Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему « **ВІДКРИТА ПЛАТФОРМА ДЛЯ ЕЛЕКРОННОЇ КОМЕРЦІЇ З МОЖЛИВІСТЮ ІНТЕГРАЦІЇ ІЗ ЗОВНІШНІМИ СЕРВІСАМИ**»

Виконав: студент 4-го року навчання,

Спеціальності

121 Інженерія програмного забезпечення

Владислав Карагаєв Володимирович

Науковий керівник Кобзар О. О.

Київ – 2024

Зміст

Вступ 2

1. Дослідження предметної області 6

   1.1. Огляд сайту-конструктору “Хорошоп” 6

   1.2 Огляд OpenCart 7

   1.3. Огляд специфікацій товарних фідів 9

2. Ініціалізація застосунку 11

   2.1. Створення моно репозиторію 11

      2.1.1. Ініціалізація Typescript & Eslint 12

      2.1.2. Build tool 12

   2.2. Використання tPRC як головного блоку будування 12

   2.3. Створення серверної частини 13

      2.3.2. Система плагінів Fastify 13

      2.3.2. Використання tRPC на сервері 14

   2.4. Створення адмін панелі 16

      2.4.1. TanstackRouter разом з tRPC 16

      2.4.2. UI компоненти 17

   2.5. Створення клієнтського сайту 17

   2.6. База даних 18

      2.6.1. Клієнт бази даних 18

      2.6.2. Міграції бази даних 20

   2.5. Створення пакетів під бізнес домени 21

3. Створення схем баз даних 23

   3.1. Створення таблиць категорій 24

   3.2. Створення таблиць продуктів 24

   3.3. Створення таблиці атрибутів 26

   3.4. Створення таблиці опцій 27

   3.5. Створення таблиці виробників і мов 27

   3.6. Створення таблиці варіантів продуктів 27

   3.7. Створення таблиці файлових завантажень 29

   3.8. Створення таблиці адміністраторів і користувачів 29

3.9. Створення таблиці замовлень 30
4. Реалізація серверної частини 31
4.1. Авторизація користувача 32
5. Реалізація клієнтських частин 34
Висновок 36

Вступ

Електронна комерція відіграє ключову роль в розвитку успішного бізнесу, все більше підприємців починають використовувати інтернет простір для розширення своєї клієнтської бази. Проте, для успішного просування своєї продукції в конкурентному середовищі потрібні для цього ефективні інструменти.

Перші онлайн підприємства почали з’являтися аж в вісімдесятих роках, аж поки в дев’яностих роках такі компанії як Amazon і eBay не почали використовувати свою інтернет платформу не тільки для продажу власних товарів, а також для дозволили звичайним користувачам продавати свої товари. Це створило зовсім новий ринок, не тільки для Америки, а і для всього світу, більше не треба створювати фізичне місце для продажу товарів, достатньо лише опублікувати свій каталог товарів на обраному маркетплейсі.

З розвитком веб технологій справжній прорив стався коли в 1994-1995 роках почали з’являтися перші сайти які дозволяли оформлювати замовлення через електронну форму. Зазвичай такі інтернет веб-сайти були написані мовою програмування PHP, і аж досі ця мова використовується здебільшого в таких випадках. За допомогою вже готових рішень можна створити функціонуючий інтернет-магазин і просувати свій бізнес, але з часом такі рішення стали застралілими. З розвитком веб-технологій все більше користувачів звикають до використання веб додатків які працюють на одній сторінці без перезавантажень, де попередні рішення спиралися на навігації по гіперпосилань по статичним HTML сторінкам.

В даній роботі досліджено потреби середніх бізнесів і реалізовано онлайн систему для підтримки і росту для підприємств, використовуючи сучасні технології з урахуванням потреб експорту для існуючих маркетплейсів. Розробка схеми бази даних та клієнт серверної розробки в екосистемі Javascript.

Важливо розуміти, нові технології не обов’язково означають що вони набагато краще, інтернет розвивався до їх появи, і все ще залишається таким

самим. Коли малий чи середній бізнес вирішує створити інтернет магазин, вони часто мають лише декілька варіантів. Це може бути вибір між готових готовий варіантів, сайти написані на PHP, онлайн-конструкотори з закритим кодом, або ж наймати спеціалістів які розроблять індивідуальний сайт з нуля, що може бути ресурсо затратно і не виправданим.

Якщо взяти сайт-конструктор, вони зазвичай більш приємні для користування звичайному користувачу ніж сайти написані на PHP, але може статися так що з'являються якісь вимоги для бізнесу і треба найняти програміста який це змінить, але в таких сайтах міняти код неможливо, оскільки власниками коду підприємство не має, і вони мають спиратися на існуючий функціонал обраної платформи.

З розробкою сайтів з сучасним інтерактивним досвідом потрібно використовувати веб-фреймворк зазвичай написаний на мові Javascript. Їх історія почалася з раннього прототипу від програмного інжерера Джордана Уоллк в компанії Meta(тогочасній Facebook) під назвою "FaxJS." Він надихнутий компонентними бібліотеками PHP, він хотів створити щось подібне для мови Javascript з генерації динамічних сторінок на клієнті, для побудови великих і комплексних користувацьких інтерфейсів. Далі цей прототип переріс в усім відому бібліотеку React.js. З 2010 по 2012 рік фреймворк використовувався для тогдашних зарадач компанії, аж поки в 2013 році не був відкритий до програний код не став публічним.

Після цього, React.js став одним з найпопулярніших front-end фреймворків у всьому світі. Він використовується і для побудови продуктів електронної комерції. Але тут є підводні камені, він ніколи не розроблявся під вимоги електронної комерції, історично склалося що сторінки веб-сайтів генеруються на сервері і індексації пошукових систем цих ресурсів використовує саме серверні відповіді. Якщо використовувати додаток з клієнтською генерацією, то пошукова система може неправильно її проіндексувати.

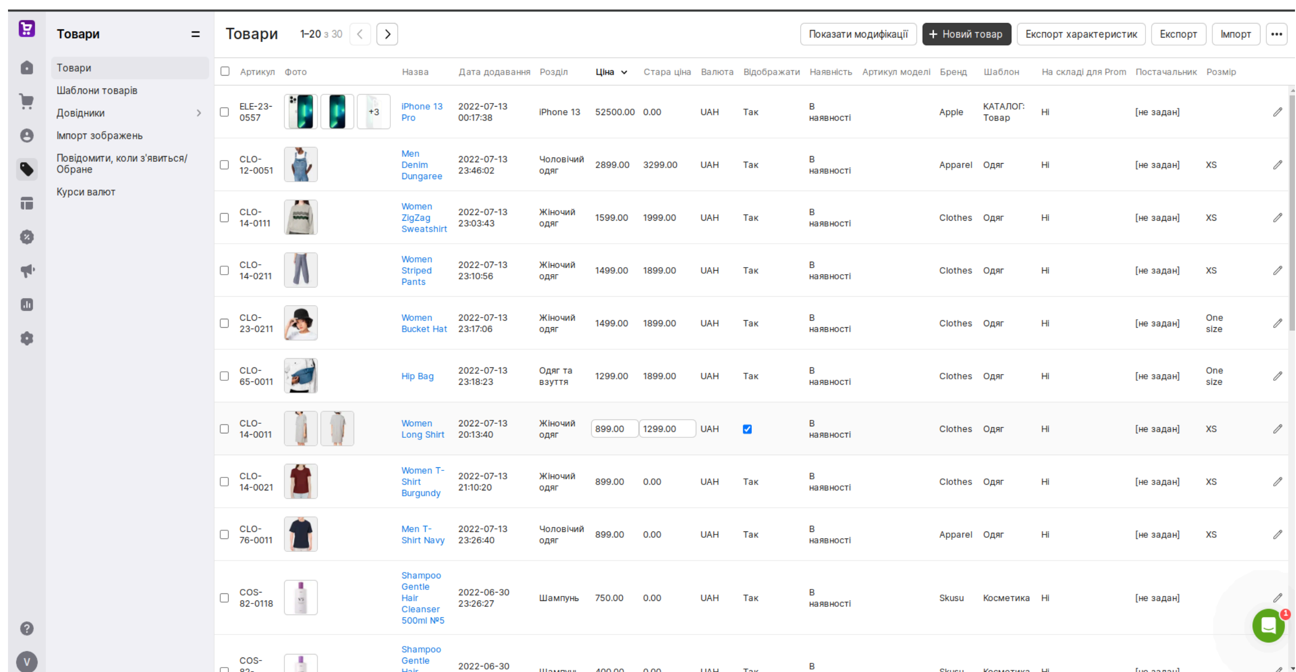
Оскільки, такі додатки зачасту не мають важкої логіки і зазвичай клієнтна частина лише показує відповіді з серверу, в роботі зроблена спроба поєднати front-end та back-end в більш серверну орієнтовну. Як клієнт так і сервер реалізовані однією мовою, ми можемо поєднати їх через комунікацію через бібліотеку tPRC і валідацію одночасно на обох боках через бібліотеку zod, це дозволяє забезпечити за допомого Typescript коректні типи по всьому проєкту, тобто, якщо змінити тип в одній частині програми, то типи будуть оновлені і перевірені. Та більш браузерно-орієнтовний маршрутизатор TanstackRouter, разом з tPRC дозволяє без зайвої логіки завантажувати дані з серверу.

Для експорту товарів на існуючий маркетплейс потрібно підготувати XML файл з відповідною структурою, їх називають фідами. За успішний формат фідів напряму відповідає правильна структура бази даних. Взято з прикладів документацій маркетплейсів Prom та Hotline, а також вже з готових реалізацій побудована структура бази даних. Тут також було обрано не використовувати ORM, оскільки складна структура вимагає створювати довгі SQL запити. Робота в динамічній мові з SQL запитами може призводити до не очевидних помилок, які виникають тільки під час інтерпретації. Зазвичай SQL запити конкатинують разом з аргументами і іншими SQL запитами, цей комплекс проблем було вирішено через генерацію Typescript коду з аналізатором SQL запитів pgTyped.

1. Дослідження предметної області

1.1. Огляд сайту-конструктору “Хорошоп”

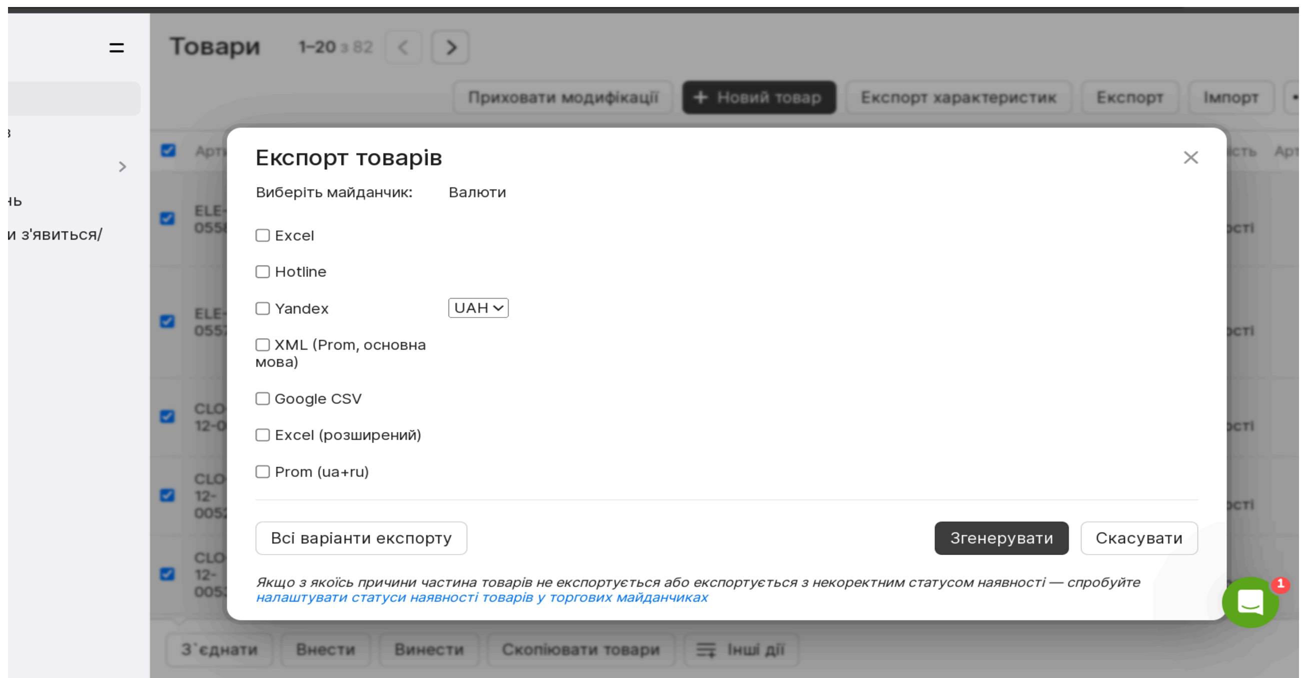
Хорошоп - онлайн сайт конструктор з можливістю експортувати каталог товарів на українські маркетплейси. Він спрямований на швидку розробку сайту для використання вже в реальному бізнесі. При створенні акаунту надається доступ до адмін панелі, в якій нас цікавить експорт продуктів і архітектура зв’язків між таблицями каталогу.



На рисе зображена таблиця продуктів, тут можна побачити безліч різних опцій продукту. При створенні продукту треба створити спочатку базовий продукт, а потім створювати з нього варіанти, наприклад, футболку вонам має декілька варіантів кольору і розоміру, вони всі вважаються як модифікація продукту. В формі продукту є багато полі, можна додати різні характеристики продукту(атрибути/теги), але такі поля як гарантія, назва, опис продукту та інші, є простими не нормалізованими полями. Бренди і категорії товарів представлені як окремі сутності і продукт має посилання на них. Категорії є

рекурсивними, тобто категорія може мати посилання на іншу категорію, це далі відіграє роль в експорті.

Для експорту товарів можна вибрати потрібні продукти і натиснути експорт, після чого відкриється діалогове вікно з опціями експорту:



Тут бачимо різні варіанти експорту, нас цікавить варіант з Hotline і Prom, вони повертаються в форматі XML, з секціями для продуктів, загальної інформації і категорій

Хорошоп надає доступ до сайту за вибраним шаблоном, але знову без кодової кастомізації. Переклади зберігаються разом з моделями в одній таблиці, і мають формат “ключ_ln,” де ln це переклад, тобто виходить український і російський переклади займають два поля таблиці.

1.2 Огляд OpenCart

Opencart - безкоштовна платформа для онлайн комерції з відкритим кодом. Код можна завантажити і самостій встановити. З базового функціоналу доступний адмінка, створення продуктів, атрибутів, опцій і сам каталог. Але не доступні експорти на маркетплейси, оскільки цей продукт не зав’язаний на українському ринку, а на всесвітньому. Пронується купляти різні доповнення для системи за різну ціну. Є які дозволяють програмувати фід прямо в адмінці, а деякі вже підточені під конкретну платформу.

Розглянемо структуру бази даних:

bitnami_opencart **oc_product**
product_id : int(11)
master_id : int(11)
model : varchar(64)
sku : varchar(64)
upc : varchar(12)
ean : varchar(14)
jan : varchar(13)
isbn : varchar(17)
mpn : varchar(64)
location : varchar(128)
variant : text
override : text
quantity : int(4)
stock_status_id : int(11)
image : varchar(255)
manufacturer_id : int(11)
shipping : tinyint(1)
price : decimal(15,4)
points : int(8)
tax_class_id : int(11)
date_available : date
weight : decimal(15,8)
weight_class_id : int(11)
length : decimal(15,8)
width : decimal(15,8)
height : decimal(15,8)
length_class_id : int(11)
subtract : tinyint(1)
minimum : int(11)
rating : int(1)
sort_order : int(11)
status : tinyint(1)
date_added : datetime
date_modified : datetime

bitnami_opencart **oc_product_attribute**
product_id : int(11)
attribute_id : int(11)
language_id : int(11)
text : text

bitnami_opencart **oc_product_filter**
product_id : int(11)
filter_id : int(11)

bitnami_opencart **oc_product_option**
product_option_id : int(11)
product_id : int(11)
option_id : int(11)
value : text
required : tinyint(1)

bitnami_opencart **oc_product_discount**
product_discount_id : int(11)
product_id : int(11)
customer_group_id : int(11)
quantity : int(4)
priority : int(5)
price : decimal(15,4)
date_start : date
date_end : date

bitnami_opencart **oc_product_option_value**
product_option_value_id : int(11)
product_option_id : int(11)
product_id : int(11)
option_id : int(11)
option_value_id : int(11)
quantity : int(3)
subtract : tinyint(1)
price : decimal(15,4)
price_prefix : varchar(1)
points : int(8)
points_prefix : varchar(1)
weight : decimal(15,8)
weight_prefix : varchar(1)

bitnami_opencart **oc_product_description**
product_id : int(11)
language_id : int(11)
name : varchar(255)
description : text
tag : text
meta_title : varchar(255)
meta_description : varchar(255)
meta_keyword : varchar(255)

bitnami_opencart **oc_product_image**
product_image_id : int(11)
product_id : int(11)
image : varchar(255)
sort_order : int(3)

В таблицях можна побачити що кожен теж має багато полів які можна було винести в інші таблиці. Кожен продукт який має багато опцій, є одним записом в таблиці products, де в Хорошопі, кожен продукт створювався від базового продукту.

Переклади полів представлені в інших таблицях, по прикладу: oc_product_description, oc_product_value_description і так далі. Кожна така таблиця містить ключ на таблицю з мовами і на саму модель, таку таблицю можна використовувати разом з операцією з’єднання “JOIN.” Взагалі розробники OpenCart не соромляться робити так всі переклади, це збільшує кількість коду, але сприяє на інтронізацію продукту.

Як і в Хорошопі продукт має атрибути, категорію і виробника в інших таблицях. Таблиця категорій також є рекурсивною.

Оскільки OpenCart є відкритою, можна заглянути в код, і подивитися як реалізовані компоненти системи. В головній директорії розміщені папки: admin, catalog, extension, image, install, system. Вони відповідають за різні частини сайту, основна архітектура цих частин є модель MVC, в моделях в основному

йде робота з базою, кожна модель дублюється як мінімум двічі, в admin та catalog:

```
product.php admin/model/catalog
<?php
namespace Opencart\Admin\Model\Catalog;
/**
 * Class Product
 *
 * @package Opencart\Admin\Model\Catalog
 */
class Product extends \Opencart\System\Engine\Model {
    /**
     * @param array $data
     *
     * @return int
     */
    public function addProduct(array $data): int {
        $this->db->query("INSERT INTO `" . DB_PREFIX . "product` SET `master_id` = '" . (int)$data['master_id'] . "', `model` = '" . $this-

        $product_id = $this->db->getLastId();

        if ($data['image']) {
            $this->db->query("UPDATE `" . DB_PREFIX . "product` SET `image` = '" . $this->db->escape((string)$data['image']) . "' WHERE `pr
        }

        // Description
        foreach ($data['product_description'] as $language_id => $value) {
            $this->db->query("INSERT INTO `" . DB_PREFIX . "product_description` SET `product_id` = '" . (int)$product_id . "', `language_i
        }

        // Categories
        if (isset($data['product_category'])) {
            foreach ($data['product_category'] as $category_id) {
                $this->db->query("INSERT INTO `" . DB_PREFIX . "product_to_category` SET `product_id` = '" . (int)$product_id . "', `catego
```

На фото можна побачити типову модель, в очі кидаються запити в одну лінію з кодовими вставками, їх довжина може бути занадто великою, а також відголосок минулого з DB_PREFIX. Раніше на однім базі даних могло бути декілька сайтів, але зараз на це відпав всілякий ссенс.

1.3. Огляд специфікацій товарних фідів

Товарний фід - це структурований файл який містить дані про вміст каталогу продуктів. Фід може бути сформованим в форматі XML, CSV, TXT і інші форматів. Успіх фіду напряму залежить від його якості, кожна платформа вирішує сама якого формату приймати фід і має для цього відповідні вимоги, до прикладу візьмемо деякі приклади з документації маркетплейсу Hotline:

- <?xml … ?> - обов'язковий елемент
- <price> - кореневий обов'язковий елемент
- <date> - дата і час створення фіду
- <categories> - містить в собі теги <category> разом з інформацією про категорії: id, parentId, name
- <items> - тег для продуктів

- <item> - тег продукта
- <id> - ідентифікатор товару
- <categoryId> - ідентифікатор продукту
- <url> - посилання на веб сторінку продукту
- <vendor> - назва виробника товару
- <name> - назва товару
- <priceRUAH> - роздрібна ціна продукту

Звісно існують інші теги і вимоги, їх кожну треба враховувати. Додатково якщо порівнювати з вимогами до маркетплейсу Prom, крім того що там інші назви тегів, можна додати що offer(product) має XML атрибут `group_id,` він як в розглянутій схемі продукту в Хорошоп, group_id це ідентифікатор базового продуку, і всі продукти які мають однаковий `group_id` стають в каталозі на Prom як з інишими параметрами.

2. Ініціалізація застосунку

Фундаментом застосунку є мова програмування Javascript, код як і на frontend так і на backend написаний цією мовою. Для побудови “Fullstack” додатків можна використати фреймворки такі як: NextJS або SveltKit, але було обрано писати більш традиційну клієнт-серверну архітектуру, через те що зазвичай серверні частини таких проєктів дуже сильно стають пов'язаними з цими фреймворками, і в подальшому такий код дуже складно перевикористати для інших задач.

При підході до проєктування було обрано виносити логіку доменів під конкретний пакет, а вже з серверної частини викликати логіку таких доменів. Це дозволить в майбутньому перевикористати цей пакет. Побудова пакетів використовується мінімально залежностей, тільки ті які відповідають за логіку самого пакету.

З розростанням проєкту і появою нових додатків або пакетів створювати окремий репозиторій для кожно може бути дуже кропіткою роботою, тому для розробки потрібно створити мон репозиторій.

Монорепозиторій - стратегія версії контролю де всі проєкти містяться в одному репозиторії. Він відмінно підходить коли багато коду які сильно пов’язани один з одним

2.1. Створення моно репозиторію

Для ініціалізації потрібно обрати пакетний менеджер, це не так важливо, але в цій роботі використовується ПМ yarn, після чого ініціалізований проєкт з головним package.json в корні проєкту, в нього додається ключ “workspaces” зі значеннями папок під проєктів, у нашому випадку “apps/*” і “packages/*,” так пакетний менеджер сам буде знаходити нові застосунки в цих папках.

2.1.1. Ініціалізація Typescript & Eslint

Спершу треба створити проєкт в папці “packages” під назвою ”typescript-config” разом, і давши йому назву. В цій папці створюються

різноманітні typescript конфігурації. Централізація Typescript конфігів дозволяє з легкістю створювати нові підпакети і додатки в репозиторії. Для використання цих конфігів можна через додавання їх в залежності, а далі в tsconfig додавати через ключ “extends” і шляхом до обраного конфігу.

Для ініціалізації ESLint варто використати такий самий алгоритм дій, так само створюється новий пакет з конфігураціями ESLint, а потім додається в бажані проєкти разом конфігурацією.

2.1.2. Build tool

З часом при додаванні нових пакеті і додатків може виникнути проблема що їх занадто багато. Збірки цих пакетів можуть стати неактуальними, доводиться вручну перебудовувати проєкт до актуальної версії. Для таких ситуацій існують інструменти інкрементальних бандлерів які вирішують проблему будування проєкту в маштабах, а також допомагають вирішувати інші проблеми як запуск тесті і діагностик.

З таких інструментів існують два основних Nx і Turborepo, було обрано Turborepo через мінімальну конфігурацію і простий інтерфейс взаємодії з cli.

2.2. Використання tPRC як головного блоку будування

Однією з цілей було зв'язати клієнт з сервером через систему Typescript, щоб мати API інтерфейс актуальної версії в будь-який час без переписування існуючого коду.

Є декілька варіантів генерації типів з back-end на front-end. Перша це генерація клієнту по існуючій документації OpenAPI використовуючи бібліотеку openapi-typescript, проблема нього в тому що він потребує додатково кроку побудови бібліотеки це не завжди є зручним, коли під час розробки типи різко змінюються актуальність типів бібліотеки може зникнути і доводиться наново перебудовувати. Іншим варіантом є бібліотека tRPC, замість генерації коду, ця бібліотека використовує можливості Typescript в виводі типів.

Ідея tPRC це створення повністю типізованих рутів для на серверній частині, після чого вони експортуються я звичайний тип Typescript без всілякої

генерації коду. Де клієнтів застосунок може додати в залежності проєкт з сервером і імпортувати повністю типізований маршрутизатор серверу. Насправді що відбувається, бібліотека tPRC на клієнтській частині не бере ніякий коди який виконується в Runtime, а лише створює ілюзію через Typescript.

tRPC розшифровується як TypeScript Remote Procedure Calls. Натхненна gRPC, ця бібліотека створена для створення типізованих серверів і клієнтів, що взаємодіють через API. На відміну від gRPC, яка використовує ProtoBuff для серіалізації даних і підтримує багато мов програмування, tRPC фокусується на екосистемі TypeScript і JavaScript.

2.3. Створення серверної частини

Поміж сучасних фреймворків вибір впав між Express та Fastify, було обрано Fastify за його плагін систему яка з легкістю дозволяє будувати новий функціонал і реалізовувати ідеї з розширення серверного застосунку, при цьому дотримуючись принципів інкасуляції.

2.3.2. Система плагінів Fastify

Fastify дозволяє користувачеві розширювати його функціонал за допомогою плагінів. Плагін може бути набором маршрутів, декораторів і будь чим завгодно. Декоратори дають змогу модифікувати об’єкти Fastify, цю техніку можна використовувати для додавання екземплярів класів з бізнес логікою. За умовчанням, плагіни є інкапсульований, тобто якщо викликати декоратор в плагіні він не зможе змінити екземпляр Fastify на рівень вище, а лише тільки його потомків.

Правило інкапсуляції можна прибрати за допомогою пакету “fastify-plugin,” він приймає для себе параметром плагін який потрібно прибрати інкапсуляцію і об’єкт мета інформації. Разом з плагіном “fastify-autoload,” завантажується мета інформації і регулюються певні залежності.

```
admins.ts
import { FastifyInstance } from "fastify";
import fp from "fastify-plugin";
import { Admins } from "@repo/admin";

declare module "fastify" {
  export interface FastifyInstance {
    admins: Admins;
  }
}

export default fp(async function (f: FastifyInstance) {
  const admins = new Admins({ pool: f.pool });

  f.decorate("admins", admins);
}, {
  name: "admin",
  dependencies: ["pool"],
});
```

На рисунку зображено типовий плагін для Fastify, він декорує екземпляр Fastify з Admins екземпляром, також через fastify-plugin відміняється інкапсуляція.

Так fastify-plugin і fastify-autoload використовуються для автоматичного завантаження плагінів в директорії server.

2.3.2. Використання tRPC на сервері

Оскільки було вирішене використовувати tRPC, звичайна маршрутизація Fastify не підходить, тому цей фреймворк відповідає за резолюцію залежностей для рутів tRPC. З’являється проблема в виводу типів tRPC, оскільки рути почали створюватися всередині Fastify, було вирішено створити функцію конструктор цих рутів які приймають аргументом Fastify і повертають повністю збудований маршрутизатор tRPC.

app.router.ts

```ts
import { FastifyZod } from "fastify";
import WebCatalogCategoryRouter from "./web/catalog/category.router";
import WebCatalogPostRouter from "./web/catalog/post/router";
import WebCatalogProductRouter from "./web/catalog/product.router";

export async function createAppRouter(fastify: FastifyZod) {
  const { t } = fastify;

  return t.router({
    web: t.router({
      catalog: t.router({
        category: t.router(await WebCatalogCategoryRouter(fastify)),
        post: t.router(await WebCatalogPostRouter(fastify)),
        product: t.router(await WebCatalogProductRouter(fastify)),
      }),
    }),
  });
}

export type AppRouter = Awaited<ReturnType<typeof createAppRouter>>;
```

На рисунку можна побачити що ми отримуємо готовий маршрутизатор і його тип використовуючи Typescript предикати Awaited та ReturnType. При цьому інкапсуляція tRPC в Fastify нікуди не зникає.

Якщо заглянути назад в код OpenCart, можна згадати що каталог і адміністративна частина знаходяться в різних директоріях такий підхід чудово підпадає під побудову серверної частини. Але з часом кількість tRPC рутів може дуже сильно розростаються і модифікувати функцію яка будує маршрутизатор стає не найбільш надійним.

Для таких задач популярним є рішення генерації коду і файлова маршрутизація, такі фреймворки якщо Nextjs і Nuxtjs вже давно популяризували даний підхід. Достатньо лише додати файл з “.route.ts” в ньому щоб він був завантаженим в маршрутизатор.

Генерація проходить в кілька кроків, спочатку зчитуються директорії з файловими маршрутизаторами, після чого будується дерево, якщо згенерований файл маршрутизатора існує то він порівнюється зі щойно згенерованим, якщо він був змінений(було додано або прибрано маршрутизатор), то цей файл оновлюється з новою схемою.

2.4. Створення адмін панелі

При створенні адміністративної частини було вирішено використати фреймворк React, оскільки його екосистема пропонує широкий вибір бібліотек.

Для адмін панелі не потрібна генерація html коду на сервері, тому достатньо створити проєкт використовуючи vite шаблон для React.

2.4.1. TanstackRouter разом з tRPC

Для маршрутизації було обрано TanstackRouter через його можливість тримати state в параметрах URL та системою завантаження даних. Маршрутизація є також пофайлова, бібліотека також генерує відповідні типи навігації, що зменшує помилки при переміщенні сторінок на інший маршрут. Кожна сторінка має свій контекст і може успадкувати від батьківської сторінки.

Для завантаження самих даних з серверу додано бібліотеку trpc-react-query і передано в головний контекст клієнт tRPC, це дозволить будь-якому маршруту отримувати доступ до методів tRPC.

```
new.tsx

import { createFileRoute, useNavigate } from "@tanstack/react-router";
import { trpc } from "../../utils/trpc";
import { toast } from "sonner";
import { LanguageForm } from "../../components/forms/language-form";

export const Route = createFileRoute("/languages/new")({
  beforeLoad: ({ context }) => ({ ...context, getTitle: () => "New Language" }),
  loader: async ({ context }) => {
    const languages = await context.trpc.admin.language.list.fetch();

    return { languages };
  },
  component: CategoryNewComponent,
});

function CategoryNewComponent() {
  const navigate = useNavigate();
  const utils = trpc.useUtils();

  const mutation = trpc.admin.language.create.useMutation({
    onSuccess: async () => {
      await utils.admin.catalog.category.listCategories.invalidate();
      toast.success("Language created");
      navigate({ to: "/languages" });
```

На рисунку зображений типова сторінка з використанням TastackRouter.

Одне з головних фіч цього роутера в тому що воно не завантажує компонент поки не завантажиться дані з методу loader.

2.4.2. UI компоненти

Доступність(accessibility) сайтів важко досягти особливо при складних компонентах і редагування їх під вимоги може бути досинь не тривіальною задачею. Для UI компонентів взятито UI коменент бібліотеку shadcn, вона є дійсно унікальною, замість простого експорту з пакету компонентів, істаляція

проходить по іншому, використовуючи CLI shadcn, він копіює існуючий код в директорію компонентів. Ці компоненти можна спокійно редагувати.

2.5. Створення клієнтського сайту

На відміну від адміністративної панелі, для аналізу сайту пошукових систем потрібно мати змогу згенерувати html сторінки прямо на сервері, це додає нові деякі умови.

SSR - це процес генерування HTML коду на сервері з надсиланням готовою сторінкою на клієнт, клієнт потім проходить процес гідратації сторінки для повної інтерактивності компонентів.

Треба враховувати що код може поводити себе по різно залежно від місця де він виконується, між популярних рішень з SSR є використання готових варіантів з фреймворків NextJS або ж писати самостійно сервер з який буде генерувати сторінки.

NextJS є чудовим рішенням при написанні Fullstack додактів. Він дає "з коробки" змогу генерації і гідрації сторінко, але має деякі рішення які ускладнюють модель взаємодії з сервером, оскільки NextJS спрямований на роботі з його внутрішнім сервером, тобто потрібно писати додаткову логіку для взаємодії з головним сервером. Ще бібліотена trpc разом з NextJS не підтримує варінту коли запити йдуть на інший крім NextJS серверу, доводиться писати власну реалізацію для комунікації з стороннім trpc сервером.

Іншим варіантом є створювати сервер з нуля і самостійно обробляти гідрацію сторінки. Коли запит приходить на сервер, він обробляється наче в запускається на клієнті, перед відправкою на клієнт, зберігається всі дані стану додатку і зберігаються в форматі JSON, після чого клієнт коли отримає цю сторінку читає стан який повернутий з серверу і гідрує залежно від цих даних.

Фреймворком так і залишається React з TanstackRouter, тільки в цей раз з генерацією HTML сторінок на сервері. Виникає проблема що контекст tRPC може неправильно авторизувати користувача на сервері, тому було прийнято рішення генерувати дані тільки ті які є публічними, тобто весь специфічні дані користувача генеруються тільки на клієнті. Для комунікації з сервером є два trpc

клієнти, один для виклику публічних маршрутів через “fetch,” і для даних для конкретного користувача.

2.6. База даних

Одним з найпопулярніших баз даних є PostgreSQL, було обрано її через її розповсюдженість. База даних MySQL є більш розповсюджена для проєктів електронної комерції, але не підтримує деякі функції PostgreSQL, такі як тип колонок jsonb. В MySQL дані в форматі JSON зберігаються просто стрічкою, а в PostgreSQL можна робити запити по цим полям.

2.6.1. Клієнт бази даних

Для комунікацій з базою потрібно мати якусь ORM або писати запити напряму базу даних. Повертаючись до реалізації OpenCart, база даних сильно залежить від Composite ключів. Враховуючи потреба потрібно обрати ORM, з популярних рішень є TypeORM і Prisma, обоє з них не підтримують складені ключі, а Prisma більш дата-фреймворк ніж Orm, і буде складно реалізувати схему бази даних разом з нею. Через складний дизайн таблиць, було обрано писати запити в базу даних.

Великий плюс ORM того що вони виводять типи від схеми бази даних, де в чистих запитах потрібно додатково переводити тип на бажани, що може призвести до людської помилки. Згадуючи код з OpenCart, в голову приходить довгі виклики бази даних разом з вставками коду, це не є дуже читабельним, альтернативою є писати разом запити з параметризацією, прикладом такого запиту є:

```
const text = "INSERT INTO users(name, email) VALUES($1, $2) RETURNING *";
const values = ["brianc", "brian.m.carlson@gmail.com"];

const res = await client.query<{ id: number; name: string; email: string }>(
  text,
  values,
);
console.log(res.rows[0]);
```

Кожен параметр має своє число і при виклику цього запиту потрібно передати масив з правильним порядком значень, і до цього потрібно додатково передати тип повернених даних, які в Runtime можуть не співпадати з дійсними. Альтернативою є використання бібліотеки “pgTyped”.

PgTyped - дозволяє писати SQL запити з безпечними типами, а також на етапі компіляції перевіряти валідність запиту. Не потрібно створювати додаткові типи для параметрів і результату, pgTyped автоматично створить типи за допомогою існуючих схем бази даних. Прикладом такого запиту є:

```
const userCreateQuery = sql<UserCreateQuery>`
  INSERT INTO users(name, email)
  VALUES ($name, $email)
  RETURNING *
`;

userCreateQuery.run({
  email: "test@email.com",
  name: "Jorge Bush",
}, client);
```

Бібліотека автоматично знаходить SQL запити і генерує в типи конфігуровану директорію, після чого можна використати цей тип. Але є обмеження, SQL запити мають бути статичними і не генеруються в Runtime, тобто нема змоги конкатинувати стрічки з SQL запитів, альтернативою є використання вбудованих предикатів: COALESCE, CASE, тощо. Це змушує писати запити більш довгими але вкладеними в один зміну яка читаєма.

При змінах таблиць можна перевірити валідність всіх запитів, це зменшує проблем при додаванні нового функціоналу.

2.6.2. Міграції бази даних

Оскільки було обрано не використовувати ORM, доводиться створювати таблиці самостійно, є багато рішень для міграцій бази даних з різним набором функціоналу. Є два основних варіанти, створення таблиць через код і запуск SQL файлів з кодом міграції. Було обрано більш традиційний варіант запуску через SQL файли, за допомогою утиліти "postgrator," для створення міграцій потрібно в директорії migrations створити файл з відповідним форматом, цей формат містить номер міграції, команду do/undo та назву самої міграції. Прикладом такого файлу є:

```
00001.do.create-language-table.sql migrations
CREATE TABLE languages (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL
);
```

Тут міститься звичайний sql код, для виклику міграції потрібно виконати комадну "db:migrate," після чого буде порівняна поточна версія бази даних з наявною. Можна вказувати конкретну версію бази даних, але для зворотних міграцій треба створювати прямо протилежні до "do" фалів "undo," в яких міститься код до повернення версії баз даних.

2.5. Створення пакетів під бізнес домени

Важливо відокремити логіку основних компонентів яка має свою бізнес логіку, а в серверна частина лише викликає ці методи. Так для домену каталог створюється відповідний пакет, в ньому розробляється логіка взаємодії каталогу. Такий підхід дозволяє декомпозувати програму на маленькі частини, це спрощує розробку і тестування.

Для створення нового пакету досить просто додати в директорію новий проєкт, в ньому має міститися package.json з іменем проєкту. В експортах мають

бути всі необхіді компоненти які можуть знадобитися при використанні. Пакет не має створювати ніякі ресурси чи глобальні змінні, їх конфігурація має створюватися через параметри функцій або конструктори класів.

При розробці в “dev” режимі Typescript код в пакеті може змінитися але на сервері залишиться старий білд, його треба перекомпілювати і тільки тоді оновиться код. Альтернативою є запуск коду напряму, але нам доведеться писати код в Javascript файлах. На перший погляд може здатися що вся типізація зникає як тільки з’являється Javascript файли, але це не так Typescript також підтримує Javascript файли з JSDoc анотаціями, ми позбуваємося етапу трансляції коду з ts в js, але нам все одно доведеться копілювати Typescript декларацийні файли.

3. Створення схем баз даних

Проаналізувавши існуючих рішення і вимоги до товарних фідів для Prom і Hotline, можна винести деякі схожі компоненти:

- Категорії - містить в собі назву і мають рекурсивний зв’язок.
- Атрибути - теги для продукту, продукт може мати багато екземплярів
- Опції - варіанти продуктів, розмір, колір, тощо. Також продукт може містити декілька груп опцій і це треба враховувати.
- Продукти - таблиця з продуктами, може містити в собі базову інформацію про продукт і є головною для побудови готового фіду
- Виробник - окрема таблиця з виробниками, кожен продукт обов’язково повинен мат його

Якщо користуватися сайтом OpenCart то опції являють собою збірку конкретного продукту, а в Хорошоп вони є окремим продуктом зі власною сторінкою. Враховуючи це треба створювати продукти з різним набором опцій, при цьому маючи конкретну групу продукта.

Також, переклади продуктів є теж важливими, ринок вимагає щоб інтернет ресурс був з підтримкою різних мов. Схожу на реалізацію OpenCart переклади контент з пов’язними таблицями. так виходить що для перекладу таблиці products потрібно мати додаткову таблицю з її перекладами, розглянемо подібну таблицю:

00004.do.create-product-table.sql migrations

```sql
CREATE TABLE product_descriptions (
  product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  language_id INTEGER NOT NULL REFERENCES languages(id) ON DELETE CASCADE,
  name VARCHAR(255) NOT NULL,
  description TEXT,
  short_description TEXT,
  PRIMARY KEY (product_id, language_id)
);
```

Такі таблиці містять Composite ключ з посиланням на конкретну таблицю і на таблицю мов(languages), і додаючі переклади під конкретну мову.

3.1. Створення таблиць категорій

Категорії мають містити в собі рекурсивний зв’язок і мати в собі переклади, також не менш важливо мати slug для подальшого отримання категорії по ньому в веб каталозі.

```
00002.do.create-category-table.sql migrations
CREATE TABLE categories (
  id SERIAL PRIMARY KEY,
  slug VARCHAR(255) NOT NULL,
  parent_id INTEGER REFERENCES categories(id) ON DELETE CASCADE,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE category_descriptions (
  category_id INTEGER NOT NULL REFERENCES categories(id) ON DELETE CASCADE,
  language_id INTEGER NOT NULL REFERENCES languages(id) ON DELETE CASCADE,
  name VARCHAR(255) NOT NULL,
  PRIMARY KEY (category_id, language_id)
);
```

На рисунку можна побачити створення таблиці категорій, варто зазначити що вона рекурсивна через поле parent_id і категорії можу бути перекладеними на інші мови.

3.2. Створення таблиць продуктів

Для реалізації такої моделі, прийнято рішення створити дві таблиці продуктів: products, product_variants. Варіанти продуктів це продукт який має стосунок до продукту ї є головою складовою каталогу, він відображається на сайті каталозі і відправляється в товарний фід. Де таблиця продукту є головною батьківською таблицею, вона містить ключ на виробника і категорію а також базовий переклад для всіх продуктів. Такий дизайн зумовлений тим що кожна опція товару приймає в собі нові дані і є окремими продуктами.

00004.do.create-product-table.sql migrations

```sql
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  vendor_id INTEGER NOT NULL DEFAULT 1 REFERENCES vendors(id),
  category_id INTEGER NOT NULL REFERENCES categories(id) ON DELETE CASCADE,
  created_at TIMESTAMP NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);

CREATE TABLE product_descriptions (
  product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  language_id INTEGER NOT NULL REFERENCES languages(id) ON DELETE CASCADE,
  name VARCHAR(255) NOT NULL,
  description TEXT,
  short_description TEXT,
  PRIMARY KEY (product_id, language_id)
);
```

Варто зазначити що базовий продукт має категорію і виробника, тобто всі опції товару успадковують дані з цієї таблиці, що логічно, опція товару не може бути від різних виробників і категорій.

3.3. Створення таблиці атрибутів

Продукт може мати багато атрибутів, ці атрибути є в групах .
При додаванні нового продукту є можливість вибрати будь-який атрибут з будь-якої групи.

```
00003.do.create-attribute-table.sql migrations
CREATE TABLE attribute_groups (
  id SERIAL PRIMARY KEY,
  sort_order INTEGER NOT NULL
);

CREATE TABLE attribute_group_descriptions (
  attribute_group_id INTEGER NOT NULL REFERENCES attribute_groups(id) ON DELETE CASCADE,
  language_id INTEGER NOT NULL REFERENCES languages(id) ON DELETE CASCADE,
  name VARCHAR(255) NOT NULL,
  description TEXT,
  PRIMARY KEY (attribute_group_id, language_id)
);

CREATE TABLE attributes (
  id SERIAL PRIMARY KEY,
  attribute_group_id INTEGER NOT NULL REFERENCES attribute_groups(id) ON DELETE CASCADE
);

CREATE TABLE attribute_descriptions (
  attribute_id INTEGER NOT NULL REFERENCES attributes(id) ON DELETE CASCADE,
  language_id INTEGER NOT NULL REFERENCES languages(id) ON DELETE CASCADE,
  name VARCHAR(255) NOT NULL,
  PRIMARY KEY (attribute_id, language_id)
);

CREATE TABLE product_attributes (
  product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  attribute_id INTEGER NOT NULL REFERENCES attributes(id) ON DELETE CASCADE,
  PRIMARY KEY (product_id, attribute_id)
);
```

На рисунку зображені таблиці атрибутів, кожен атрибут має свою групу, замість додавання груп атрибутів до про продукту, атрибути додаються напряму. Також містяться переклади для груп і продуктів.

3.4. Створення таблиці опцій

Таблиці опцій схожі на таблиці атрибутів але грають іншу роль замість додавання до продукту, групи опцій додаються до базового продукту, а вже самі опції до їх варіантів.

```
00008.do.create-options-table.sql migrations
CREATE TABLE option_groups (
    id SERIAL PRIMARY KEY,
    sort_order INTEGER NOT NULL,
);

CREATE TABLE option_group_descriptions (
    option_group_id INTEGER NOT NULL REFERENCES option_groups(id) ON DELETE CASCADE,
    language_id INTEGER NOT NULL REFERENCES languages(id) ON DELETE CASCADE,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    PRIMARY KEY (option_group_id, language_id)
);

CREATE TABLE options (
    id SERIAL PRIMARY KEY,
    value jsonb NOT NULL,
    option_group_id INTEGER NOT NULL REFERENCES option_groups(id) ON DELETE CASCADE
);

CREATE TABLE option_descriptions (
    option_id INTEGER NOT NULL REFERENCES options(id) ON DELETE CASCADE,
    language_id INTEGER NOT NULL REFERENCES languages(id) ON DELETE CASCADE,
    name VARCHAR(255) NOT NULL,
    PRIMARY KEY (option_id, language_id)
);

CREATE TABLE product_option_groups (
    product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    option_group_id INTEGER NOT NULL REFERENCES option_groups(id) ON DELETE CASCADE,
    PRIMARY KEY (product_id, option_group_id)
);
```

На рисунку зображені таблиці опцій

3.5. Створення таблиці виробників і мов

Таблиці, languages і vendors є важливими при побудові схеми бази даних, хоча містять лише одне поле - name. Мови відповідають за доступні переклади, а vendors потрапляють в каталог.

3.6. Створення таблиці варіантів продуктів

Варіантами продуктів є продукти з набором опцій, вони потрапляють в експорт, тому важливо всю необхідну інформацію зв’язувати з цією таблицею

00016.do.create-product-variant-table.sql migrations

```
CREATE TYPE product_variant_stock_status AS ENUM ('in_stock', 'out_of_stock', 'preorder');

CREATE TABLE product_variants (
  id SERIAL PRIMARY KEY,
  slug varchar(255) NOT NULL UNIQUE,
  product_id INTEGER NOT NULL REFERENCES products(id),
  stock_status product_variant_stock_status NOT NULL DEFAULT 'in_stock',
  price FLOAT NOT NULL DEFAULT 0,
  old_price FLOAT NOT NULL DEFAULT 0,
  article varchar(255) NOT NULL,
  discount FLOAT DEFAULT 0.0 NOT NULL,
  popularity INT DEFAULT 0 NOT NULL,
  images jsonb DEFAULT '[]'::jsonb NOT NULL,
  barcode varchar(255) NOT NULL,
  is_active BOOLEAN NOT NULL DEFAULT FALSE,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE product_variant_options (
  product_variant_id INTEGER NOT NULL REFERENCES product_variants(id) ON DELETE CASCADE,
  option_id INTEGER NOT NULL REFERENCES options(id),
  PRIMARY KEY (product_variant_id, option_id)
);
```

Проаналізувавши приклади продуктових фідів, спроєктована така таблиця, вона містить всі необхідні варіанти для експорту і відповідні відносини до інших таблиць.

Реалізувати наявність продукту можна по різному, через просте поле зі значенням статусу продукту, або ж реалізовувати комплексну систему продуктового інвентарю.

В полі images зберігається jsonb масив ідентифікаторів файлів, потім ці значення можна отримати при виводі зображень.

Таблиця перекладів для продуктів створюється ідентично до попередніх і містить в собі поля: name та short_description.

3.7. Створення таблиці файлових завантажень

Для завантаження файлів і потім їхнього показу, потрібно створити таблицю file_uploads

```
00011.do.create-file-upload-table.sql migrations M
CREATE TABLE file_uploads (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  filename text NOT NULL,
  mimetype text NOT NULL,
  url text NOT NULL,
  created_at timestamp with time zone DEFAULT now()
);
```

В цій таблиці збережені посилання на файл, його ім'я, тип і його ідентифікатор. Було обрано створювати його з UUID первинним ключем через потребу генерувати його на сервері, а цей формат дозволяє убезпечити унікальність такого ідентифікатор.

3.8. Створення таблиці адміністраторів і користувачів

Для доступу до адміністративних прав потрібно перевіряти чи користувач має на це право, для перевірки варто створити для них таблицю

```
00006.do.create-admin-table.sql migrations
CREATE TABLE IF NOT EXISTS admins (
  id         SERIAL PRIMARY KEY,
  name       VARCHAR(255),
  surname    VARCHAR(255),
  email      VARCHAR(255) NOT NULL,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  UNIQUE (email)
);
```

В цій таблиці не зберігається пароль, тому що ідентифікація користувача здійснюється через адрес електронної пошти, аналогічно створюється таблиця з користувачами

3.9. Створення таблиці замовлень

Замовлення є одним з найважливіших компонентів, при створенні таблиці потрібно враховувати різні варіанти його оформлення, варто дозволяти не авторизованому користувачеві створювати замовлення, можуть бути різними

деталі замовлення, оскільки залежно від типу товарів вимоги можуть змінюватися

```
CREATE TYPE order_status as ENUM ('created', 'processing', 'shipped', 'cancelled');

CREATE TABLE IF NOT EXISTS orders (
  id          SERIAL PRIMARY KEY,
  status      order_status NOT NULL DEFAULT 'created',
  price       NUMERIC(10, 2),
  information JSONB,

  user_id    BIGINT REFERENCES users(id),
  created_at  TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS order_items (
  id         SERIAL PRIMARY KEY,
  order_id   BIGINT NOT NULL REFERENCES orders(id),
  product_variant_id INTEGER NOT NULL REFERENCES product_variants(id),
  price      BIGINT NOT NULL,
  quantity   BIGINT NOT NULL
);

CREATE TABLE IF NOT EXISTS order_history (
  id         SERIAL PRIMARY KEY,
  order_id   BIGINT NOT NULL REFERENCES orders(id),
  status     order_status NOT NULL,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

При створенні замовлення треба зберігати загальну так і ціну індивідуальних продуктів в мить оформлення. Статуси можуть змінюватися з часом, і варто їх зміни теж зберігати.

4. Реалізація серверної частини

Серверна частина відповідає за провадження ендпоінтів для клієнтних частин, вони написатині використовуючи tRPC процедури, в цих процедурах викликається логіка з домених пакетів. Домени імпортуються в Fastify як окремий плагін щоб далі їх можна було використовувавати.

Серверна частина поділена на дві під домени, для адміністраторів(admin) і для загального каталогу(web). Важливо побудувати web частину з урахуванням того що фронтенд може бути заміненим на інший по прикладу шаблону.

На відміну від REST архітектури, процедури виглядають більш як методи класів, вони мають два типи mutation або query. Mutation відповідають за зміну стану програми, а query за запити інформації, різниці між ними логічної нема, лише симантична.

Важливий спосіб виклику цих процедур, коли на frontend йде запит на сервер, можна передавати параметри, вони будуть перетворенні в JSON і на сервері оброблені, для валідації цих параметрів використовується бібліотека zod, при невалідній формі запиту буде повернена помилка валідації. Перевагою такої архітектури є в тому що клієнт може використовувати так само zod, і виводити помилки, тобто досить провалідувати на сервері і показати помилку на клієнті. Це допомагає схильнутися більш до серверу, прибирає необхідність створювати код для валідації форм.

Відокремлено наступні домени:

- Адміністрація
- Каталог
- Завантаження файлів
- Мови
- Експорт
- Замовлення

Для кожного з цих доменів створено власний проєкт, і на сервері доданий для них Fastify плагін. Для них написані маршрути, залежно від частини сайту, деякі методи можуть бути в web або admin маршрутах.

4.1. Авторизація користувача

Реєстрація і авторизація може бути складною річчю, і реалізація авторизація через стандарт OAuth2 може зайняти багато часу і потребує на це підтримки. На томість, використано сторонній сервіс Clerk, через нього можливо реалізувати авторизацію користувача, фронтенд сам має авторизуватися через цю платформу, а вже після передавати ключі на сервер. На кожен реквест з цим токеном буде перевірений через сервіс Clerk, він поверне дані про користувача.

4.2. Створення каталогу

Каталог присутній як на адмінський так й на клієнтських сайтах, варто розділити його логіку на ці дві частини. Для адміністративної панелі потрібні повні CRUD операції, а для публічної можливість фільтрації по різним параметра, таких як: ціна, атрибути, опції, виробник, тощо. При зміні відношень з Composite ключем варто спочатку видаляти всі зв'язні реляції, а потім їх записати наново.

4.3. Завантаження файлів

Фото є невід'ємною частиною продуктів каталогу, для них потрібно створити окремий пакет. Треба врахувати що збережені файли можуть в майбутньому перенесені на іншу платформу, тому варто враховувати різні стратегії деплою. Файли не тільки треба завантажити, а і зберегти інформацію про ни в базу даних, щоб можна були ними потім керувати.

Проблемою стало що tRPC не підтримує метод енкодування multipart/form-data, тому залишається лише передавати файли в форматі base64, після декодування цього файлу повертається Buffer з файлом і зберігається за поточною стратегією збереження.

4.4. Замовлення

Потрібно мати можливість створення і перегляду замовлень поточного користувача в клієнті, та перегляд і редагувань в адмін панель. В адміністративній частини можливо тільки змінювати статус замовлення, товари в замовленні після його оформлення є імутабельними.

4.5. Експорт каталогу

Важливою частиною для просування бізнесу є можливість завантаження фідів. Їх формування відбувається за обраною платформою, вимог для ци фідів є дуже багато, залежно від категорій товарів, вони можуть бути різними, тому варто створити відокремити завантаження товарів з генерацією товарних фідів. Генерація експорту відбувається в функціональній манері через бібліотеку “xmlbuilder2,” в результаті отримуємо згенерований xml файл.

Також каталог може бути експортований автоматично, варто налаштувати експорт товарів за окремим шляхом.

5. Реалізація клієнтських частин

Оскільки на back-end використовуються такі ж технології як і на клієнтних частинах, треба використати цю можливість. Такі сайти здебільшого сформовані для переляду і для заповнення форм, ніж на більш інтерактивному досвіді. Такі випадки покриває tRPC з TanstackRouter, завантаження серверної інформації відбувається до рендеренгу сторінки, логіка завантажень стає більш синхронною

```
4
5  export const Route = createFileRoute(
6    "/option-groups/$optionGroupId/option/new",
7  )({
8    beforeLoad: ({ context }) => ({
9      ...context,
0      getTitle: () => "New option",
1    }),
2    loader: async ({ context }) => {
3      const languages = await context.trpc.admin.language.list.fetch();
4
5      return { languages };
6    },
7    component: New,
8  });
9
0  function New() {
1    const { languages } = Route.useLoaderData();
2    const params = Route.useParams();
3    const navigate = useNavigate();
4    const utils = trpc.useUtils();
5
```

При декларації сторінки обирається потрібний tRPC ресурс і додається в функцію loader, вона відповідає за завантаження стану сторінки, після чого буде згенерована сама сторінка. На відміну від адмін панелі, публічний сайт не може отримати контекст поточного користувача, оскільки завантаження користувача в Clerk відбувається в браузері.

З появою бібліотек на кшталт Redux і інших state-management рішень,

Для роботи з формами використовується бібліотека “react-hook-form,” для неї створена хук обгортка для роботи з сервером, коли користувач заповнює форму, запит приходить на сервер, сервер валідує цю форму і може повернути

помилку валідації. Оскільки на сервері використовується бібліотека zod, її можна використати і на клієнтських частинах, після виникнення помилки валідації є можливість передати цю помилку в “react-hook-form,” і обробити її для показу відповідних помилок форми.

При створенні форм може стати в нагоді і типи серверу, бібліотека tRPC пропонує вивід типів через предикат “inferReactQueryProcedureOptions,”

```
import { AppRouter } from "@repo/server";
import { inferRouterInputs, inferRouterOutputs } from "@trpc/server";

type RouterInputs = inferRouterInputs<AppRouter>;
type RouterOutputs = inferRouterOutputs<AppRouter>;
type AdminOutputs = RouterOutputs["admin"];
type AdminInputs = RouterInputs["admin"];

type LanguageCreateInputs = AdminInputs["language"]["create"];
type LanguageUpdateInputs = Omit<AdminInputs["language"]["update"], "id">;
```

За допомогою такої техніки можна створювати форми які є типізованими, а при зміні параметртів процедур, компілятор Typescript може на ранньому етапі вивести помилку. Форми створення і редагування, де це можливо, створюються в одному компоненті і залежно від сторінки ці форми створюються по різному. Для виклику мутації використовується клієнт tRPC з вибраним шляхом і передаються дані з форми.

Висновок

Дана робота присвячена дослідженню потреб ринку електронної комерції та створення продукту який відповідає умов сучасного бізнесу. В порівнянні з іншими готовими рішеннями розроблено систему стійкою до змін, зокрема, вирішено проблему змін типів клієнтської частини при зміні серверної, змін схем баз даних. Створена архітектура бази даних з урахування вимогам товарних маркетплейсів.

У даній роботі розглянуто екосистему Javascript, створений проєкт відповідає сучасній архітектурі веб додатків, досліджено альтернативу REST - бібліотеку tRPC, та її випадки використання. Створення дослідження зв'язку типів між декількома проєктами.

Основними результатами цієї роботи є:

- Розробка серверної частини з використання фреймворку Fastify і tRPC.
- Розроблено схеми бази даних які підпадають вимогам маркетплейс платформ.
- Створено каталогу товарів.
- Розробка системи експорту товарів.
- Розроблено систему оформлення замовлення.
- Розроблена front-end частина з більш браузер-орієнтовним дизайном.

Список використаних джерел

1. The History of React.js: A Story of Innovation and Community - https://www.linkedin.com/pulse/history-reactjs-story-innovation-community-l-anderson
2. Правила рекламного обслуговування - https://hotline.ua/about/pricelists_specs/
3. What Is tRPC Protocol? - https://www.wallarm.com/what/trpc-protocol
4. What is server-side rendering: definition, benefits and risks - https://solutionshub.epam.com/blog/post/what-is-server-side-rendering
5. Using Turborepo to Build Your First Monorepo - https://earthly.dev/blog/build-monorepo-with-turporepo/
6. TypeScript and the Database: Who Owns the Types? - https://portal.gitnation.org/contents/typescript-and-the-database-who-owns-the-types