізації вказуються інтерфейси. Наприклад, визначення контролера запиту POST /orders виглядає наступним чином:

```
pub async fn create_order<R, O>(
    State(db): State<Arc<dyn OrderRepository<CreateOrderResult = R, Order = O>>>,
    Json(order): Json<O>,
) -> impl IntoResponse {
```

У цьому прикладі замість того, щоб вказати конкретну реалізації MongoRepository використовується інтерфейс OrderRepository.

І тільки на найвищому рівні в програмі вказується та передається конкретна реалізація інтерфейсів:

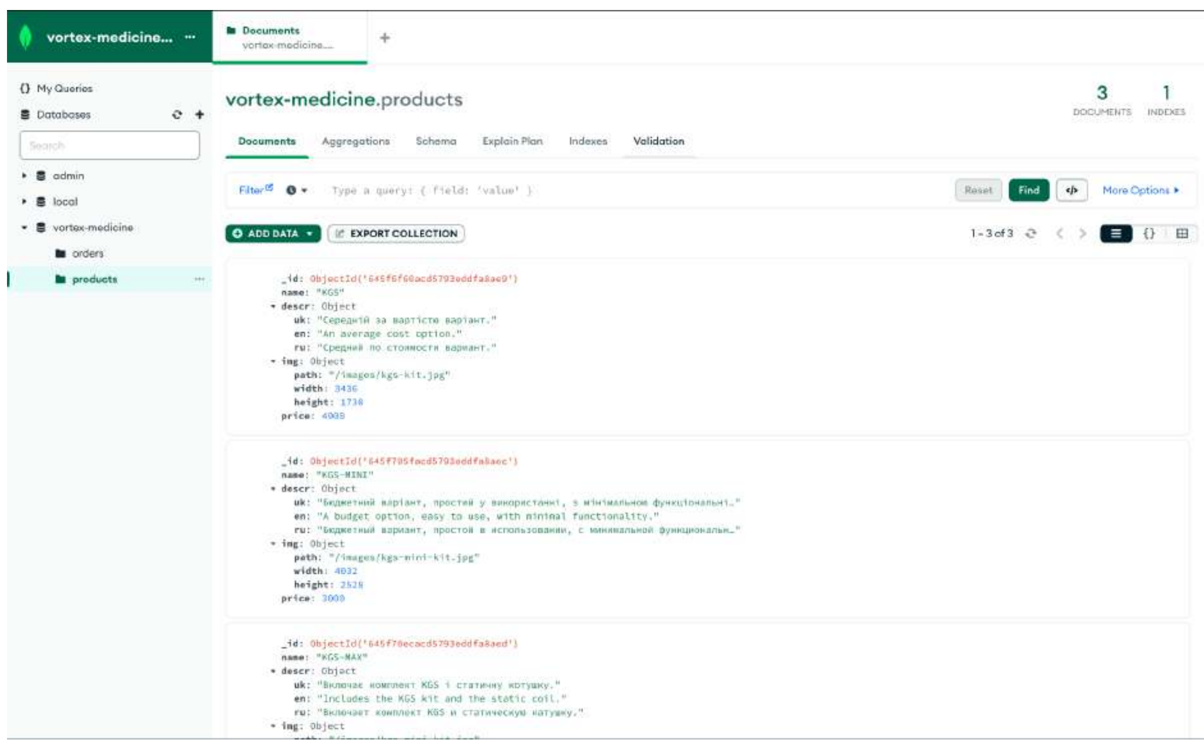
```
let db = Arc::new(MongoRepository::init().await);

let app = Router::new()
    .route("/orders", post(create_order))
    .route("/products", get(get_all_products))
    .route("/products/:product_id", get(get_product_by_id))
    .layer(
        CorsLayer::new()
            .allow_origin("http://localhost:3001".parse::<HeaderValue>().unwrap())
            .allow_methods([Method::GET, Method::POST])
            .allow_headers([http::header::CONTENT_TYPE]),
    )
    .with_state(db);
```

У результаті такого підходу дуже легко тестувати різні реалізації інтерфейсів CRUD операцій, просто передаючи різні екземпляри в одному місці.

Щоб дозволити доступ до API тільки для створенного Next.js клієнта, було зроблено налаштування CORS політики:

```
CorsLayer::new()
    .allow_origin("http://localhost:3001".parse::<HeaderValue>().unwrap())
    .allow_methods([Method::GET, Method::POST])
    .allow_headers([http::header::CONTENT_TYPE]),
```

У якості GUI для простої взаємодії з MongoDB було використано MongoDB Compass:



За допомогою MongoDB Compass зручно робити запити з фільтруванням. Наприклад, якщо потрібно отримати всі замовлення, у яких була вказана певна електронна адреса, то це легко зробити за допомогою рядка filter:

```
Filter  { "email": "maslov.n.e@gmail.com" }

ADD DATA   EXPORT COLLECTION

_id: ObjectId('645fa1b2e9a6174ef79e43ba')
firstName: "Нікіта"
lastName: "Маслов"
email: "maslov.n.e@gmail.com"
phoneNumber: "+380 (67) 538 47 98"
delivery: Object
orderedProducts: Array

_id: ObjectId('645fca5348b255dad446d3e1')
firstName: "Нікіта"
lastName: "Маслов"
email: "maslov.n.e@gmail.com"
phoneNumber: "+380 (67) 538 47 98"
delivery: Object
orderedProducts: Array
```

2.15. Відправка повідомлень про нове замовлення через Телеграм-бот

Щоб адміністратори інтернет-магазину якомога швидше дізнавалися про нове замовлення, було реалізовано автоматичну відправку повідомлень через Телеграм-бот.

Для цієї задачі було використано офіційне API Телеграм:

```
pub async fn send_telegram_notification(message: &str, token: &str, chat_id: &str) -> Result<()> {
    let payload : Value = json!({
        "text": message,
        "chat_id": chat_id
    });

    let client : Client = reqwest::Client::new();
    client
        .post( url: format!(
            "https://api.telegram.org/bot{token}/sendMessage",
            token = token
        ))
        .json(&payload)
        .send()
        .await?;

    Ok(())
}
```

У результаті в створений через BotFather Телеграм-бот надходять повідомлення про нове замовлення у відформатованному вигляді у форматі JSON:


```
New order: {
 "_id": {
  "$oid": "6460e3eea582fcd1cfe0809e"
 },
 "creationTime": "2023-05-14T13:36:45.790793Z",
 "firstName": "Ярема",
 "lastName": "Label",
 "email": "yar@gmail.com",
 "phoneNumber": "+380 (67) 538 47 98",
 "delivery": {
  "kind": "novaposhta",
  "data": {
   "city": "Київ",
   "warehouse": "Відділення №145 (до 30 кг): вул. Героїв
Дніпра, 53"
  }
 },
 "orderedProducts": [
  {
   "product": {
    "id": "645f6f60acd5793eddfa8ae9",
    "name": "KGS",
    "price": 4900.0
   },
   "quantity": 3
  },
  {
   "product": {
    "id": "645f705facd5793eddfa8aec",
    "name": "KGS-MINI",
    "price": 3000.0
   },
   "quantity": 2
  }
 ]
}
16:36
```

Висновки

У даній роботі було розглянуто багато технологій, фреймворків, бібліотек, підходів та архітектурних рішень для створення сучасних інтерактивних веб-застосунків. Якщо зробити перелік по назвах, то буде виглядати приблизно так:

- Figma, Photoshop;
- SPA;
- React;
- Next.js;
- TypeScript;
- SASS modules;
- MUI;
- Mobile First;
- адаптивна, відгукова, гібридна верстка;
- оптимізація зображень;
- SEO в контексті SPA;
- SSR, SSG, CSR;
- віртуалізація на фронтенді;
- локалізація за допомогою next-i18next;
- Rust, Axum;
- MVC;
- слабке зв'язування;
- MongoDB, MongoDB Compass;
- Telegram API

Над чим ще планується працювати:

- *Покращення адмін-панелі.* Наразі в якості адмін-панелі використовується MongoDB Compass. Але цей GUI більш підходить для програмістів, ніж для звичайних менеджерів. Тому планується реалізувати окрему адмін-панель за допомогою React Admin.
- *Авторизація.* Щоб пришвидшити для користувача процес оформлення замовлення, планується реалізувати авторизацію, завдяки якій доведеться запомнювати менше полів форми. Також авторизація допоможе зберігати окремо список користувачів, до яких можуть бути прив'язані додаткові дані, наприклад, стан у партнерській програмі.
- *Партнерська програма.* Планується реалізація партнерської програми, в якій кожному користувачу, який купив хоча б один прилад, присвоюється партнерський код. Якщо інша людина купляє прилад за цим партнерським кодом, то власник цього коду отримує нагороду за партнерською програмою. Функціонал керування партнерською програмою планується додати до адмін-панелі.
- *Реалізація сторінок, які залишилися.* Ще залишилися сторінки, які потрібно зверстати. Переважно це статичні інформаційні сторінки, де необхідна просто механічна робота.
- *Деплой.* Інтернет-магазин потрібно буде опублікувати, коли буде готова перша production-версія. Наразі можна сказати, що планується використання serverless підходу, який буде оптимальним рішенням при початковій невеликій кількості користувачів для економії коштів на серверах.

Список використаних джерел

1. ChatGPT [Електронний ресурс]. Режим доступу до ресурсу: https://chat.openai.com
2. React документація [Електронний ресурс]. Режим доступу до ресурсу: https://react.dev
3. Next.js документація [Електронний ресурс]. Режим доступу до ресурсу: https://nextjs.org/docs
4. TypeScript документація [Електронний ресурс]. Режим доступу до ресурсу: https://www.typescriptlang.org/docs/handbook/intro.html
5. Книга з Rust [Електронний ресурс]. Режим доступу до ресурсу: https://doc.rust-lang.org/book/
6. Axum документація [Електронний ресурс]. Режим доступу до ресурсу: https://docs.rs/axum/latest/axum/