МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ЧЕРНОВ АНТОН ОЛЕКСІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
канд. техн. наук, доцент
_____О. В. Зелінська
«____»_____2024 р.

ПРОЄКТУВАННЯ ВЕБСАЙТУ З ЕЛЕМЕНТАМИ ГОЛОСОВОГО ІНТЕРФЕЙСУ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (магістерська) робота

Науковий керівник:
Р.М. Бабаков, професор кафедри
інформаційних технологій,
докт. техн. наук, доцент
_____
(підпис)

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК:_____
(підпис)

Вінниця 2024

АНОТАЦІЯ

Чернов А.О. Проєктування вебсайту з елементами голосового інтерфейсу

Спеціальність 122 «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2024.

Головною метою виконання даної кваліфікаційної роботи є побудова вебсайту із використанням голосовго інтерфейсу. Це зумовлено зростанням попиту на такі програмні продукти.

У вступі акцентовано на актуальності розробки додатків із елементами голосового інтерфейсу.

Перший розділ присвячено аналізу предметної області та розгляду існуючих аналогів з визначенням їхніх переваг і недоліків.

Другий розділ детально розглядає особливостірозробки фронтенд частини додатку, інтерфейсу голосового управління, бекенд частини додатку.

Третій розділ аналізує основні архітектурні рішення, процес побудови бекенд частини додатку, процес побудови фронтенд частини додатку, базового функціоналу інтерфейсу голосового управління.

Ключові терміни: Frontend, Web Speech API, Angular, Material UI, Backend, Mongodb, API.

ABSTRACT

Chernov A.O. Designing a website with voice interface elements

Speciality 122 ‘Computer Science’. Vasyl' Stus Donetsk National University, Vinnytsia, 2024.

The main purpose of this qualification work is to build a website using a voice interface. This is due to the growing demand for such software products.

The introduction emphasises the relevance of developing applications with voice interface elements.

The first section is devoted to analysing the subject area and reviewing existing analogues with the identification of their advantages and disadvantages.

The second section discusses in detail the peculiarities of developing the front-end part of the application, the voice control interface, and the back-end part of the application.

The third section analyses the main architectural solutions, the process of building the backend of the application, the process of building the frontend of the application, the basic functionality of the voice control interface. Keywords: Frontend, Web Speech API, Angular, Material UI, Backend, Mongodb, API.

ЗМІСТ

ВСТУП

У сучасному світі, де технології швидко розвиваються, веб-додатки стали важливим інструментом для бізнесу та повсякденного життя. Зростаюча потреба у швидких, зручних та доступних інтерфейсах спонукає до пошуку нових рішень для підвищення якості взаємодії користувача з додатками. Однією з перспективних технологій є впровадження голосових інтерфейсів, що надають користувачам можливість взаємодіяти з додатками на основі голосових команд. Це особливо актуально у сферах, де важлива швидкість виконання операцій, або ж для користувачів із обмеженими можливостями, які не можуть користуватися традиційними інтерфейсами.

Магістерська робота зосереджена на розробці вебсайту з елементами голосового інтерфейсу, що стане прикладом застосування новітніх веб-технологій та бібліотек для створення інтерактивних клієнт-серверних додатків. Ключовим елементом у побудові вебсайту є браузерний Web Speech API, що надає можливість обробляти голосові команди для забезпечення ефективної та інтуїтивної взаємодії з користувачем.

Об’єктом дослідження є частина Web Speech API – інтерфейс SpeechRecognition. Він дозволяє в режимі реального часу отримувати транскрипцію голосових команд користувача на завчасно заданій мові.

Завданням роботи є створення вебсайту, де буде запроваджений метод взаємодії за допомогою голосових команд.

Головними інструментами побудови такого додатку є мова програмування JavaScript, який нативно виконується у браузерах. Точніше буде використано надбудову над JavaScript - TypeScipt, та фреймворк Angular, що дозволить створити правильний та струкутровнаий, з точки зору парадигми Об’єктно Орієнтованого Програмування додаток.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області

Вебсайти відіграють ключову роль у сучасному цифровому середовищі. Вони слугують основним засобом інформаційної взаємодії, представляючи компанії, надаючи послуги, поширюючи інформацію та забезпечуючи взаємодію з користувачами. Вебсайти можна поділити на декілька основних типів залежно від їх функцій та цілей.

- Інформаційні вебсайти зосереджені на наданні користувачам доступу до різноманітної інформації. Це можуть бути новинні портали, блоги, освітні ресурси та інші джерела, де користувачі можуть отримати необхідні дані. Такі сайти зазвичай структуруються таким чином, щоб полегшити пошук та перегляд інформації.
- Комерційні вебсайти, як-от інтернет-магазини, дозволяють компаніям продавати товари та послуги онлайн. Вони включають каталоги продукції, кошики для покупок та платіжні системи, забезпечуючи зручність покупок для користувачів. Важливою функцією таких сайтів є надання безпечних і надійних способів оплати.
- Корпоративні вебсайти представляють інформацію про компанії, їхню діяльність, продукти та послуги. Вони часто використовуються для залучення нових клієнтів, партнерів та інвесторів, а також для підтримки зв'язку з існуючими клієнтами. Такі сайти можуть включати новини компанії, інформацію про події та контактні дані.
- Соціальні мережі та форуми створюють платформи для спілкування та обміну інформацією між користувачами. Вони дозволяють створювати спільноти за інтересами, обговорювати різноманітні теми та ділитися контентом. Такі сайти значною мірою базуються на користувацькому контенті та взаємодії.

Для ефективного функціонування вебсайтів існує низка вимог, які вони повинні виконувати. Однією з основних вимог є юзабіліті, тобто легкість використання сайту. Інтерфейс має бути інтуїтивно зрозумілим, а доступ до основної інформації – швидким і простим. Доступність вебсайтів також є важливим аспектом. Сайти повинні бути доступними для всіх користувачів, включаючи людей з обмеженими можливостями, що вимагає дотримання стандартів доступності.

Продуктивність вебсайту визначається його швидкістю завантаження сторінок та загальною продуктивністю. Користувачі очікують швидкого доступу до контенту, і повільний сайт може відштовхнути відвідувачів. Безпека є критичним аспектом, особливо для комерційних вебсайтів. Вона включає захист від несанкціонованого доступу та захист персональних даних користувачів.

Адаптивність є ще однією важливою вимогою. Вебсайт має коректно відображатися на різних пристроях, включаючи мобільні телефони та планшети. Це забезпечує зручність користування незалежно від типу пристрою, який використовує відвідувач.

Сучасні тенденції у веб-дизайні спрямовані на покращення користувацького досвіду. Адаптивний дизайн дозволяє сайтам автоматично підлаштовуватися під розмір екрану користувача, забезпечуючи зручний перегляд на будь-якому пристрої. Односторінкові додатки (single-page applications) надають користувачам швидкий та плавний досвід завдяки тому, що всі необхідні ресурси завантажуються одразу.

Однією з найцікавіших інновацій є інтеграція голосових інтерфейсів та інтерактивних помічників. Використання технологій розпізнавання мови та голосових команд дозволяє користувачам взаємодіяти з вебсайтами, не використовуючи клавіатуру або мишу. Це значно полегшує доступ до інформації та виконання завдань, особливо на мобільних пристроях.

Також варто зазначити, що використання голосових інтерфейсів також підвищує рівень доступності продукту. В сьогоденні це питання є неймовірно важливим як в Україні, так і в усьому світі, адже саме таким чином люди із вадами зору, люди з порушенням моторики можуть ефективно використовувати усі доступні функції при правильній реалізації даного підходу.

1.2 Огляд існуючих аналогій

Наразі широкої інтеграції голосових інтерфейсів у веб-сайти немає, проте варто розглянути низку продуктів, які виконують схожі функції.

1. Google Assistant (рис. 1.1)

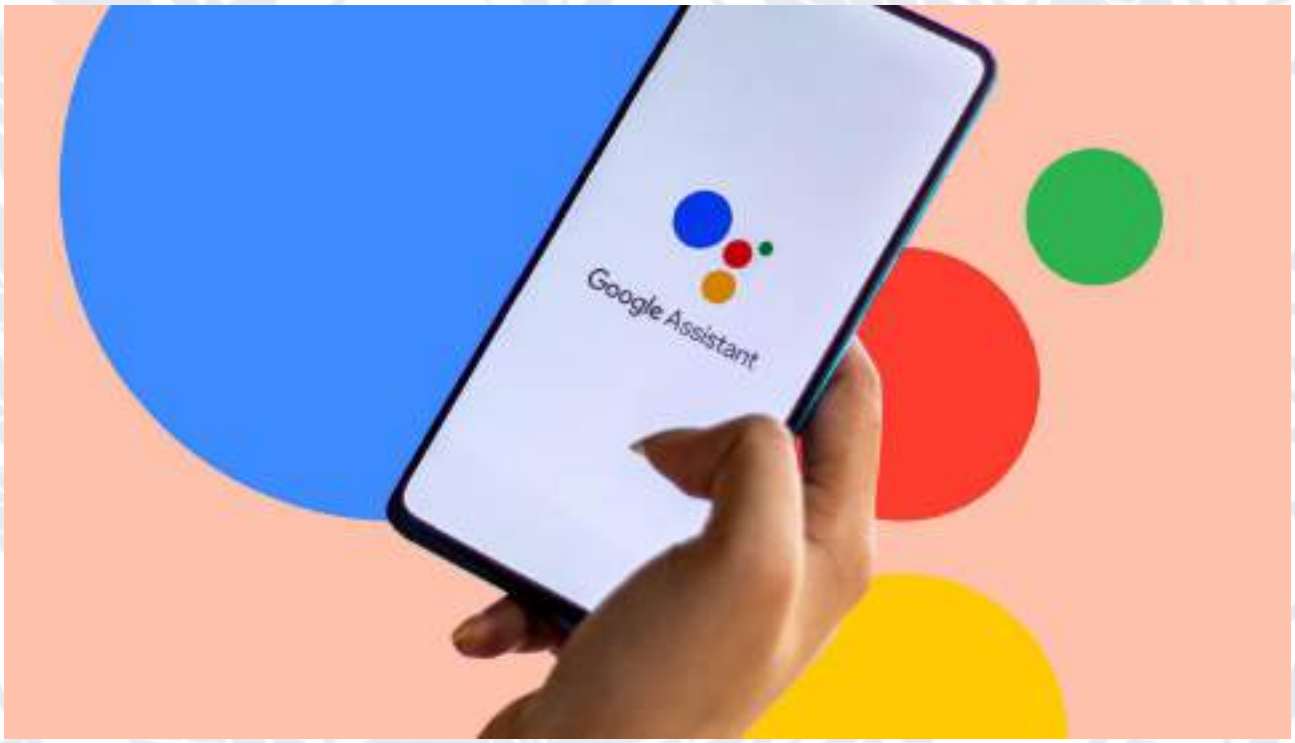


Рис. 1.1 – Система “Google Assistant”

Google Assistant — це голосовий асистент, розроблений компанією Google, який вперше був представлений у 2016 році. Він працює на основі потужних алгоритмів машинного навчання та здатний розпізнавати і відповідати на природні вербальні запити користувачів. Google Assistant інтегрований у широкий спектр пристроїв, включаючи смартфони, смарт-колонки, розумні телевізори та автомобілі. Його основні функції включають:

- Розпізнавання та синтез мови.
- Відповіді на запити користувачів з використанням пошукової системи Google.
- Керування розумним домом (світло, термостати, замки тощо).
- Інтеграція з різними сервісами Google (Gmail, Календар, YouTube).

2. Amazon Alexa (рис.1.2)

Рис. 1.2 – Система “Amazon Alexa”

Amazon Alexa — це голосовий асистент, розроблений компанією Amazon, який вперше з'явився у 2014 році в смарт-колонці Amazon Echo. Alexa має широкий спектр функцій, таких як:

- Розпізнавання та синтез мови.
- Відповіді на запити користувачів з використанням пошукової системи Bing.
- Керування розумним домом (світло, термостати, замки тощо).
- Інтеграція з численними сторонніми сервісами через навички Alexa (Skills).
- Підтримка створення голосових команд для автоматизації завдань.

3. Apple Siri (рис. 1.3)

Рис. 1.3 – Система “Siri”

Apple Siri — це голосовий асистент, розроблений компанією Apple, який вперше з'явився у 2011 році. Siri інтегрований у всі пристрої Apple, включаючи iPhone, iPad, Mac, Apple Watch та HomePod. Основні функції Siri включають:

- Розпізнавання та синтез мови.
- Інтеграція з додатками та сервісами Apple (Apple Music, Apple Maps, iMessage).
- Виконання голосових команд для керування пристроями Apple та розумним домом.
- Підтримка створення нагадувань, подій у календарі, відправки повідомлень.

4. Microsoft Cortana (рис. 1.4)

Рис. 1.4 – Система “Microsoft Cortana”

Microsoft Cortana — це голосовий асистент, розроблений компанією Microsoft, який вперше був представлений у 2014 році. Cortana інтегрований у Windows 10 та інші продукти Microsoft. Основні функції Cortana включають:

- Розпізнавання та синтез мови.
- Інтеграція з сервісами Microsoft (Office 365, Outlook, OneDrive).
- Підтримка пошуку через Bing.
- Керування пристроями розумного дому.

Таким чином ми можемо зробити висновок, що кожен із цих голосових асистентів є дуже корисним в рамках власної екосистеми, проте в питанні використання їх для перегляду та взаємодії із веб-сайтами існують складності. Саме тому використання голосових інтерфейсів в рамках веб-сайту або веб додатку є більш пріоритетним.

1.3 Постановка задачі

В рамках даної роботи задача полягає у створенні простого функціонуючого сайту із можливістю взаємодії з ним із використанням голосового інтерфейсу.

Таким чином буде розроблено веб-додаток з клієнтської та серверної частини у базовий функціонал якого входить:

- Створення, перегляд, редагування та видалення задач. У задачі зазначено опис, виконавець, оцінка складності.
- Створення, перегляд, редагування та видалення відео-зустрічей у форматі посилання на сервіс, де воно буде проводитись

До базового функціоналу буде додано голосовий інтерфейс.

1.4 Алгоритми розпізнавання мови

Алгоритми розпізнавання мови є основою голосових інтерфейсів, дозволяючи комп'ютерам та іншим пристроям інтерпретувати людську мову. Ці алгоритми перетворюють звукові хвилі, що утворюються під час мовлення, на текстові дані, які можуть бути оброблені машиною. Існує кілька ключових методів та підходів, які використовуються для цієї мети, кожен з яких має свої переваги та обмеження.

Одними з перших підходів до розпізнавання мови були статистичні моделі, такі як приховані марковські моделі (Hidden Markov Models, HMM) та гаусові змішувальні моделі (Gaussian Mixture Models, GMM). Ці моделі використовують ймовірнісні методи для визначення послідовностей звуків у мовленні та їх відповідності текстовим представленням.

Приховані марковські моделі (HMM) є одним з найважливіших та найвживаніших інструментів у розпізнаванні мови. Вони використовуються для моделювання послідовностей даних, таких як мовлення, де кожен елемент послідовності залежить від попереднього стану.

HMM є статистичними моделями, які представляють систему як набір прихованих станів, кожен з яких має певну ймовірність переходу до іншого стану та ймовірність генерування спостереження. Формально, HMM визначається такими елементами:

- Станами (S): Множина станів, які може займати система. Ці стани не спостерігаються напряму.
- Вихідними символами (O): Множина спостережуваних символів або ознак.
- Ймовірностями переходів (A): Ймовірності переходу від одного стану до іншого.
- Ймовірностями спостережень (B): Ймовірності генерування спостереження в кожному стані.
- Початковими ймовірностями (π): Ймовірності того, що система починається в кожному стані.

HMM використовуються для вирішення трьох основних завдань:

1. Оцінка ймовірності послідовності спостережень: Обчислення ймовірності того, що дана послідовність спостережень була згенерована моделлю. Це завдання вирішується за допомогою алгоритму прямих і зворотних проходів.
2. Розшифровка (декодування): Визначення найбільш ймовірної послідовності прихованих станів, яка могла згенерувати задану послідовність спостережень. Це завдання вирішується за допомогою алгоритму Вітербі.
3. Навчання моделі: Налаштування параметрів моделі (ймовірностей переходів, спостережень та початкових ймовірностей) на основі даних спостережень. Це завдання вирішується за допомогою алгоритму Баума-Велча, який є окремим випадком алгоритму очікування-максимізації (EM).

Алгоритм Вітербі є динамічним програмуванням, яке дозволяє знайти найімовірнішу послідовність прихованих станів для заданої послідовності спостережень. Він працює шляхом обчислення найкращих шляхів до кожного

стану в кожен момент часу, а потім зворотнім проходом визначає оптимальну послідовність.

Алгоритм Баума-Велча використовується для навчання HMM, максимізуючи ймовірність спостережень даних для заданої моделі. Він складається з двох етапів:

1. Крок очікування (E-step): Обчислення очікувань частот переходів і спостережень, використовуючи поточні оцінки параметрів моделі.
2. Крок максимізації (M-step): Оновлення параметрів моделі для максимізації очікуваних частот, обчислених на попередньому кроці.

HMM широко використовуються для моделювання часових аспектів мовлення. Вони дозволяють ефективно враховувати послідовності звуків і їхню тривалість, що є важливим для точного розпізнавання слів і фраз. У розпізнаванні мови HMM часто комбінуються з іншими моделями, такими як гаусові змішувальні моделі (GMM), для моделювання акустичних ознак.

Гаусові змішувальні моделі (GMM) є важливим інструментом у розпізнаванні мови, оскільки вони дозволяють моделювати складні розподіли даних шляхом комбінації декількох гаусових компонентів. Кожна компонент відповідає за моделювання певної частини даних, що дозволяє точніше описати різноманітні властивості мовних сигналів.

GMM широко використовуються у поєднанні з прихованими марковськими моделями (HMM) для розпізнавання мови. У такій комбінації HMM моделюють послідовність звуків, тоді як GMM використовуються для моделювання ймовірностей спостережень (акустичних ознак) у кожному стані HMM. Цей підхід дозволяє ефективно враховувати часові аспекти мовлення та варіації у вимові.

Web Speech API є однією з основних технологій, яка дозволяє реалізувати розпізнавання мови у вебсередовищі. Вона використовує хмарні сервіси для обробки мовних даних, забезпечуючи високу точність і швидкість розпізнавання.

API підтримує розпізнавання мовлення в режимі реального часу та синтез мовлення, дозволяючи розробникам створювати інтерактивні голосові інтерфейси для вебсайтів.

Алгоритми розпізнавання мови пройшли довгий шлях від простих статистичних моделей до складних нейронних мереж і трансформерів. Сучасні методи забезпечують високу точність і надійність, роблячи голосові інтерфейси все більш популярними. Інтеграція цих технологій у вебсайти за допомогою таких інструментів, як Web Speech API, відкриває нові можливості для взаємодії з користувачами та покращення їхнього досвіду.

1.5 Аналіз інструментів розробки

1.5.1 Вибір технології бекенд частини

NestJS – це прогресивний фреймворк для побудови серверних додатків на Node.js, який використовує TypeScript. Його основна мета – створення ефективних, надійних та масштабованих серверних рішень.(рис.1.5)

NestJS побудований на основі архітектури, яка включає модулі, провайдери, контролери та послуги, що дозволяє легко керувати залежностями та поліпшує читабельність коду. Це допомагає розробникам створювати додатки, які легко масштабуються та підтримуються.

NestJS повністю підтримує TypeScript, що забезпечує сильну типізацію та поліпшену автодопомогу у редакторах коду. TypeScript допомагає уникнути багатьох помилок на етапі розробки та робить код більш надійним і зрозумілим.

NestJS має вбудовану підтримку для роботи з різними базами даних через ORM (Object-Relational Mapping) такі як TypeORM, Mongoose для MongoDB та інші. Крім того, він легко інтегрується з іншими бібліотеками та фреймворками, такими як GraphQL, WebSockets та інші.

NestJS має активну спільноту та добре задокументований API, що дозволяє розробникам швидко знайти відповіді на питання та вирішити проблеми, що

виникають під час розробки. Крім того, велика кількість готових модулів та пакетів дозволяє скоротити час на розробку та зосередитися на бізнес-логіці додатку.

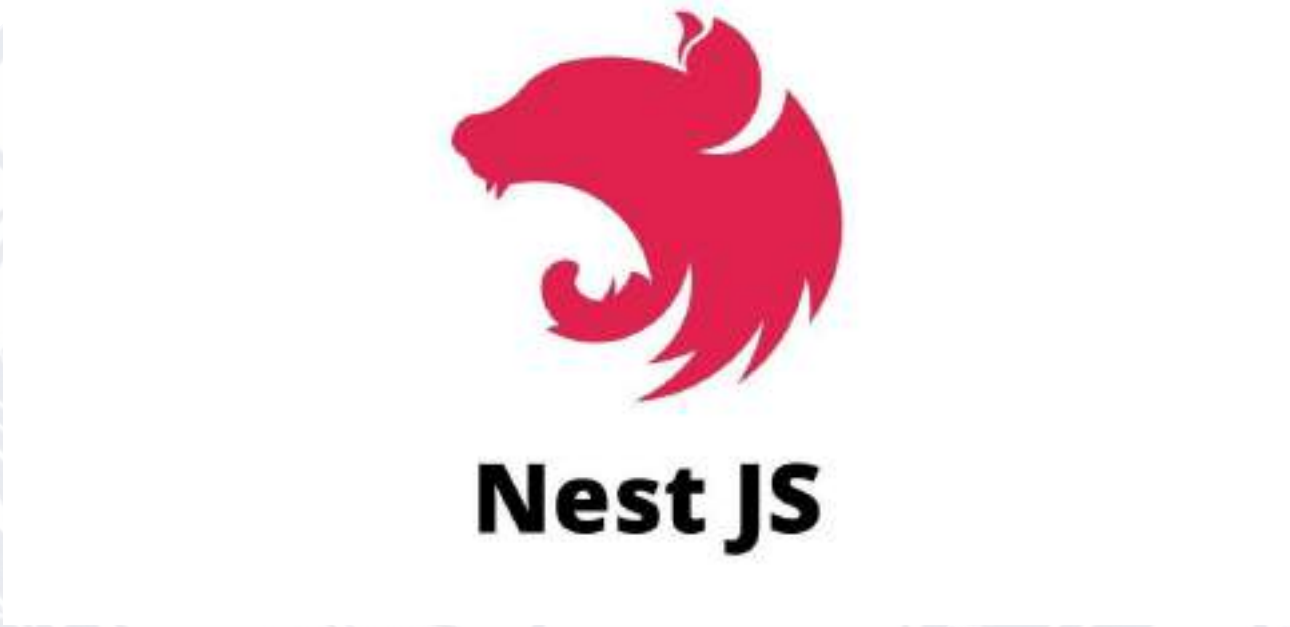


Рис. 1.5 – Фреймворк “NestJS”

Варто зауважити і розгялнуті альтернативи, в більш стислому форматі.

Express.js – це мінімалістичний та гнучкий веб-фреймворк для Node.js, який надає простий і зрозумілий API для створення веб-додатків та API. Його простота у використанні та швидке налаштування дозволяють розробникам швидко почати роботу з проектом. Express.js забезпечує гнучкість у використанні різних компонентів та підходів залежно від потреб проекту, що робить його популярним вибором для багатьох розробників. Крім того, велика кількість доступних модулів та бібліотек, активна спільнота та багатий досвід використання роблять Express.js відмінним вибором для багатьох проектів. Однак, відсутність вбудованої архітектури може ускладнити підтримку великих проектів, і для багатьох функцій потрібне додаткове налаштування.

Django – це високорівневий фреймворк для веб-розробки на Python, який дозволяє швидко створювати безпечні та підтримувані додатки. Django надає багато вбудованих компонентів, таких як аутентифікація, адміністративний інтерфейс та ORM, що значно прискорює процес розробки. Вбудовані засоби захисту від загальних веб-загроз роблять Django одним із найбезпечніших фреймворків. Його всеохоплюючий підхід дозволяє розробникам зосередитися на бізнес-логіці замість вирішення технічних деталей. Django добре підходить для створення великих та складних додатків завдяки своїй структурованості та багатству функцій. Проте, цей фреймворк може бути надмірним для простих додатків і менш гнучким у деяких випадках через свою строгість.

Flask – це мікрофреймворк для Python, який надає лише основні компоненти для створення веб-додатків. Його простота та легкість у використанні дозволяють швидко розпочати розробку проекту. Flask легко інтегрується з іншими бібліотеками та компонентами, що забезпечує гнучкість у виборі технологій та підходів. Мікрофреймворк підходить для невеликих та середніх проектів, де не потрібна складна архітектура. Однак, для створення більш складних додатків з багатьма функціями розробникам може знадобитися додавати багато компонентів вручну, що може ускладнити підтримку та масштабування проекту.

Spring Boot – це фреймворк для створення автономних, виробничих додатків на Java. Він надає багато вбудованих функцій для розробки складних додатків та забезпечує високу масштабованість і надійність. Spring Boot добре підходить для великих корпоративних додатків, де важливі безпека, продуктивність та підтримка багатьох користувачів. Вбудовані засоби захисту та аутентифікації роблять його надійним вибором для безпечних додатків. Проте, високий поріг входу для новачків і вимогливість до ресурсів можуть стати перешкодами для використання Spring Boot у менших проектах або для команд з обмеженими ресурсами.

.NET – це потужна платформа для розробки веб-додатків, створена Microsoft. Вона підтримує кілька мов програмування, включаючи C#, що дозволяє розробникам використовувати знайомі інструменти та технології. ASP.NET Core, зокрема, є крос-платформовою та високопродуктивною версією, яка підходить для створення сучасних, хмарно-орієнтованих додатків. .NET забезпечує високу продуктивність завдяки своїй ефективній обробці запитів та оптимізації ресурсів. Він включає в себе багато вбудованих функцій, таких як аутентифікація, авторизація, обробка помилок та логування, що значно полегшує розробку складних додатків. Велика екосистема бібліотек і інструментів, таких як Entity Framework для роботи з базами даних та інтеграція з Azure, робить .NET відмінним вибором для корпоративних додатків. Проте, використання .NET може вимагати додаткових витрат на ліцензії та інфраструктуру, що варто враховувати при виборі технології.

1.5.2 Вибір технології фронтенд частини

Angular – це популярний фреймворк для розробки односторінкових додатків (SPA), створений та підтримуваний Google. Він надає потужні інструменти для побудови динамічних вебзастосунків з високою продуктивністю.

Angular побудований на модульній архітектурі, що дозволяє розділити додаток на окремі функціональні блоки. Це спрощує управління залежностями, повторне використання коду та тестування.

Однією з основних особливостей Angular є двостороннє прив'язування даних (two-way data binding), яке автоматично синхронізує модель та представлення. Це значно спрощує роботу з формами та іншими інтерактивними елементами інтерфейсу користувача.

Angular використовує компонентний підхід, що дозволяє створювати багаторазові та ізольовані компоненти інтерфейсу. Це робить код більш структурованим та легко підтримуваним.

Angular надає потужні інструменти для розробників, такі як Angular CLI для створення та управління проектами, інструменти для тестування (Karma, Jasmine) та налагодження (Augury) (рис.1.6). Ці інструменти допомагають автоматизувати процес розробки та підвищують продуктивність.

Angular добре підходить для великих проектів завдяки своїй структурованості, багатству функцій та можливостям масштабування. Він дозволяє легко інтегрувати нових розробників у проект та підтримувати високу якість коду.

Рис. 1.6 – Фреймворк “Angular”

Варто зауважити і розгялнуті альтернативи, в більш стислому форматі.

React – це бібліотека для побудови користувацьких інтерфейсів, розроблена Facebook. Основною перевагою React є його компонентний підхід, який дозволяє створювати багаторазові та ізольовані компоненти. Використання Virtual DOM забезпечує високу продуктивність, оскільки React ефективно оновлює та рендерить лише ті компоненти, які змінилися. React також надає можливість управління станом додатка за допомогою бібліотек, таких як Redux або MobX,

що спрощує розробку складних інтерфейсів. Крім того, велика спільнота та екосистема пакетів роблять React гнучким та легко розширюваним. React підходить для проектів різного масштабу, від невеликих додатків до великих корпоративних систем.

Vue.js – це прогресивний фреймворк для створення користувацьких інтерфейсів, відомий своєю простотою та легкістю у використанні. Vue.js поєднує в собі найкращі риси Angular і React, надаючи потужний двосторонній зв'язок даних та компонентний підхід. Його основні переваги включають легку інтеграцію в існуючі проекти, високу продуктивність та зручність у навчанні. Vue.js має добре задокументований API та активну спільноту, що забезпечує швидке вирішення проблем та багатий вибір готових рішень. Vue.js підходить для створення додатків будь-якої складності, від простих компонентів до великих SPA.

Svelte – це сучасний фреймворк для побудови користувацьких інтерфейсів, який відрізняється підходом до компіляції. Замість використання Virtual DOM, Svelte компілює компоненти у високоефективний імперативний код, що безпосередньо оновлює DOM. Це забезпечує високу продуктивність та менший обсяг коду, необхідного для виконання додатка. Svelte пропонує простий та зрозумілий синтаксис, що полегшує навчання та розробку. Незважаючи на відносну новизну, Svelte швидко набирає популярність завдяки своїй ефективності та простоті. Svelte підходить для проектів, де важливі продуктивність та легкість коду.

Ember.js – це потужний фреймворк для розробки амбітних веб-додатків, що надає багато вбудованих функцій та дотримується суворих архітектурних принципів. Ember.js використовує підхід "сильного правила", що забезпечує структурованість коду та полегшує підтримку великих проектів. Він включає такі функції, як автоматичне оновлення шаблонів, багатий набір утиліт для роботи з даними та інструменти для розробників, що допомагають створювати

продуктивні та масштабовані додатки. Ember.js підходить для великих проектів з вимогами до надійності та структурованості.

1.5.3 Вибір технології розпізнавання мови

Web Speech API є потужним інструментом для інтеграції функціональності розпізнавання та синтезу мови у вебдодатки. Вона забезпечує високу точність розпізнавання та підтримку кількох мов, що робить її ідеальним вибором для створення голосових інтерфейсів.

Web Speech API дозволяє розпізнавати мовлення в реальному часі, перетворюючи звукові сигнали на текст. Це особливо корисно для створення голосових команд, пошуку за голосом та інших інтерактивних функцій. API підтримує як безперервний режим розпізнавання, так і розпізнавання окремих фраз.

Крім розпізнавання, Web Speech API також забезпечує синтез мови, що дозволяє генерувати звукові повідомлення з тексту. Це може бути використано для створення аудіовідповідей на запити користувачів, озвучування повідомлень та інших інтерактивних функцій.

Web Speech API легко інтегрується у вебдодатки та може бути використана разом з іншими вебтехнологіями, такими як Angular. Для використання API достатньо кількох рядків коду, що робить її доступною для розробників з різним рівнем досвіду.

Web Speech API підтримується більшістю сучасних браузерів, таких як Chrome, Firefox та Edge, що забезпечує широку доступність функцій розпізнавання та синтезу мови для користувачів. Проте, для повного покриття можливої аудиторії, варто перевірити сумісність з конкретними версіями браузерів та реалізувати альтернативні рішення для непідтримуваних платформ.

Web Speech API надає розробникам потужний інструмент для створення інтерактивних голосових інтерфейсів з мінімальними зусиллями. Вона значно знижує поріг входу для додавання голосових функцій у вебдодатки та відкриває нові можливості для покращення користувацького досвіду.(рис.1.7)

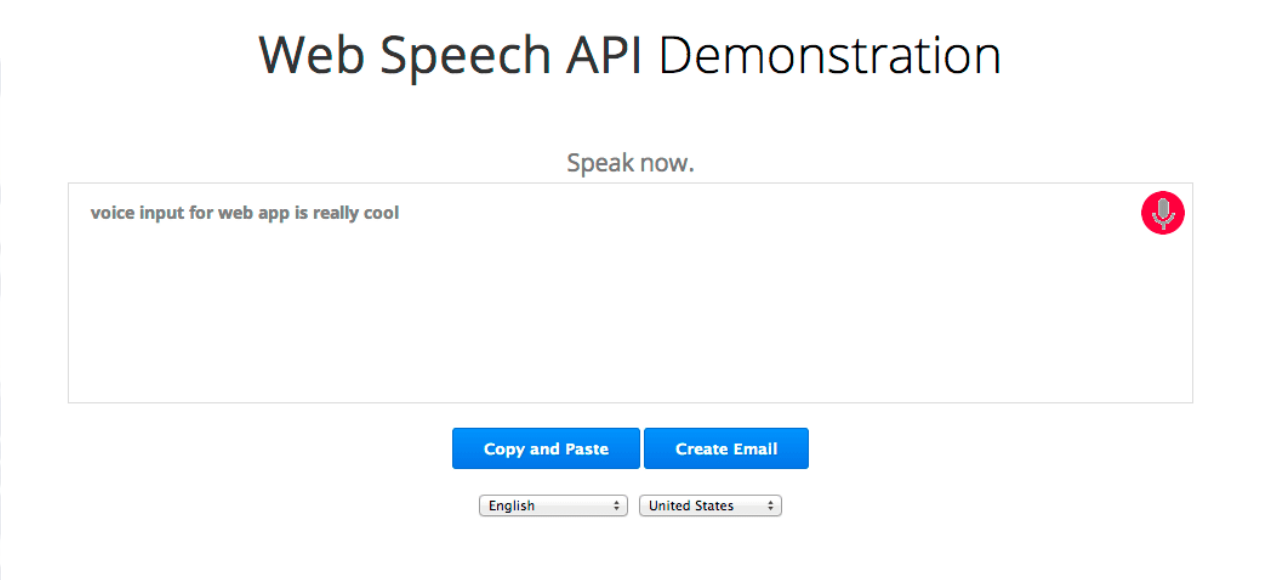


Рис. 1.7 – Технологія “ Web Speech API”

Варто зауважити і розгялнуті альтернативи, в більш стислому форматі.

Google Cloud Speech-to-Text – це потужний сервіс розпізнавання мови, наданий Google Cloud Platform. Він підтримує більше 120 мов та варіантів мовлення, що робить його придатним для глобальних застосувань. Сервіс забезпечує високу точність розпізнавання завдяки використанню передових моделей машинного навчання. Google Cloud Speech-to-Text може обробляти як попередньо записані аудіофайли, так і реальні часи мовлення, що робить його гнучким рішенням для різних сценаріїв використання. Крім того, цей сервіс підтримує спеціальні моделі для окремих галузей, таких як медична або юридична сфера, що підвищує точність розпізнавання для специфічних термінологій. Інтеграція з іншими сервісами Google Cloud дозволяє створювати комплексні рішення для обробки та аналізу аудіоданих.

Microsoft Azure Speech Service – це комплексний набір інструментів для розпізнавання та синтезу мови, наданий Azure. Цей сервіс підтримує більше 70 мов та діалектів, забезпечуючи високу точність розпізнавання завдяки використанню найсучасніших технологій машинного навчання та штучного інтелекту. Azure Speech Service надає різні можливості, включаючи транскрипцію мовлення в текст, переклад мовлення та синтез мови з тексту. Особливою перевагою є можливість налаштування моделей розпізнавання мови для конкретних сценаріїв, що дозволяє підвищити точність розпізнавання в специфічних галузях. Інтеграція з іншими сервісами Azure, такими як Azure Cognitive Services, дозволяє створювати комплексні рішення для аналізу та обробки мовних даних.

IBM Watson Speech to Text – це потужний сервіс для розпізнавання мови, що входить до складу IBM Watson. Він підтримує кілька мов і надає високу точність розпізнавання завдяки використанню потужних алгоритмів машинного навчання. Сервіс дозволяє обробляти як аудіофайли, так і потоки реального часу, що робить його гнучким для різних сценаріїв використання. Watson Speech to Text підтримує спеціалізовані моделі для різних галузей, таких як медична, юридична або технічна, що підвищує точність розпізнавання для специфічних термінологій. Крім того, сервіс надає можливості для автоматичного розпізнавання акцентів та діалектів, що забезпечує більш точне розпізнавання для глобальних користувачів. Інтеграція з іншими сервісами IBM Watson дозволяє створювати комплексні рішення для обробки та аналізу мовних даних.

1.5.4 Обрані технології

Вибір інструментів розробки є критично важливим для успішної реалізації проекту. NestJS забезпечує надійну та масштабовану бекенд архітектуру, Angular надає потужні інструменти для створення інтерактивних фронтенд додатків, а Web Speech API дозволяє інтегрувати передові функції розпізнавання та синтезу мови. Ці технології разом створюють сильну основу для побудови сучасного

вебсайту з елементами голосового інтерфейсу, забезпечуючи високу продуктивність, гнучкість та зручність для користувачів.

В якості бази даних для проекту бело обрано MongoDB. MongoDB – це сучасна нереляційна база даних, яка використовує документно-орієнтовану модель даних. Вона зберігає дані у форматі JSON-подібних документів, що забезпечує високу гнучкість і масштабованість. Основною перевагою MongoDB є її здатність обробляти великі обсяги даних і забезпечувати швидкий доступ до них завдяки індексації та розподіленій архітектурі. Це робить MongoDB ідеальним вибором для веб-додатків, які потребують динамічного управління даними та високої продуктивності. MongoDB також підтримує горизонтальне масштабування за допомогою шардингу, що дозволяє легко розподіляти дані на декількох серверах. Крім того, MongoDB надає потужні засоби для агрегації даних, що дозволяє виконувати складні аналітичні запити. Інтеграція з популярними фреймворками та бібліотеками, такими як Mongoose для Node.js, спрощує розробку та управління базою даних. Велика спільнота користувачів та добре задокументовані ресурси роблять MongoDB надійним вибором для багатьох проектів, від невеликих стартапів до великих корпоративних додатків.

РОЗДІЛ 2
ТЕОРЕТИЧНА ЧАСТИНА РОЗРОБКИ

2.1 Особливості розробки фронтенд частини

Однією з ключових особливостей Angular є компонентний підхід. Кожен компонент є самодостатньою одиницею, яка має свою логіку, стиль і шаблон для відображення. Це забезпечує можливість перевикористання коду і розділення логіки додатка на окремі частини.

Основні елементи компонента в Angular:

- Шаблон (Template): HTML-код, який відповідає за відображення даних. Він може містити директиви Angular, такі як *ngFor або *ngIf, що дозволяють контролювати логіку відображення елементів на сторінці.
- Логіка (Class): Логіка компонента пишеться на TypeScript і містить властивості та методи, які використовуються в шаблоні. Це забезпечує зв'язок між даними та їх відображенням.
- Стилі (Styles): CSS або SCSS стилі, що застосовуються лише до цього компонента. Це дозволяє ізолювати стилі між компонентами, уникаючи конфліктів.

Angular – компонет визначається за допомогою декоратору @Component. Він додає метадані, які вказують Angular, що конкретний клас є компонентом, а також надає конфігурацію для його відображення в шаблоні. @Component виступає декоратором, який повідомляє Angular, що, для прикладу, клас AppComponent є компонентом, і додає метадані, такі як selector (ім'я тега, яке використовується в HTML для виклику компонента), templateUrl (шаблон HTML) та styleUrls (шляхи до файлів стилів).

Патерн Декоратор у програмуванні дозволяє динамічно додавати нову функціональність до об'єктів без зміни їхньої структури або створення підкласів. У контексті Angular, цей патерн реалізовано за допомогою декораторів —

спеціальних функцій, які додають метадані до класів, методів, властивостей або параметрів. Ці метадані дозволяють Angular керувати компонентами, сервісами, директивами та іншими сутностями у вашому додатку.

Декоратори — це ключовий механізм Angular, який використовується для:

- Опису компонентів (@Component)
- Інжекції залежностей (@Injectable)
- Оголошення модулів (@NgModule)
- Створення директив (@Directive)
- Інших дій, що пов'язані з управлінням класами та їх поведінкою.

Інжекція залежностей — це патерн проектування, який дозволяє передавати необхідні об'єкти (залежності) в клас замість того, щоб клас сам створював ці об'єкти. В Angular цей патерн застосовується для керування залежностями між компонентами, сервісами та іншими сутностями.

Цей підхід має кілька ключових переваг:

- Модульність: Класи залишаються ізольованими і не мають жорстких зв'язків із зовнішніми залежностями.
- Тестованість: Легко тестувати класи, оскільки можна передавати макети (mock) залежностей.
- Гнучкість: Залежності можна змінювати або оновлювати без зміни основного класу.

Angular використовує власний механізм інжектора, який відповідає за створення та управління екземплярами сервісів і їхнє впровадження в компоненти. Це відбувається через конструктор класу, де оголошуються залежності, або за допомогою функції inject(), яка повинна бути викликана в контексті інжектування, тобто в конструкторі, або явно задана. Angular автоматично створює екземпляр цього сервісу та передає його компоненту.

Залежно від місця реєстрації сервісу, Angular визначає його scope (область видимості):

- root (корінь додатку): Сервіс доступний у всьому додатку і буде єдиним екземпляром у всіх компонентах (singleton).
- модуль: Сервіс доступний лише в певному модулі, де він зареєстрований.
- компонент: Сервіс створюється та знищується разом із компонентом.

Незважаючи на всі переваги, підхід з інжекцією залежностей може створювати певні виклики.

Іноді може виникати спокуса інжектувати занадто багато залежностей в один компонент, що робить його складним для тестування та підтримки. Це порушує принцип Single Responsibility (єдина відповідальність), коли кожен клас чи компонент має виконувати лише одну задачу. У такому випадку компонент стає занадто залежним від зовнішніх сервісів, що робить його складним для тестування та підтримки.

Деякі сервіси можуть потребувати управління життєвим циклом. Наприклад, якщо сервіс створюється на рівні компонента, то після його знищення сервіс також зникає, що може бути небажаним, якщо він містить важливі дані.

У випадку, коли сервіс зареєстрований на рівні root, він стає singleton (одиночним екземпляром). Це корисно, коли сервіс повинен бути єдиним на весь додаток (наприклад, сервіс аутентифікації). Однак це може створювати проблеми, якщо різні частини додатку потребують своїх окремих екземплярів сервісу для унікальних сценаріїв.

Інжекція через конструктор є очевидною, але інколи можуть бути приховані залежності, що ускладнює розуміння взаємодії між різними частинами

системи. Наприклад, якщо сервіс залежить від іншого сервісу, це може бути неочевидно на перший погляд.

Базово розібравшись із поняттями компоненту та сервісу, визначимо як повинен прцювати функціонал із голосовим інтерфейсом. В контексті даної роботи буде розглянута взаємодія із додатком та його окремими частинами з використанням голосових команд. Для цього потрібно обрати шлях, за допомогою якого команди будуть об’явлені та оброблені.

Розглянемо кілька патернів та оберемо найбліш доречний

Патерн Стратегія (Strategy) — це шаблон проектування, який дозволяє визначати сімейства алгоритмів, інкапсулювати їх та робити взаємозамінними. Іншими словами, патерн надає можливість вибирати алгоритм на етапі виконання програми, не змінюючи код компонентів, що його використовують. У контексті Angular, цей патерн може бути дуже корисним для реалізації різних наборів команд або дій залежно від компонентів, які використовуються у додатку.

Патерн Стратегія дозволяє виділити змінювані частини логіки додатка в окремі класи, які можна легко замінювати на інші, не змінюючи основну структуру коду. Він дозволяє інкапсулювати кілька варіантів поведінки або дій у різні стратегії та використовувати ту, яка найбільш підходить в конкретній ситуації.

Структура патерна:

- Інтерфейс Стратегії: Описує методи, які мають реалізувати всі стратегії.
- Конкретні Стратегії: Реалізують інтерфейс та надають різні варіанти поведінки.
- Контекст: Зберігає посилання на стратегію та викликає її методи.

У випадку проєкту з голосовим інтерфейсом, існує можливість використовувати патерн Стратегія для управління різними наборами команд

залежно від компонентів або контексту використання. Наприклад, один компонент може обробляти команди для навігації по сайту, а інший — для керування календарем або іншим функціоналом.

Переваги:

- Заміна логіки без зміни компонентів: Ви можете змінювати алгоритми або набори команд без необхідності змінювати логіку самого компонента або сервісу.
- Легкість тестування: Окремі стратегії можна легко тестувати незалежно одна від одної.
- Гнучкість і масштабованість: Можна додавати нові стратегії без змін основного коду. Це особливо корисно у великих додатках, де необхідно обробляти різні види команд.
- Зрозумілість і підтримуваність: Патерн розділяє різні аспекти поведінки додатку, що підвищує зрозумілість коду та спрощує його підтримку.

Недоліки:

- Збільшення кількості класів: Для кожної стратегії потрібно створювати окремий клас, що може збільшувати обсяг коду.
- Вибір стратегії на етапі виконання: Якщо логіка вибору стратегії стає складною, можна випадково ускладнити код або зробити його менш очевидним.
- Підтримка багатьох стратегій: Якщо кількість стратегій зростає, необхідно ретельно слідкувати за тим, щоб всі вони були належним чином керовані та інтегровані у додаток.

Патерн Команда, фокусується на інкапсуляції запиту або дії у вигляді об'єкта. Кожна команда представляє одну дію, яку можна викликати, зберігати або передавати, що особливо корисно в системах з можливістю скасування дій.

Принцип роботи:

- Створюється об'єкт-команда для кожної окремої дії.
- Команди можуть виконуватися, скасовуватися або передаватися до іншого об'єкта.
- Використовується, якщо необхідно виконувати конкретні дії або якщо необхідно впровадити логіку скасування (undo).

Створюються команди як окремі класи, що реалізують інтерфейс Command. У класі-команді визначаються методи execute і undo (якщо потрібно).

Переваги:

- Можливість скасування команд.
- Легко розширюється новими командами.

Недоліки:

- Велика кількість класів-команд, якщо існує багато різних дій.
- Зростає складність, якщо додається багато логіки для скасування.

Фабричний метод – це патерн, що дозволяє створювати об'єкти (наприклад, стратегії або команди) через спеціальні фабрики, залежно від умов. Це допомагає уникнути жорсткого зв'язування коду з конкретними реалізаціями класів і полегшує динамічний вибір необхідної логіки.

Принцип роботи:

- Існує фабричний клас або метод, який вирішує, яку стратегію (або команду) потрібно створити і повертає її.
- Використовується, коли логіку вибору реалізації потрібно інкапсулювати в окремому класі.

Переваги:

- Позбавляє клас від прив’язки до конкретних класів продуктів.

- Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
- Спрощує додавання нових продуктів до програми.
- Реалізує принцип відкритості/закритості.

Недоліки:

- Може призвести до створення великих паралельних ієрархій класів, адже для кожного класу продукту потрібно створити власний підклас творця.

В даному додатку оберемо патерн стратегія, оскільки кожна окрема частина додатку матиме власний набір команд, які мають сенс виконання лише при взаємодії з ними.

Варто також зазначити, що команди потрібно виконувати за допомогою асинхронності. Асинхронність в JavaScript є ключовим аспектом, що дозволяє виконувати код не блокуючи основний потік (головний потік) програми. Основні механізми асинхронності в JavaScript включають колбеки (callback), проміси (promises) і асинхронні функції (async/await). Далі наведено детальний опис цих механізмів.

JavaScript працює в однопотоковому середовищі, що означає, що він може виконувати лише один блок коду одночасно. Це пов’язано з його архітектурою, де всі команди виконуються в одному потоці, а асинхронні операції обробляються через чергу.

Коли JavaScript виконує код, він використовує стек викликів (call stack) для зберігання інформації про те, що виконується в даний момент. Коли функція викликається, вона додається до стека, а коли вона завершує виконання, вона видаляється зі стека. Якщо в коді є асинхронні операції, вони не блокують виконання інших функцій.

Колбеки — це функції, які передаються як аргументи до інших функцій і виконуються після завершення асинхронної операції. Недолік колбеків полягає в тому, що при великій кількості вкладених колбеків код стає важким для читання (так званий "callback hell").

Проміси — це об'єкти, що представляють завершення або невдачу асинхронної операції. Вони мають три стани: pending (очікування), fulfilled (виконано) і rejected (відхилено). Проміси значно покращують читабельність коду та полегшують обробку помилок.

Async/await — це синтаксичний цукор, що дозволяє писати асинхронний код у стилі синхронного. Функція, позначена ключовим словом async, автоматично повертає проміс. Ключове слово await використовується для "очікування" завершення проміса.(рис.2.1)

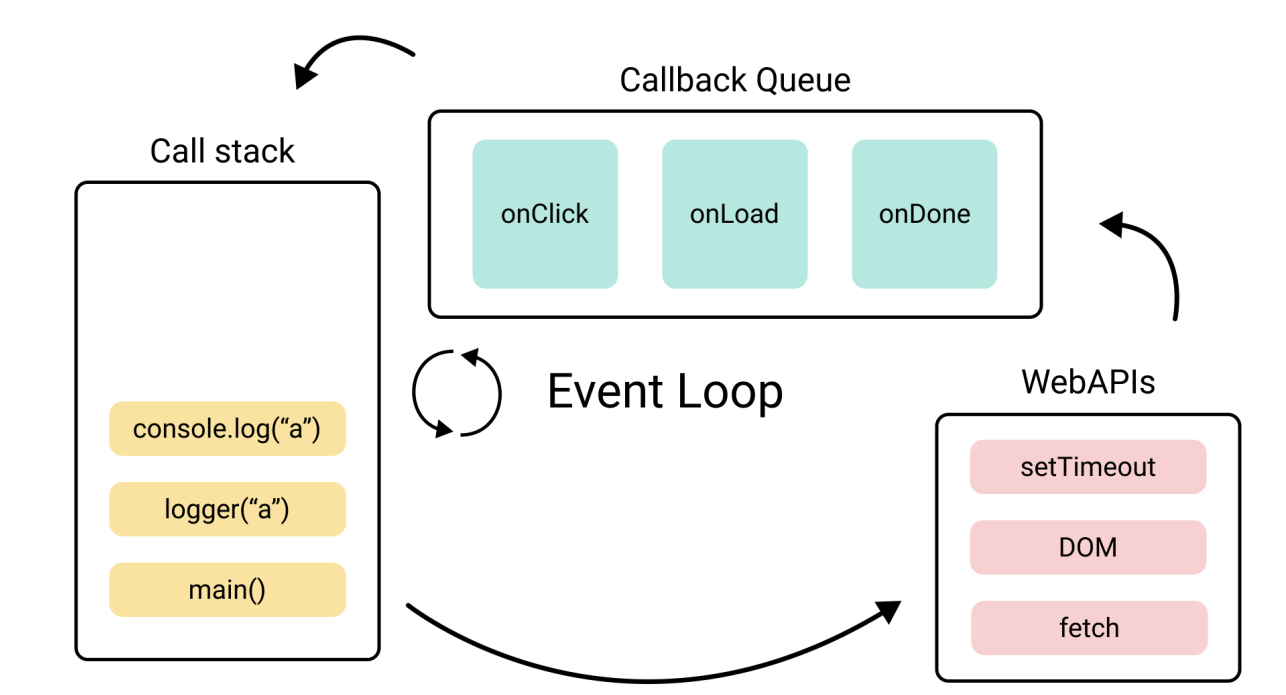


Рис. 2.1 – Система "Event Loop"

JavaScript має цикл подій, який контролює виконання асинхронних завдань. Коли асинхронна операція завершується, її колбек або проміс ставиться в чергу подій. Цикл подій постійно перевіряє, чи є в стеку викликів вільне місце, і якщо так, він виконує перший обробник подій у черзі.

Окрім вбудованих рішень, має сенс розглянути бібліотеку RxJs.(рис.2.2)

Рис. 2.2 – Бібліотека “RxJs ”

RxJS (Reactive Extensions for JavaScript) — це бібліотека для реактивного програмування з використанням спостережуваних потоків даних (observables), яка є важливою частиною Angular. Вона дозволяє працювати з асинхронними операціями (наприклад, HTTP-запити, користувацькі події або вебсокети) в декларативний спосіб, що робить код чистим, гнучким і легко підтримуваним.

Реактивне програмування — це парадигма програмування, яка базується на роботі з потоками даних, що змінюються з часом, і асинхронними подіями. У такому підході ви працюєте не з окремими значеннями, а з послідовностями значень, які можуть бути отримані в майбутньому. Цей підхід робить роботу з подіями та асинхронними процесами інтуїтивнішою та ефективнішою.

Observable — це основний будівельний блок RxJS, що представляє потік даних або подій. Це може бути будь-яка послідовність значень: події користувача (наприклад, кліки миші), відповіді від HTTP-запитів або навіть дані з WebSocket.

Observable може емінувати кілька значень і завершити роботу, або залишатися відкритим і емінувати нові значення з часом.

Observer — це об'єкт, що містить три основні методи:

- next: викликається, коли Observable емінує нове значення.
- error: викликається, коли відбувається помилка.
- complete: викликається, коли Observable завершив емінування значень.

Щоб почати отримувати значення від Observable, необхідно підписатися на нього. Підписка викликає внутрішню логіку Observable і дозволяє отримувати еміновані дані.

Оператори — це функції, які дозволяють перетворювати, фільтрувати, комбінувати, або іншим чином обробляти потоки даних. Вони нагадують функції масиву на кшталт map, filter, але працюють з потоками даних.

Найпоширеніші оператори RxJS:

- map: перетворює кожне значення Observable.
- filter: пропускає тільки ті значення, що відповідають умові.
- mergeMap / switchMap: дозволяють обробляти асинхронні потоки і зливати їх у єдиний Observable.
- debounceTime: затримує видачу значень на певний час.
- catchError: обробляє помилки в Observable.

Також варто зауважити, що існує також класс Subject. Subject — це об'єкт, який реалізує інтерфейс Observable, але також є Observer, що означає, що ви можете підписатися на нього, а також надсилати значення або повідомлення.

Основні характеристики:

- Багатопотоковість: Subject може мати кілька підписників, які отримують однакові значення.

- Негайна емітація: На відміну від звичайних об'єктів Observable, які емітують значення тільки після підписки, Subject може емітувати значення в будь-який момент.
- Стан: Subject не зберігає жодного значення — лише те, що було надіслано, буде доступно для нових підписників.

Існує кілька типів Subject, а саме:

- BehaviorSubject: Зберігає останнє значення і надає його новим підписникам. Вимагає початкового значення.
- ReplaySubject: Зберігає кілька останніх значень і надає їх новим підписникам. Кількість значень, які зберігаються, можна налаштувати.
- AsyncSubject: Емітує значення тільки після завершення потоку, передаючи останнє значення всім підписникам.

RxJS може використовуватися для роботи з подіями DOM у компонентах Angular. Наприклад, за допомогою операторів можна легко реалізувати функціонал debounce для затримки обробки подій.

Angular підтримує реактивні форми, які також використовують Observable для відстеження змін у формах і їхнього стану.

Таким чином, будемо використовувати бібліотеку RxJs для асинхронності.

Для розробки фронтенд частини також було обрано бібліотеку компонентів MaterialUI. Вибір Material UI для розробки додатку зумовлений рядом переваг, які роблять цю бібліотеку ідеальним інструментом для створення сучасного інтерфейсу користувача. Material UI заснована на концепціях дизайну Google Material Design, що надає готові компоненти з уніфікованим стилем, які легко адаптуються до різних платформ. Це дозволяє забезпечити естетичний та функціональний інтерфейс, який відповідає сучасним стандартам, зменшуючи витрати часу на розробку власних стилів та компонентів.

Однією з ключових переваг Material UI є її висока інтеграція з Angular. Компоненти бібліотеки легко вписуються у структуру Angular-додатків, забезпечуючи чисту і зрозумілу архітектуру. Наприклад, використання готових компонентів, таких як таблиці, модальні вікна чи форми, дозволяє зосередитися на бізнес-логіці, а не витрачати час на розробку базових елементів з нуля. Крім того, бібліотека підтримує темізацію, що робить її зручною для створення кастомізованих інтерфейсів.

Ще однією перевагою Material UI є її відмінна документація та активна спільнота. Це знижує криву навчання для розробників і забезпечує швидке вирішення можливих проблем. Завдяки документації, можна швидко знайти приклади використання компонентів, оптимальні рішення для інтеграції або навіть готові шаблони для складних інтерфейсів. Така підтримка значно пришвидшує розробку та мінімізує потенційні помилки.

Нарешті, важливою характеристикою Material UI є її відповідність принципам доступності (accessibility). Вбудовані компоненти оптимізовані для роботи з екранними читачами та іншими технологіями допомоги, що робить додаток доступним для більш широкої аудиторії. У контексті сучасних вимог до веб-додатків це значно підвищує їх конкурентоспроможність та задовольняє законодавчі норми щодо інклюзивності.

2.2 Особливості роботи з Web Speech API

Web Speech API — це API, яке дозволяє веб-додаткам взаємодіяти з голосовим вводом і виводом. Він складається з двох основних компонентів: Speech Recognition (розпізнавання мови) та Speech Synthesis (синтез мови). Цей API є частиною специфікації HTML5 і підтримується більшістю сучасних браузерів.

Ми будемо використовувати Speech Recognition (Розпізнавання мови), цей компонент дозволяє веб-додаткам розпізнавати голосові команди та перетворювати їх на текст.

Основні можливості:

- Розпізнавання голосу в реальному часі: Можливість обробки голосу, коли користувач говорить, з мінімальними затримками.
- Підтримка мов: Доступні різні мови, залежно від підтримки браузера.
- Події: API підтримує різні події, такі як start, end, result, error, які допомагають розробникам управляти процесом розпізнавання.

Алгоритм роботи системи Speech Recognition (розпізнавання мови) включає кілька етапів, які дозволяють перетворювати звукові сигнали у текст.

На початку, коли користувач говорить, мікрофон записує звуковий сигнал, який є аналоговим. Цей сигнал передається в цифровому вигляді для подальшої обробки.

Процес переобробки звукового сигналу:

1. Фільтрація
   - Фонові шуми: Видалення небажаних фонового шуму, що може спотворити результати розпізнавання.
   - Нормалізація гучності: Збалансування гучності сигналу для покращення якості розпізнавання.
2. Аліасінг (зниження частоти)
   Перетворення звукового сигналу з - високої частоти (наприклад, 44.1 кГц) на нижчу частоту (наприклад, 16 кГц) для зменшення обсягу даних і підвищення швидкості обробки.

Акустичне моделювання полягає в поділі звукового сигналу на короткі сегменти, звані фреймами (зазвичай 20-40 мілісекунд). Використання методів, таких як Мель-частотні кепстральні коефіцієнти (MFCC), для витягування характерних ознак з кожного фрейму звукового сигналу, дозволяє зосередитися на найбільш важливих частинах сигналу, які визначають його звучання.

Далі йде крок моделювання мови:

- Лексичне моделювання: Створення словника, що містить усі слова, які може розпізнати система. Кожне слово представляється у вигляді фонетичних транскрипцій.
- Моделювання послідовностей: Використання мовних моделей (наприклад, n-grams) для оцінки ймовірності послідовностей слів. Це допомагає системі вибрати найбільш ймовірне слово з можливих варіантів на основі контексту.

Найбільш цікавим можна назвати етап розпізнавання. Звучання кожного фрейму порівнюється з ознаками слів у словнику. Використовуються алгоритми, такі як вірогіднісний метод (наприклад, Baum-Welch), для оцінки ймовірності того, що кожний фрейм відповідає певному слову.

Алгоритм Baum-Welch — це метод, який використовується для тренування скритих марківських моделей (Hidden Markov Models, HMM) на основі неструктурованих спостережуваних даних. Алгоритм є варіантом алгоритму очікування-максимації (Expectation-Maximization, EM) і дозволяє оцінювати параметри моделі, такі як ймовірності переходів і емісій, шляхом максимізації правдоподібності спостережуваних даних.

Алгоритм Baum-Welch складається з двох основних етапів: очікування (E-step) та максимізації (M-step):

1. Початкові параметри моделі (A, B, π) ініціалізуються випадковими значеннями або значеннями, отриманими з попередніх досліджень.
2. На етапі очікування обчислюється ймовірність перебування в кожному зі станів для кожного спостереження за допомогою алгоритму Віттака.

   Розраховуються два основні значення:

   - α-алгоритм: Визначає ймовірність спостереження до моменту t в стані i.

   - β-алгоритм: Визначає ймовірність залишити стан і та спостерігати

решту спостережень до моменту T.

Ці значення використовуються для обчислення:

- γ_t(i): Ймовірність, що в момент часу t модель знаходиться в стані i.

- ξ_t(i, j): Ймовірність, що в момент часу t модель переходить з стану i в стан j.

3. На етапі максимізації оновлюються параметри моделі на основі ймовірностей, отриманих на попередньому етапі.
4. Етапи E-step і M-step повторюються доти, поки зміни в параметрах не стануть меншими за заздалегідь визначений поріг, або поки не досягнеться максимальна кількість ітерацій.

Найважливішої подією, в контексті нашого додатку у екземпляра класу Speech Recognition API є подія result. Подія result викликається щоразу, коли система успішно розпізнає частину звуку, яка представляє собою текст. Це може статися для коротких фраз або під час безперервного розпізнавання, при необхідному налаштуванні, а саме interimResults = true.

Подія result передає об'єкт події, який містить результат розпізнавання. Основним елементом є масив results, який складається з об'єктів, що містять текст та інші дані.

Кожен об'єкт у масиві results містить три важливі поля:

1. isFinal: Булеве значення, яке вказує, чи є результат остаточним (true) чи проміжним (false).
2. transcript: Текст, який був розпізнаний. Це є текстова репрезентація вимовленого звуку.
3. confidence: Числове значення, яке відповідає за впевненість у транскрибованому тексті.

Подія result є критично важливою частиною механізму розпізнавання мови в Web Speech API, оскільки вона дозволяє отримати та обробити текст, який був

успішно розпізнаний. Розуміння цієї події та її обробки допомагає створювати ефективні голосові інтерфейси для користувачів.

2.3 Особливості розробки бекенд частини

Оскільки додаток побудований із більшим акцентом на фронтенд частину, то про осболивості роботи із бекендом інформації буде менше.

NestJS — це прогресивний фреймворк для побудови ефективних і надійних серверних додатків на основі Node.js. Він побудований на базі TypeScript і використовує модульну архітектуру, що робить його дуже гнучким та масштабованим. У контексті розробки веб-додатка з елементами голосового інтерфейсу, NestJS пропонує ряд особливостей, які можуть суттєво спростити реалізацію функціональності. Ось ключові аспекти, які варто врахувати:

NestJS розподіляє код на модулі, що дозволяє розробникам структурувати проект у зручний для читання та підтримки спосіб. Кожен модуль може містити контролери, провайдери та сервіс, що робить його самодостатнім.

NestJS підтримує потужний механізм ін'єкції залежностей, що полегшує управління сервісами і компонентами. Це особливо важливо для реалізації складних бізнес-логік і забезпечення гнучкості.

NestJS має потужні механізми для валідації даних за допомогою бібліотеки класів-валідаторів. Це дозволяє перевіряти коректність голосових команд, які надходять від користувачів.

NestJS використовує декоратори для визначення контролерів, маршрутів та інших компонентів. Це робить код більш зрозумілим і легким для читання.

Для обробки HTTP-запитів, що надходять від клієнтів, можна створити контролери, які оброблятимуть команди, надіслані користувачами через голосовий інтерфейс.

Контролери в NestJS відповідають за обробку вхідних запитів до API. Вони отримують запити, виконують відповідну логіку і повертають відповіді. У

контексті REST API контролери керують ресурсами, такими як користувачі, продукти або замовлення, і забезпечують необхідні методи для CRUD (Create, Read, Update, Delete) операцій. Щоб створити контролер у NestJS, використовуйте декоратор @Controller(рис. 2.3).

```
import { Controller, Get, Post, Body, Param, Put, Delete } from '@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto, UpdateUserDto } from './dto';

@Controller('users')
export class UsersController {
    constructor(private readonly usersService: UsersService) {}

    @Post()
    create(@Body() createUserDto: CreateUserDto) {
        return this.usersService.create(createUserDto);
    }

    @Get()
    findAll() {
        return this.usersService.findAll();
    }

    @Get(':id')
    findOne(@Param('id') id: string) {
        return this.usersService.findOne(id);
    }

    @Put(':id')
    update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
        return this.usersService.update(id, updateUserDto);
    }

    @Delete(':id')
    remove(@Param('id') id: string) {
        return this.usersService.remove(id);
    }
}
```

Рис. 2.3 – Приклад контролера

Основні компоненти контролера:

- Декоратор @Controller('users'): Вказує, що цей контролер обробляє запити, які надходять на шлях /users.

- Метод create: Обробляє POST запити на /users. Використовується декоратор @Body(), щоб отримати дані, які передає користувач.
- Метод findAll: Обробляє GET запити на /users для отримання списку всіх користувачів.
- Метод findOne: Обробляє GET запити на /users/:id, де :id — це параметр маршруту. Декоратор @Param('id') отримує значення параметра id.
- Метод update: Обробляє PUT запити на /users/:id для оновлення інформації про користувача. Використовується декоратор @Body() для отримання даних для оновлення.
- Метод remove: Обробляє DELETE запити на /users/:id для видалення користувача.

Для валідації і типізації даних, що передаються в запитах, рекомендується використовувати DTO (Data Transfer Object).

Контролер зазвичай взаємодіє зі службою (service), яка містить бізнес-логіку та маніпуляції з даними.

Створення контролерів для REST API у NestJS є простим і зрозумілим процесом завдяки використанню декораторів, структурованої архітектури та вбудованої підтримки валідації. Це дозволяє розробникам швидко і ефективно розробляти серверні додатки, які взаємодіють із клієнтами, виконуючи операції з ресурсами. Контролери можуть бути легко розширені для підтримки нових функціональностей, що робить NestJS відмінним вибором для сучасних веб-додатків.(рис.2.4)

Рис. 2.4 – база даних “MongoDB”

MongoDB — це документно-орієнтована NoSQL база даних, яка дозволяє зберігати дані у форматі JSON-подібних документів. Вона забезпечує високу гнучкість і масштабованість, що робить її ідеальним вибором для сучасних веб-додатків. У контексті розробки веб-додатка з елементами голосового інтерфейсу, MongoDB може використовуватися для зберігання та управління даними, такими як користувацькі налаштування, історія команд, відгуки, результати розпізнавання мови та інше.

Документно-орієнтоване зберігання є основним принципом роботи MongoDB. Це означає, що дані зберігаються у вигляді документів, які в свою чергу представлені у форматі BSON (Binary JSON). BSON є бінарною формою JSON, що забезпечує ефективність зберігання та швидкість обробки даних.

У MongoDB документ є основною одиницею зберігання. Він складається з пар "ключ-значення", де ключ — це назва поля, а значення може бути будь-якого типу (рядок, число, масив, об'єкт тощо). Документи можуть мати різну структуру, що дозволяє гнучко налаштовувати зберігання даних.

Документи групуються в колекції. Колекція є аналогом таблиці в реляційних базах даних, але вона не вимагає однорідності структури документів. Наприклад, у колекції "users" можуть зберігатися документи, що описують користувачів, з різними полями та структурами.

Колекції об'єднуються в бази даних. MongoDB дозволяє створювати кілька баз даних на одному сервері.

Однією з головних переваг документно-орієнтованого зберігання є гнучкість схеми. Це означає, що структура документів у колекції не є жорсткою і може змінюватися без необхідності в міграції даних.

Динамічні зміни: Ви можете легко додавати або видаляти поля в документах без шкоди для інших документів у тій же колекції. Приклад: У той час як один документ може містити поле "age", інший документ у тій же колекції може не мати цього поля.

Переваги документно-орієнтованого зберігання:

- Простота: Зберігання даних у форматі JSON робить його легким для розуміння і роботи з ним, особливо для веб-розробників, які звикли працювати з JSON.
- Гнучкість: Легкість у модифікації структури даних дозволяє швидше реагувати на зміни в бізнес-вимогах, оскільки немає необхідності в складних міграціях бази даних.
- Продуктивність: Завдяки бінарному формату BSON та оптимізації запитів MongoDB забезпечує високу швидкість читання та запису даних. (рис.2.5)
- Вкладені документи: MongoDB підтримує вкладені документи, що дозволяє зберігати складні структури даних в одному документі. Це може зменшити кількість запитів до бази даних.
- Масштабованість: Документно-орієнтоване зберігання дозволяє горизонтально масштабувати базу даних, використовуючи шардінг, що забезпечує високу доступність і ефективність обробки даних.

Рис. 2.5 – модуль “mongoose”

Mongoose — це бібліотека для Node.js, яка забезпечує простий та зручний спосіб роботи з MongoDB. Вона дозволяє розробникам визначати моделі та схеми для документів, а також надає інтерфейси для роботи з базою даних. У контексті NestJS, Mongoose інтегрується для спрощення роботи з MongoDB.

Mongoose діє як обгортка навколо MongoDB, дозволяючи розробникам працювати з даними у формі об'єктів JavaScript. Замість роботи безпосередньо з колекціями та документами MongoDB, ви визначаєте моделі, які представляють ваші дані, і Mongoose автоматично обробляє зберігання та запити до бази даних.

Схеми в Mongoose визначають структуру документів, які будуть зберігатися в колекціях. Схема включає в себе:

- Поля документа та їх типи даних (рядки, числа, дати, масиви тощо).
- Валідацію полів (обов'язкові поля, унікальність, формати).
- Методів для обробки документів (методи моделі, статичні методи).

Моделі в Mongoose — це конструктори, які створюються на основі схем. Вони дозволяють вам створювати, зберігати, оновлювати та видаляти документи в MongoDB. Моделі забезпечують інтерфейс для виконання CRUD (створення, читання, оновлення, видалення) операцій з документами.

Mongoose надає вбудовані механізми для валідації даних перед їх збереженням у базі даних. Ви можете визначати правила валідації в схемах, що дозволяє уникнути збереження некоректних даних.

Mongoose підтримує middleware, що дозволяє виконувати функції перед або після певних операцій (наприклад, перед збереженням документа). Це корисно для валідації, обробки даних або виконання додаткових дій (наприклад, логування).

Mongoose є потужним інструментом для роботи з MongoDB у Node.js, що спрощує розробку додатків, зменшуючи кількість коду, необхідного для управління даними. Використання схем, моделей, валідації та можливостей запитів робить Mongoose ідеальним вибором для розробників, які хочуть ефективно управляти документно-орієнтованими базами даних.

Для розробки API буде використано Rest архітектуру. REST (Representational State Transfer) — це архітектурний стиль для розробки веб-сервісів, який базується на використанні простих та уніфікованих інтерфейсів для роботи з ресурсами через інтернет. REST дозволяє системам взаємодіяти одна з одною, використовуючи стандартизовані методи HTTP (Hypertext Transfer Protocol).

REST вимагає, щоб кожен запит від клієнта до сервера був незалежним і повністю містив всю необхідну інформацію для його обробки. Це означає, що сервер не зберігає стан клієнта між запитами. Кожен запит містить всю необхідну інформацію (дані автентифікації, параметри, стан). Сервер не зберігає інформацію про попередні взаємодії з клієнтом.

REST встановлює чіткий набір правил для взаємодії між клієнтом і сервером. Уніфікований інтерфейс означає, що всі ресурси мають бути доступні через спільний інтерфейс, використовуючи стандартизовані HTTP методи та статус-коди. Ресурси представлені у форматах, таких як JSON або XML, щоб клієнт міг їх зрозуміти. Гіпермедіа як двигун додатка (HATEOAS): кожен ресурс містить інформацію про наступні доступні дії (посилання на інші ресурси).

Клієнт повинен бути відповідальним за управління станом взаємодії (наприклад, стан сесії або транзакції). Усі дані стану, необхідні для кожного

запиту, мають бути надані клієнтом, що полегшує горизонтальне масштабування серверів, оскільки вони не повинні зберігати сесію.

РОЗДІЛ 3
ПРАКТИЧНА ЧАСТИНА РОЗРОБКИ

3.1 Розробка бекенд частини

Для побудови бекенд частини додатку створимо невелику його частину із використанням MVC архітектури.

В нашому випадку створимо модуль у NestJs, у якому опишемо та реалізуємо роботу таких сутностей:

- Продукт (Product): товар, який можна додати до замовлення.
- Замовлення (Order): запис, який включає список продуктів та їх кількість.

MVC (Model-View-Controller) — це одна з найпоширеніших архітектур для розробки програмного забезпечення. Вона спрямована на розділення відповідальностей між трьома основними компонентами: Model, View та Controller. Це забезпечує масштабованість, спрощує підтримку коду та робить його модульним.

- *Модель – Model*

Модель відповідає за управління даними, бізнес-логікою та правилами їх обробки. Вона є "серцем" додатку, яке працює з даними незалежно від того, як вони відображаються або взаємодіють із користувачем. Цей шар має такі особливості:

- Зберігає дані та працює з ними.
- Реалізує бізнес-правила та логіку додатку.
- Виконує запити до бази даних (у випадку з MongoDB — через ORM або ODM, наприклад, Mongoose).
- Не знає про існування View або Controller.

Виконує такі завдання:

- Визначення структури даних (наприклад, у Mongoose: схеми документів).
- Реалізація бізнес-логіки (наприклад, валідація або обробка даних перед їх збереженням).

Розглянемо як будуть виглядати моделі для описаних вище сутностей (рис.3.1):

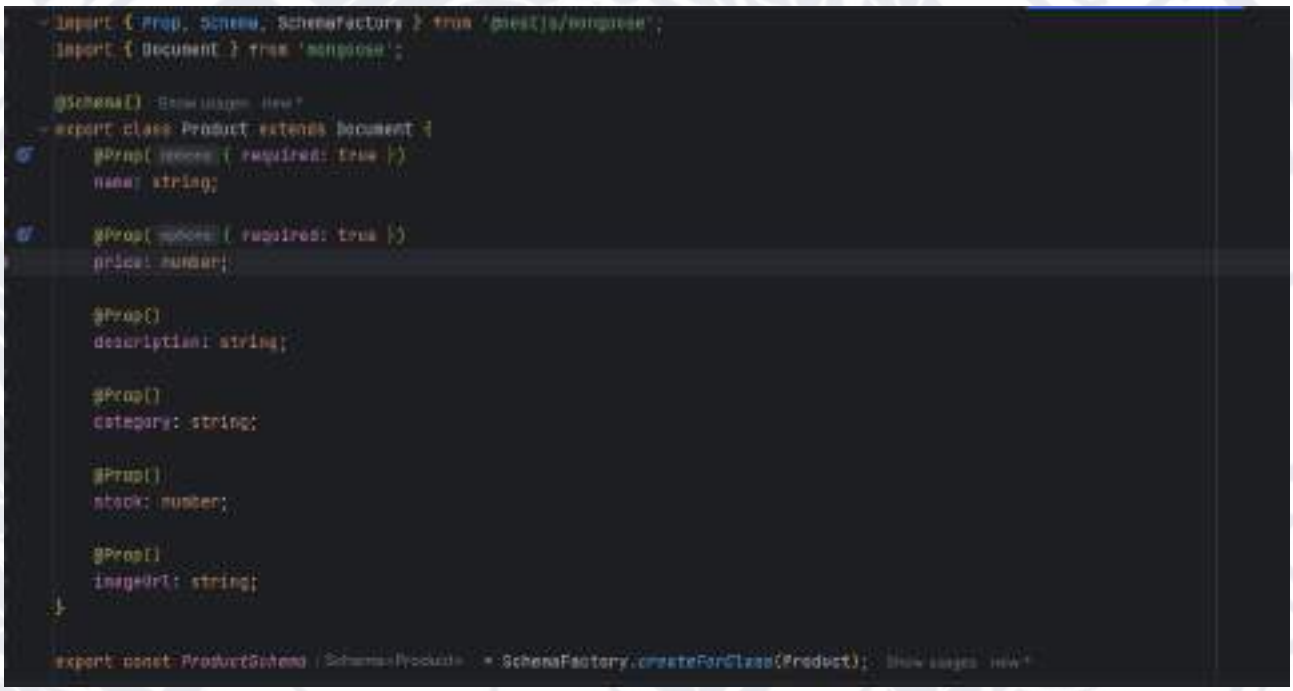


Рис. 3.1 – схема сутності “Product”

Таким чином бачимо, що для цієї сутності існують поля із відповідною валідацією:

- name – необхідне поле. Вказує на назву товару.
- price – необхідне поле. Вказує на ціну товару.
- description – опціональне поле. Вказує на опис товару
- category – опціональне поле. Вказує на категорію товару
- stock – опціональне поле. Вказує на кількість товару у залишку
- imageUrl – опціональне поле. Вказує на посилання до зображення товару.

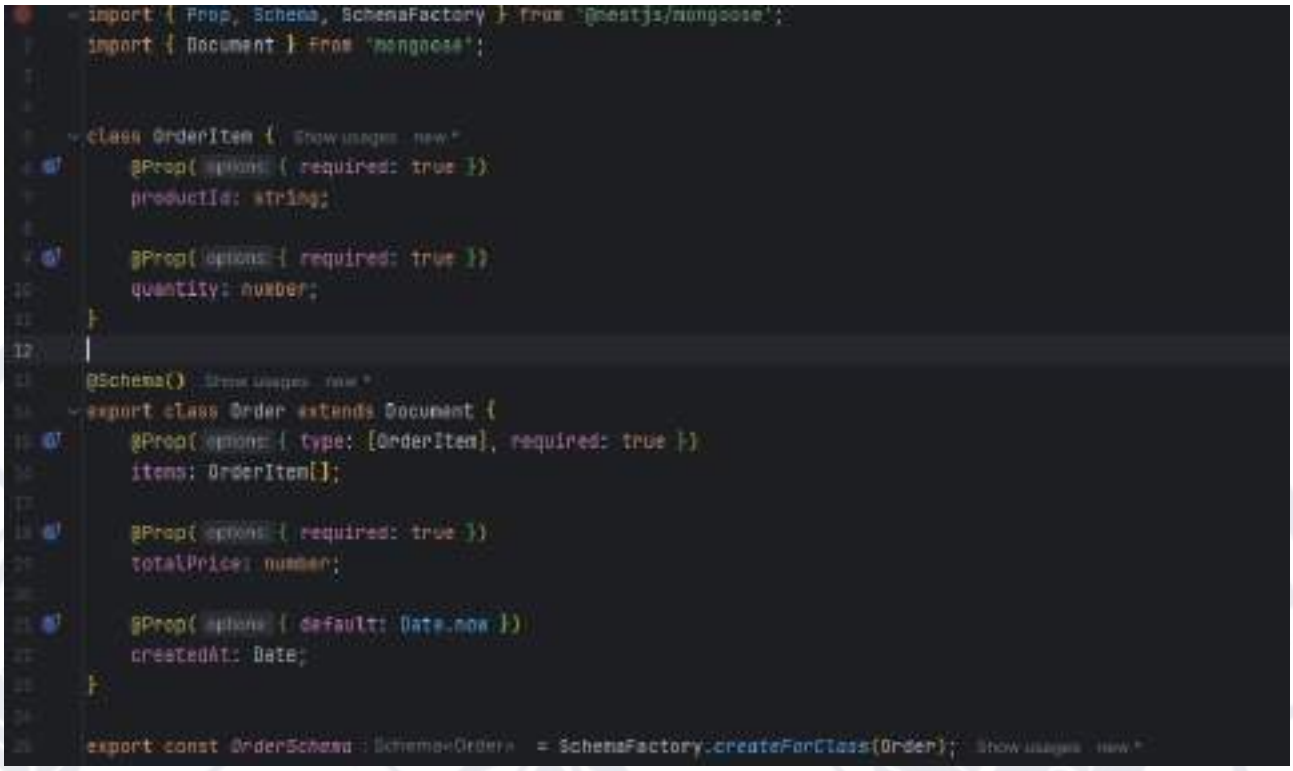


Рис. 3.2 – схема сутності “Order”

Таким чином бачимо, що для цієї сутності існують поля із відповідною валідацією:

- items – необхідне поле. Вказує на колекцію товарів, наявних у замовленні. Кожен товар визначається його айді та кількістю замовлених одиниць.
- totalPrice – необхідне поле. Вказує на загальну вартість замовлення.
- createdAt – опціональне поле. Вказує на момент створення замовлення.

Варто зауважити, що при використанні mongodb до кожного елементу колекцій автоматично додаються певні поля. Серед них є і унікальний ідентифікатор, тому явно задавати його у схемі не потрібно.

- *View - Представлення*

View відповідає за відображення інформації користувачу та обробку його дій. У випадку бекенду (REST API) View формується у вигляді HTTP-відповідей.

В нашому випадку вбудовані механізми NestJs уже реалізували усі необхідні завдання для цієї частини архітектури. Тому перейдемо до наступної частини.

- *Controller – Контролер*

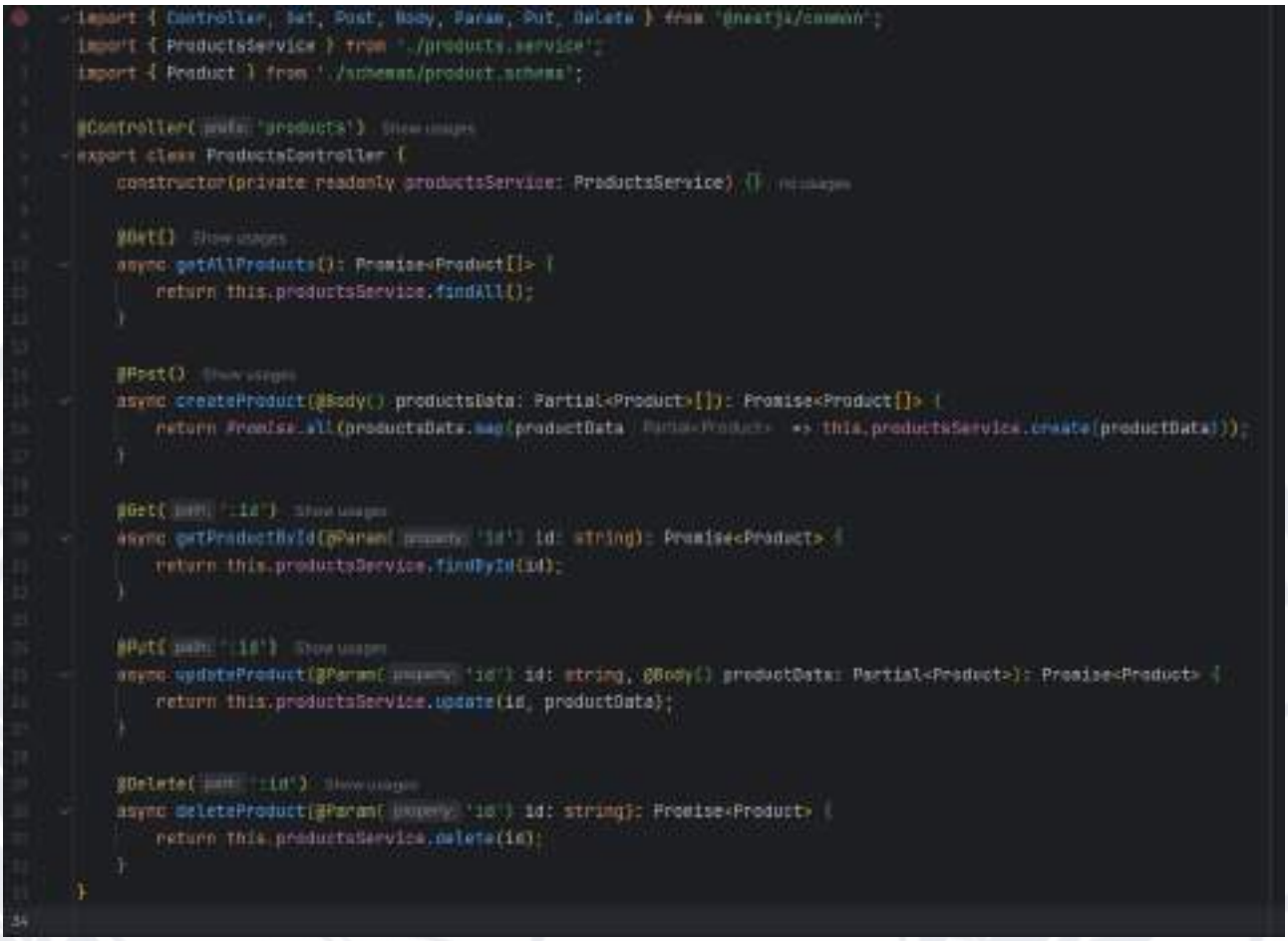
Контролер виступає як "посередник" між View і Model. Він отримує запити від користувача, обробляє їх і повертає відповіді. Ця частина має такі особливості:

- Визначає, які дії потрібно виконати залежно від запиту.
- Викликає методи моделі для доступу до даних.
- Передає дані у View для відображення.

Виконує такі завдання:

- Маршрутизація запитів (наприклад, у NestJS за допомогою декораторів @Get(), @Post() тощо).
- Взаємодія з сервісним шаром для виконання бізнес-логіки.
- Обробка помилок і виклик відповідних відповідей.

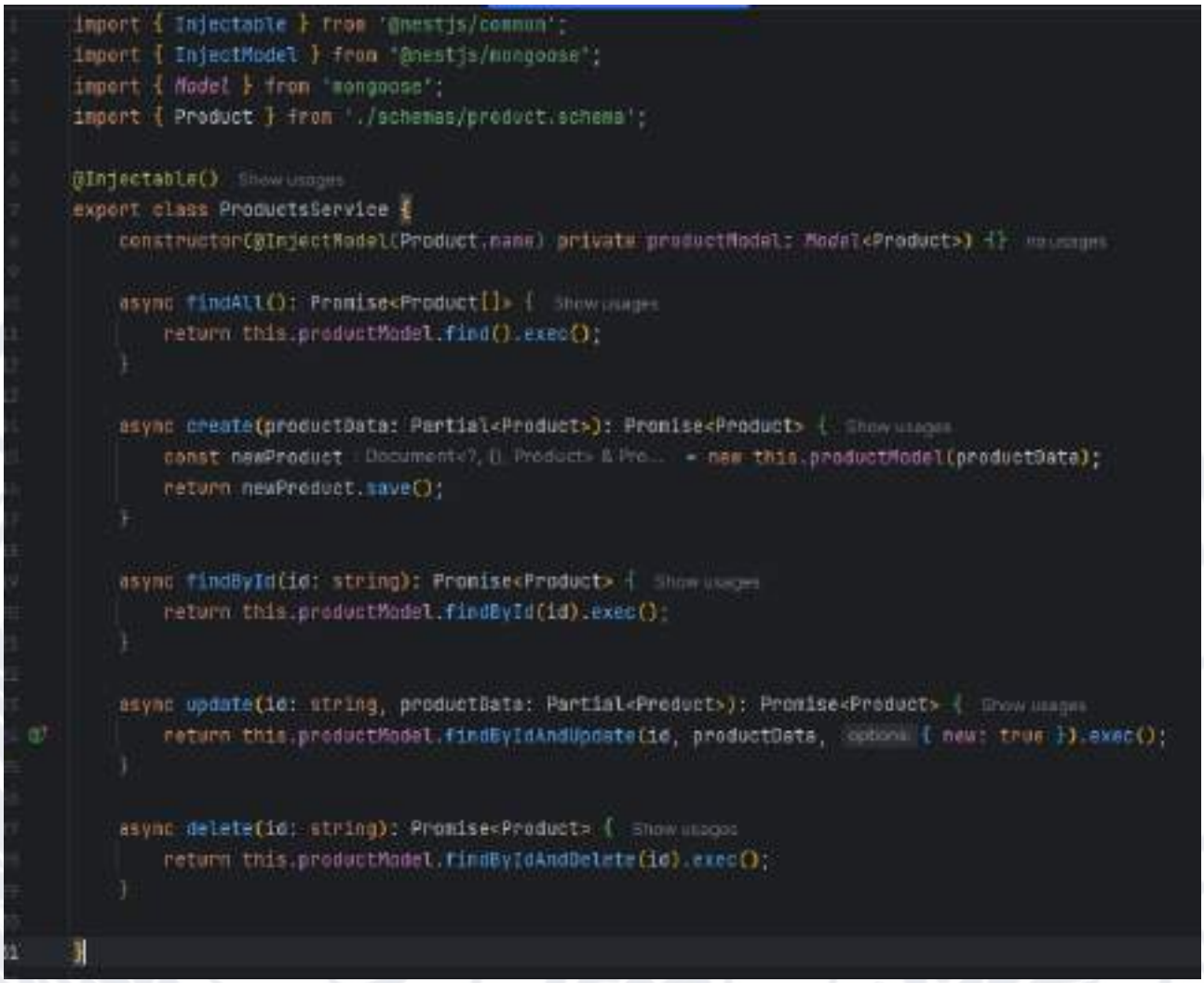
Розглянемо реалізацію даного шару у нашому додатку (рис.3.3):



```
import { Controller, Get, Post, Body, Param, Put, Delete } from '@nestjs/common';
import { ProductsService } from './products.service';
import { Product } from './schemas/product.schema';

@Controller('products')
export class ProductsController {
    constructor(private readonly productsService: ProductsService) {}

    @Get()
    async getAllProducts(): Promise<Product[]> {
        return this.productsService.findAll();
    }

    @Post()
    async createProduct(@Body() productsData: Partial<Product>[]): Promise<Product[]> {
        return Promise.all(productsData.map(productData => this.productsService.create(productData)));
    }

    @Get(':id')
    async getProductById(@Param('id') id: string): Promise<Product> {
        return this.productsService.findById(id);
    }

    @Put(':id')
    async updateProduct(@Param('id') id: string, @Body() productData: Partial<Product>): Promise<Product> {
        return this.productsService.update(id, productData);
    }

    @Delete(':id')
    async deleteProduct(@Param('id') id: string): Promise<Product> {
        return this.productsService.delete(id);
    }
}
```

Рис. 3.3 – контролер сутності “Product”

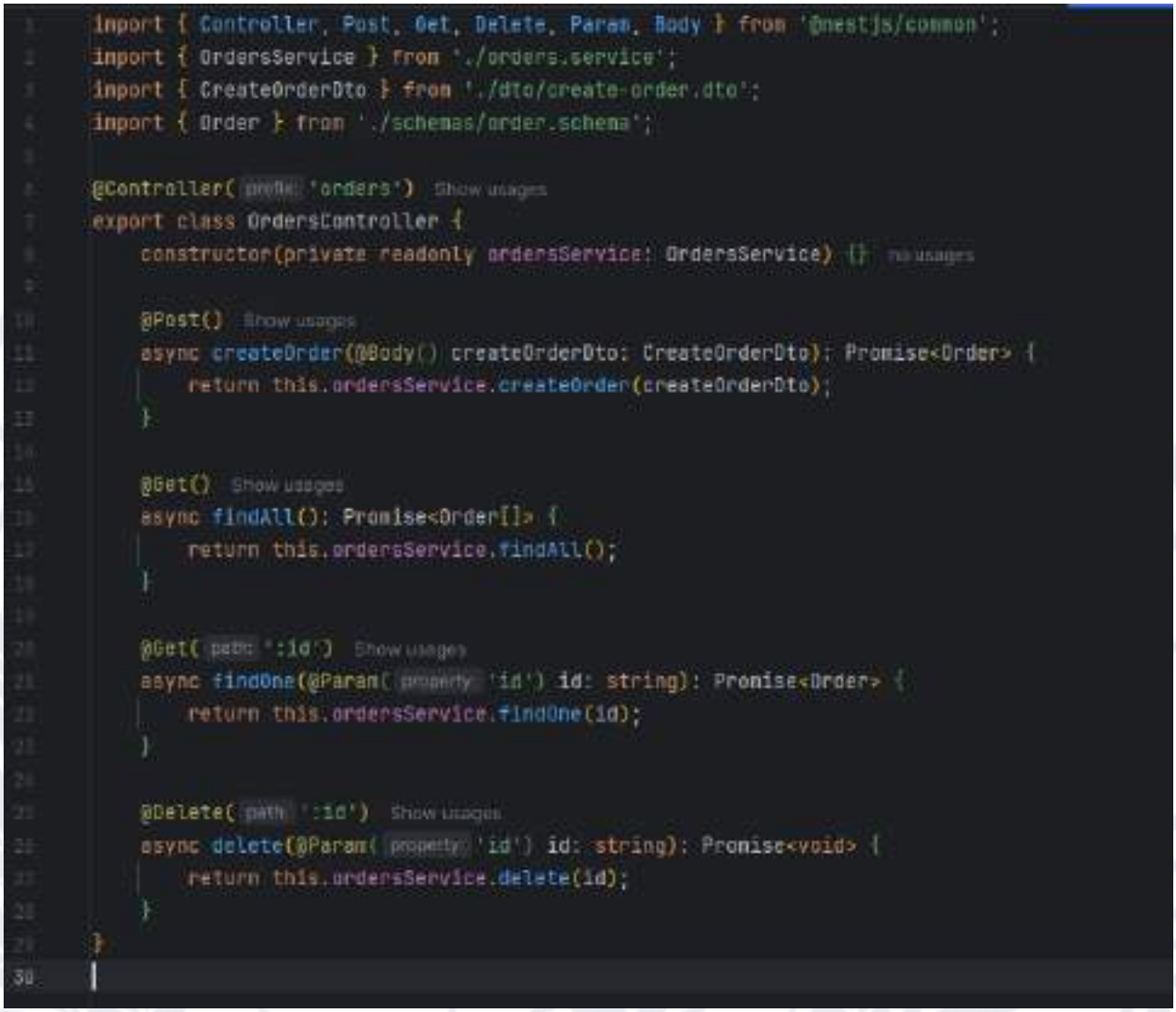
Бачимо, що даний контролер реалізує усі методи CRUD, та маршрутизує їх, пов’язуючи із сервісним шаром. Саме він відповідає за взаємодію із моделлю.(рис.3.4)



```
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Model } from 'mongoose';
import { Product } from './schemas/product.schema';

@Injectable()
export class ProductsService {
    constructor(@InjectModel(Product.name) private productModel: Model<Product>) {}

    async findAll(): Promise<Product[]> {
        return this.productModel.find().exec();
    }

    async create(productData: Partial<Product>): Promise<Product> {
        const newProduct = new this.productModel(productData);
        return newProduct.save();
    }

    async findById(id: string): Promise<Product> {
        return this.productModel.findById(id).exec();
    }

    async update(id: string, productData: Partial<Product>): Promise<Product> {
        return this.productModel.findByIdAndUpdate(id, productData, { new: true }).exec();
    }

    async delete(id: string): Promise<Product> {
        return this.productModel.findByIdAndDelete(id).exec();
    }

}
```

Рис. 3.4 – сервісний шар сутності “Product”

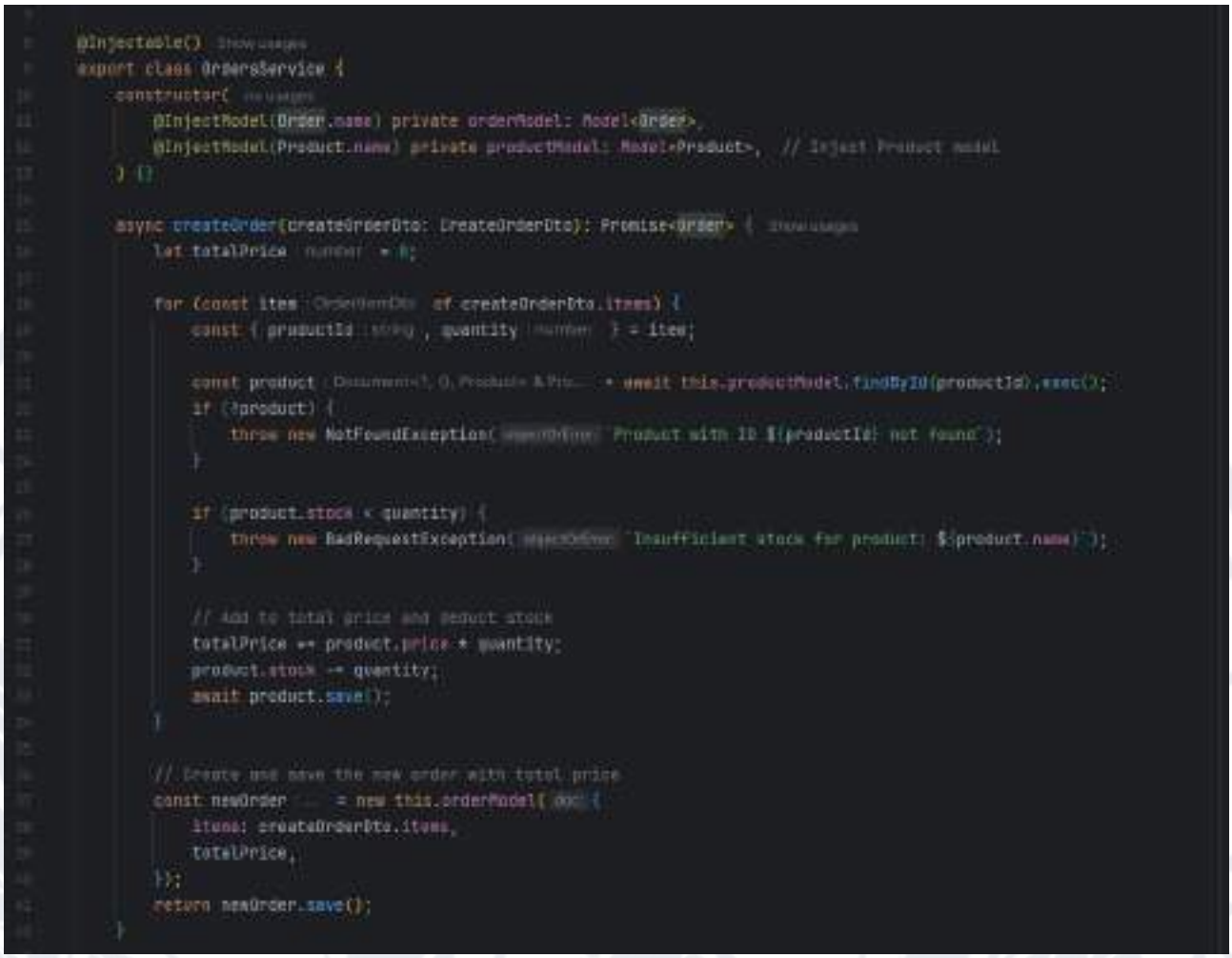
В цій частині реалізації ми реалізуємо взаємодію із моделлю за допомогою бібліотеки mongoose для коректного виконання усіх необхідних дій.(рис.3.5)



```
import { Controller, Post, Get, Delete, Param, Body } from '@nestjs/common';
import { OrdersService } from './orders.service';
import { CreateOrderDto } from './dto/create-order.dto';
import { Order } from './schemas/order.schema';

@Controller('orders') Show usages
export class OrdersController {
    constructor(private readonly ordersService: OrdersService) {} no usages

    @Post() Show usages
    async createOrder(@Body() createOrderDto: CreateOrderDto): Promise<Order> {
        return this.ordersService.createOrder(createOrderDto);
    }

    @Get() Show usages
    async findAll(): Promise<Order[]> {
        return this.ordersService.findAll();
    }

    @Get(':id') Show usages
    async findOne(@Param('id') id: string): Promise<Order> {
        return this.ordersService.findOne(id);
    }

    @Delete(':id') Show usages
    async delete(@Param('id') id: string): Promise<void> {
        return this.ordersService.delete(id);
    }
}
```

Рис. 3.5 – контролер сутності “Order”

На відміну від іншої сутності, в даному контролері не реалізовано метод Put, оскільки за наявною бізнес-логікою редагування замовлення неможливе.



```
@Injectable()
export class OrdersService {
    constructor(
        @InjectModel(Order.name) private orderModel: Model<Order>,
        @InjectModel(Product.name) private productModel: Model<Product>, // Inject Product model
    ) {}

    async createOrder(createOrderDto: CreateOrderDto): Promise<Order> {
        let totalPrice = 0;

        for (const item of createOrderDto.items) {
            const { productId, quantity } = item;

            const product = await this.productModel.findById(productId).exec();
            if (!product) {
                throw new NotFoundException(`Product with ID ${productId} not found`);
            }

            if (product.stock < quantity) {
                throw new BadRequestException(`Insufficient stock for product: ${product.name}`);
            }

            // Add to total price and deduct stock
            totalPrice += product.price * quantity;
            product.stock -= quantity;
            await product.save();
        }

        // Create and save the new order with total price
        const newOrder = new this.orderModel({
            items: createOrderDto.items,
            totalPrice,
        });
        return newOrder.save();
    }
```

Рис. 3.6 – метод Create сутності “Order ”

На мою думку, серед усього сервісного шару варто розглянути лише 1 метод.(рис.3.6) Це створення замовлення, адже тут необхідно провалідувати отримані дані та оновити кожен екземпляр сутності “Product” при успішному створенні замовлення. Таким чином, ми перевіряємо замовлену кількість кожного товару, та за умови, що в наявності знаходиться менше товарів – повертаємо помилку через API. В успішному ж сценарії – зменшуємо кількість відповідного товару у наявності.

3.2 Тестування бекенд частини

Щоб упевнетись у коректній роботі бекенд частини необхідно перевірити його. Для цього можна скористатися двома інструментами.

Перший з них: Swagger - потужний інструмент для документування API, який забезпечує зручність використання та тестування бекенд-додатків. У контексті демонстраційного додатку для управління продуктами та замовленнями Swagger дозволяє створити інтерактивну документацію, що відображає всі доступні ендпоінти, їхні параметри, структуру запитів та відповідей.

У нашому додатку Swagger виконує роль центрального інтерфейсу для роботи з REST API. Після запуску серверної частини доступ до Swagger UI надається через веб-інтерфейс, зазвичай на маршруті /api. Через цей інтерфейс розробники або тестувальники можуть легко переглянути всі доступні методи для роботи з продуктами (створення, отримання, оновлення, видалення) та замовленнями (створення, перегляд, видалення).

Swagger також дозволяє тестувати API безпосередньо з інтерфейсу. Наприклад, щоб створити новий продукт, користувач може ввести потрібні дані у форму, яку Swagger генерує автоматично на основі DTO, і відправити запит безпосередньо через браузер. Результати виконання запиту, включно з кодом статусу та відповіддю, відображаються в реальному часі, що значно полегшує процес відлагодження.(рис.3.7)

Рис. 3.7 – інтерфейс Swagger

Іншим інстурментом для відлагодження та тестування є Postman. Postman — це популярний інструмент для тестування API, який забезпечує зручний і функціональний інтерфейс для взаємодії з бекенд-додатками. У контексті демонстраційного додатку для управління продуктами та замовленнями Postman дозволяє розробникам та тестувальникам легко надсилати HTTP-запити до REST API та аналізувати відповіді, які повертає сервер.

Однією з головних переваг Postman є можливість створення колекцій запитів, що представляють собою структуровані групи для роботи з API. У нашому додатку можна створити окремі колекції для ендпоінтів, що керують продуктами та замовленнями. Кожен запит, будь то POST, GET, PUT чи DELETE, додається до відповідної колекції з повною інформацією про URL, заголовки, тіло запиту та очікувані параметри.

Postman спрощує тестування функціональності API. Наприклад, для перевірки створення нового продукту користувач може відкрити відповідний запит у Postman, ввести дані у форматі JSON (назва, ціна, категорія) та надіслати запит. Відповідь сервера, включаючи статус код (наприклад, 201 Created) і тіло відповіді, буде миттєво відображена у зручному форматі. Це дозволяє швидко перевіряти коректність роботи бекенду та знаходити помилки.

Крім того, Postman дозволяє генерувати звіти та ділитися колекціями з іншими членами команди. Наприклад, після завершення розробки можна експортувати колекцію запитів, які ілюструють, як працює API додатку, і надати її фронтенд-розробникам або тестувальникам для інтеграції чи подальшої перевірки. Відповідну колекцію запитів прикріплено у додатку А.

Postman також підтримує змінні середовища, що дуже зручно для роботи з додатками, які мають різні конфігурації (наприклад, тестовий, розробницький або продуктивний сервер). Для нашого додатку можна створити кілька середовищ, задавши змінні для базового URL (http://localhost:3000) та інших параметрів, що спрощує зміну конфігурацій.

У підсумку, Postman є потужним інструментом, який спрощує тестування, документування та налагодження бекенд-додатків. У нашій демонстраційній системі він забезпечує зручний спосіб перевірки роботи API для продуктів і замовлень, значно скорочуючи час на тестування та відлагодження.(рис.3.8)

Рис. 3.8 – інтерфейс Postman

3.3 Розробка інфраструктури голосового інтерфейсу

Для створення фронтенд додатку із елементами голосового інтерфейсу необхідно визначитись із методом взаємодії. В розробленому додатку використовується патерн стратегія, що дозволяє реалізувати власний список команд для окремих частин додатку. До прикладу, команда “Select” в модулі продуктів та модулі замовлень виконує схожі, проте різні команди.

Усі стратегії повинні бути сервісам, які будуть активуватись та використовуватись лише у відповідній частині додатку, що дозволить нам підтримувати принцип OCP. Тобто наш додаток буде відкритим для розширення, оскільки такий механізм дозволить створювати нові стратегії та легко використовувати їх. З іншого боку, немає необхідності модифікувати раніше створені стратегії для впровадження нових, оскільки обрана архітектура не доволі гнучка.(рис.3.9)

Фінальна концепція взаємодії команд виглядає так:

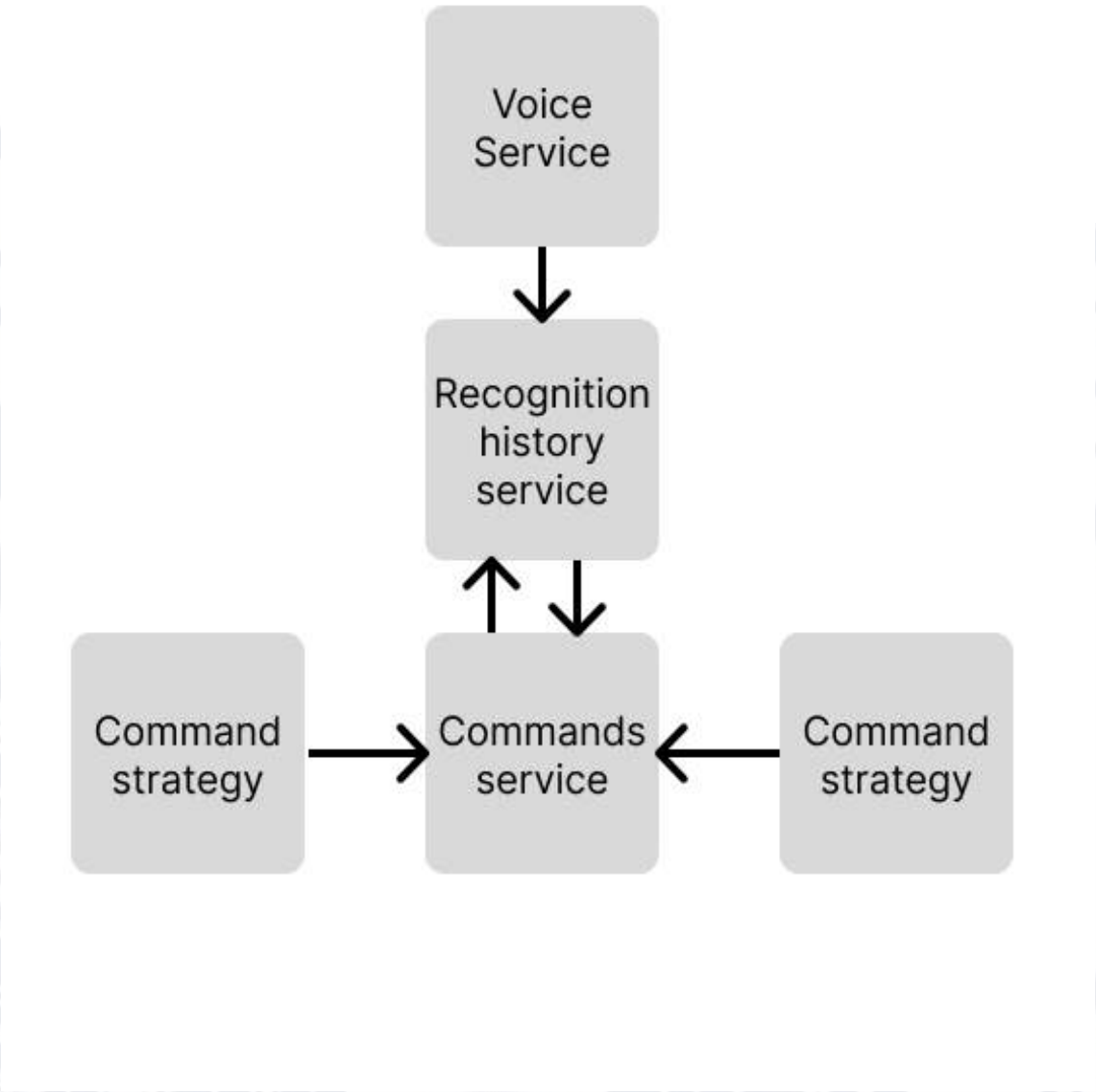


Рис. 3.9 – схема взаємодії сервісів

Розглянемо окремі складові детальніше.

- *Voice service (рис.3.10)*

Основні елементи сервісу:

- turnedOn$ – асинхронна змінна, що відповідає за стан розпізнавання мови. Використовується у додатку для індикації.
- recognition – об’єкт в якому зберігається реалізований у різних браузерах інтерфейс SpeechRecognition.

- transcript$ – асинхронна змінна із останньою розпізнаною командою.

Основні методи:

- startListening – починає вмикає розпізнавання мови. У разі відсутності SpeechRecognition – створює його із відповідними базовими налаштуваннями. При розпізнаванні команди передає її останнє значення у змінну transcript$.
- stopListening – призупиняє запис при наявності створеного об’єкту.

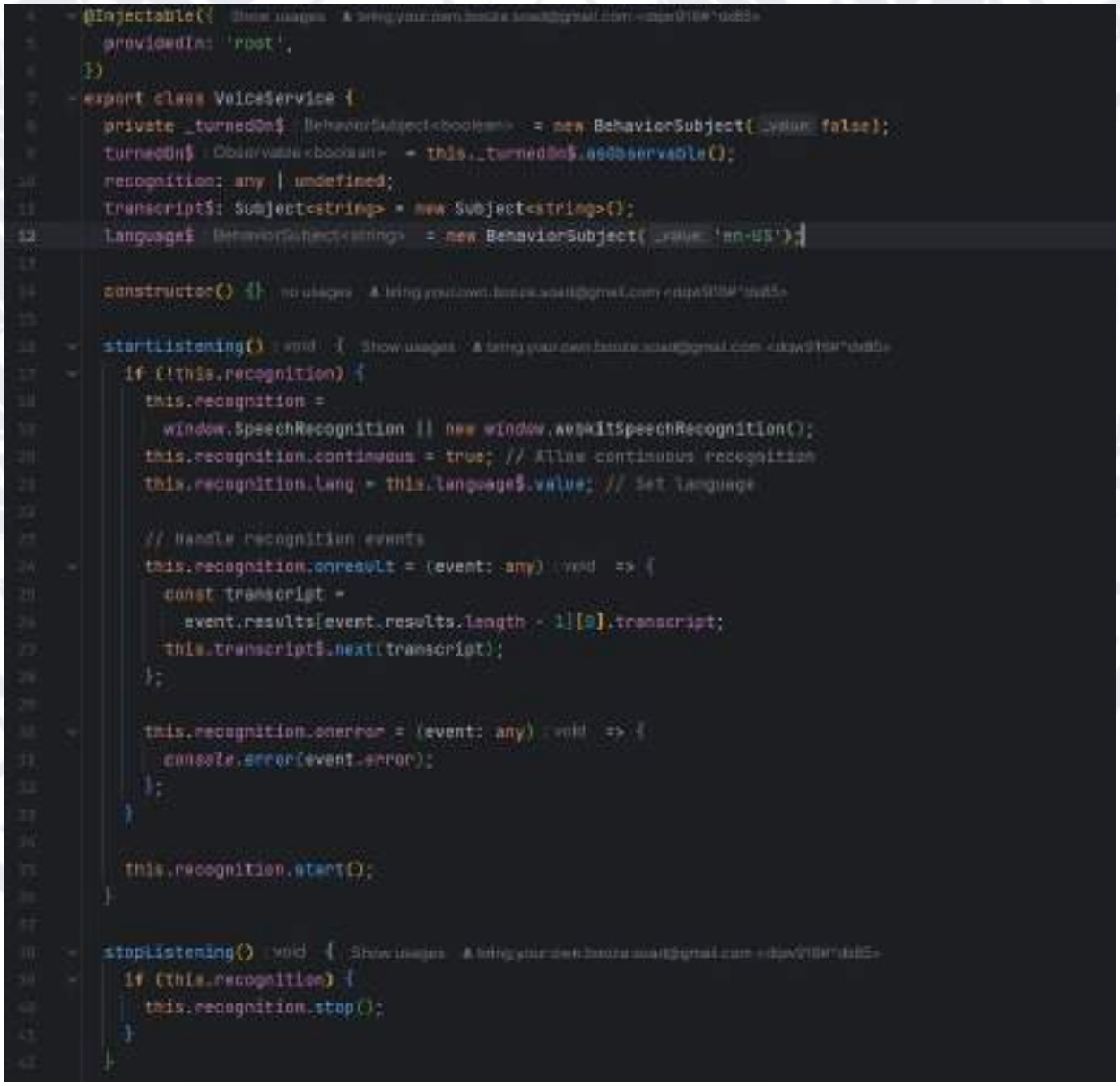


Рис. 3.10 – VoiceService

Присутні і інші методи, проте вони не настільки критичні, до прикладу: toggleVoiceRecognition – змінює значення turnedOn$ на протилежне та запускає відповідний метод.

- RecognitionHistoryService (рис.3.11)

В цьому сервісі ми інжектуємо VoiceService та підписуємось на поле transcript$, що дозволяє нам кожного разу, коли розпізнано нову команду записати її до історії. Це необхідно для компоненту, де користувач зможе побачити в хронологічному порядку послідовність команд, та для взаємодії із CommandService.

Рис. 3.11 – RecognitionHistoryService

- CommandsService (рис. 3.12)

Цей сервіс в Angular виконує управління командами голосового розпізнавання, обробляючи історію транскрипцій і виконуючи відповідні дії на основі визначених команд. Він використовує стратегії (CommandsStrategy) (рис.3.13) для налаштування списку команд залежно від мови та підтримує можливість підключення стратегії, що використовується на окремій сторінці.

Основні елементи:

- recognitionHistoryService – інжектований сервіс RecognitionHistoryService
- strategies – масив стратегій, команди яких будуть виконуватись у режимі реального часу.

Принцип роботи:

- Щоразу, коли в історію транскрипцій додається нова транскрипція - сервіс перевіряє останній запис у масиві транскрипцій.
- Якщо виявлено відповідність фрази ключовій команді, викликається дія, пов'язана з цією командою.
- За допомогою методів addStrategy та removeStrategy відбувається зміна списку команд.

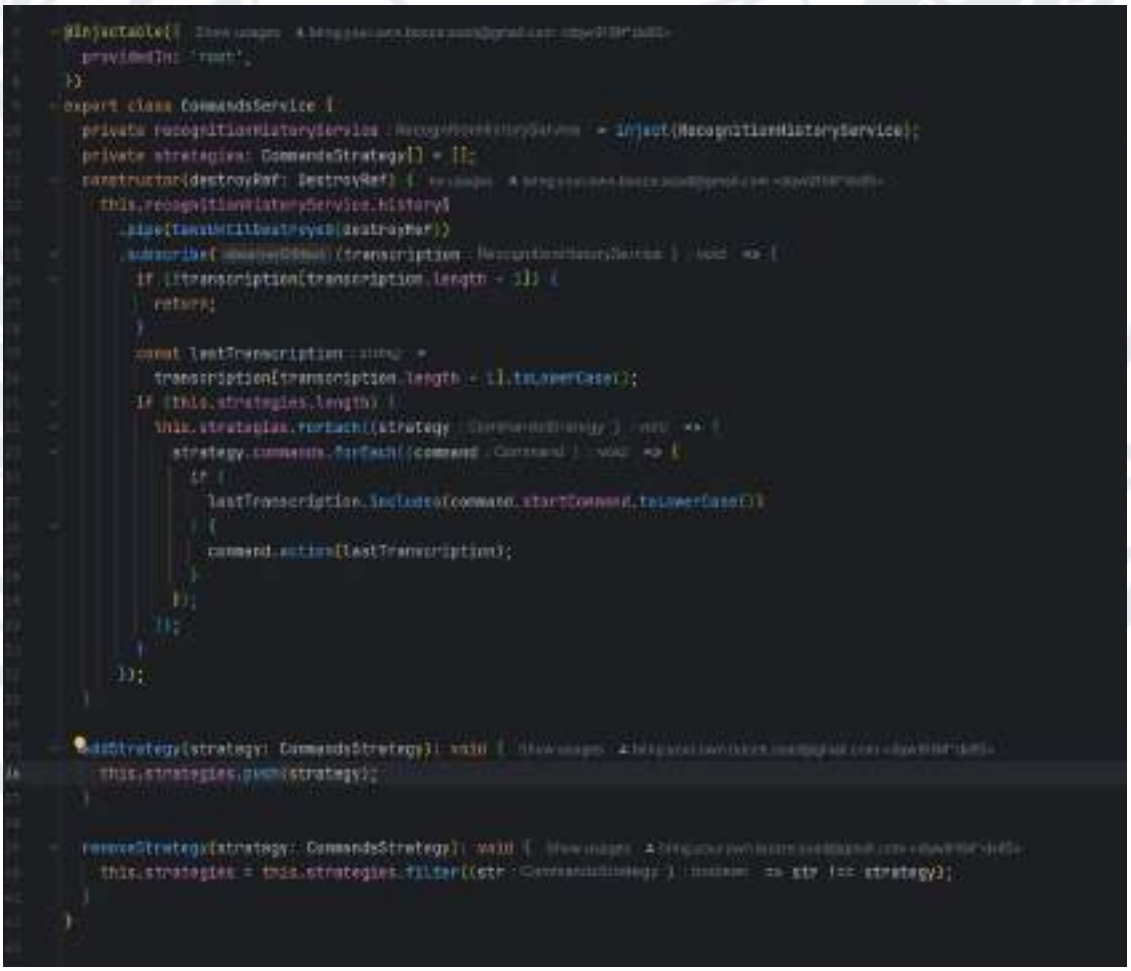


Рис. 3.12 – CommandsService

Рис. 3.13 – Інтерфейс Command та клас CommandStrategy

Команда — це окремий функціональний блок, який визначає, яку дію потрібно зробити у відповідь на конкретний голосовий запит. Вона складається з двох основних частин: startCommand та action. Поле startCommand задає ключову фразу або тригер, який розпізнається у голосовому вводі, наприклад, "Navigate to" або "Select". Поле action є колбеком, який виконується, коли тригерна фраза виявлена у введеному тексті.

Принцип роботи команди полягає у її реєстрації в одній із стратегій (глобальній або локальній), після чого стратегія перевіряє, чи містить розпізнаний текст її тригерну фразу. Якщо так, виконується відповідна дія, яка може включати навігацію між сторінками, маніпуляції з елементами інтерфейсу або зміни стану додатку. Команди є гнучкими, тому їх можна налаштовувати для різних сценаріїв, дозволяючи створювати інтуїтивний та адаптивний інтерфейс, що реагує на голосові команди користувача.

У додатку реалізовано 2 типи стратегій – глобальні та локальні.

Глобальні стратегії надають команди, доступні з будь-якої точки додатку, незалежно від активного контексту. Наприклад, DefaultCommandsStrategyEn і DefaultCommandsStrategyUa дозволяють виконувати загальні дії, такі як навігація, перемикання бокового меню або припинення розпізнавання голосу.

Їх роль полягає у забезпеченні базової функціональності додатку, яка не залежить від того, де перебуває користувач. Приклади дій:

- Відкрити або закрити бокове меню.

- Повернутися на попередню сторінку.
- Навігація за URL.

Глобальні стратегії визначають набір команд і прив'язують їх до функціоналу сервісів, таких як Router, Location, або SidebarService. Наприклад, команда "Sidebar" у DefaultCommandsStrategyEn викликає метод toggle() у SidebarService для перемикання стану бокового меню. Завдяки ін'єкції сервісів, глобальні стратегії можуть взаємодіяти з будь-якою системою додатку.(рис. 3.14)

```
@Injectable({
  providedIn: 'root',
})
export class DefaultCommandsStrategyEn implements CommandsStrategy {
  private router: Router = inject(Router);
  private sidebarService: SidebarService = inject(SidebarService);
  private location: Location = inject(Location);
  private voiceService: VoiceService = inject(VoiceService);

  commands: Command[] = [
    {
      startCommand: 'Sidebar',
      action: () => this.sidebarService.toggle(),
    },
    {
      startCommand: 'Navigate to',
      action: (transcription: string): void => {
        const path: string = transcription.toLowerCase().split('navigate to ')[1];
        this.router.navigate([path]);
      },
    },
    {
      startCommand: 'Back',
      action: () => this.location.back(),
    },
    {
      startCommand: 'Stop',
      action: () => this.voiceService.toggleVoiceRecognition(),
    },
  ];
}
```

Рис. 3.14 – Глобальна стратегія англійською мовою

Локальні стратегії у контексті роботи з голосовими командами представляють собою вузько спрямовані набори команд, які активуються в конкретних частинах додатку або для обмеженого набору функціональності.

Вони відповідають за те, щоб забезпечити користувача інструментами, адаптованими до поточного контексту роботи. Наприклад, на сторінці списку продуктів локальна стратегія може включати команди для вибору продукту, перемикання між елементами списку або перегляду деталей. Натомість на сторінці з деталями продукту інша локальна стратегія може додати команду "Add", яка дозволяє додати продукт у кошик.

Основний принцип роботи локальних стратегій полягає у тому, що вони стають активними лише у певному контексті. Це досягається шляхом додавання стратегії до списку активних у момент, коли користувач переходить до відповідного розділу додатку. Локальні стратегії не перетинаються з глобальними, а радше доповнюють їх, надаючи більше контролю над конкретними аспектами роботи.

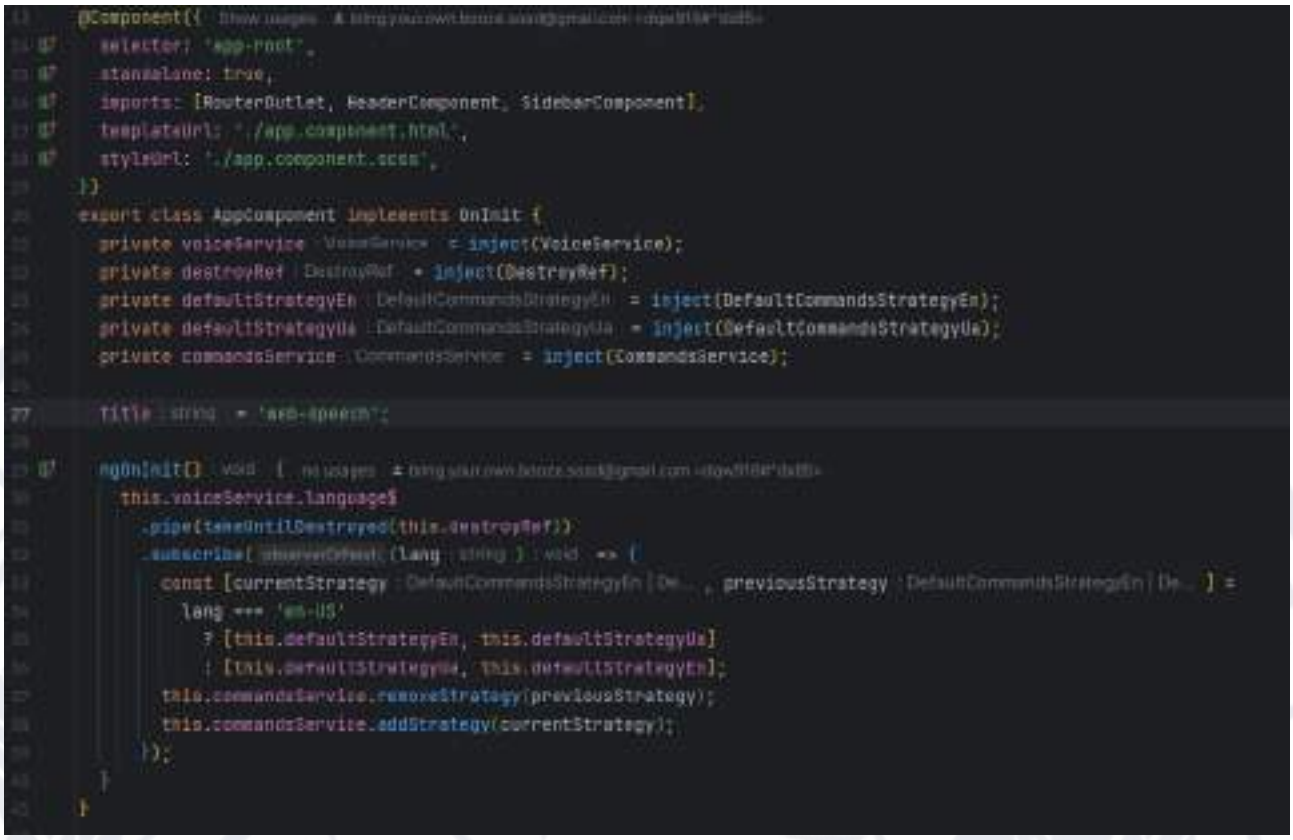
Перевага локальних стратегій у тому, що вони зменшують перевантаження додатку зайвими командами, які не стосуються поточного контексту. Користувач взаємодіє лише з тими командами, які мають сенс у даній ситуації, що робить інтерфейс голосових команд більш зрозумілим і зручним. Це також дозволяє уникнути конфліктів між командами, які можуть виникнути, якщо один тригер використовується для різних дій у різних частинах додатку.(рис.3.15)

```
@Injectable({
  providedIn: 'root',
})
export class CommandStrategyProductList implements CommandsStrategy {
  private productsService = inject(ProductsService);
  commands: Command[] = [
    {
      startCommand: 'Select',
      action: () => {
        this.productsService.selectProduct(0);
      },
    },
    {
      startCommand: 'Reset',
      action: () => {
        this.productsService.resetSelectedProduct();
      },
    },
    {
      startCommand: 'Next',
      action: () => {
        const newIndex =
          this.productsService.selectedProductIndex$.value === null
            ? 0
            : this.productsService.selectedProductIndex$.value + 1;
        this.productsService.selectProduct(newIndex);
      },
    },
    {
      startCommand: 'Previous',
      action: () => {
        const newIndex =
          this.productsService.selectedProductIndex$.value === null
            ? 0
            : this.productsService.selectedProductIndex$.value - 1;
        this.productsService.selectProduct(newIndex);
      },
    },
```

Рис. 3.15 – приклад локальної стратегії

3.4 Розробка фронтенд додатку голосового інтерфейсу

Варто зазначити, що глобальну стратегію, як і конфігурацію усіх сторінок ми задаємо у кореневій частині додатку – AppComponent.(рис.3.16) Тут ми підписуємось на Observable із заданою мовою та додаємо чи міняємо мову глобальної стратегії у відповідності з нею.



```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, HeaderComponent, SidebarComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.scss',
})
export class AppComponent implements OnInit {
  private voiceService = inject(VoiceService);
  private destroyRef = inject(DestroyRef);
  private defaultStrategyEn = inject(DefaultCommandsStrategyEn);
  private defaultStrategyUa = inject(DefaultCommandsStrategyUa);
  private commandsService = inject(CommandsService);

  title = 'web-speech';

  ngOnInit(): void {
    this.voiceService.language$
      .pipe(takeUntilDestroyed(this.destroyRef))
      .subscribe((lang) => {
        const [currentStrategy, previousStrategy] =
          lang === 'en-US'
            ? [this.defaultStrategyEn, this.defaultStrategyUa]
            : [this.defaultStrategyUa, this.defaultStrategyEn];
        this.commandsService.removeStrategy(previousStrategy);
        this.commandsService.addStrategy(currentStrategy);
      });
  }
}
```

Рис. 3.16 –AppComponent

```
export const routes: Routes = [
  { path: '', redirectTo: 'help', pathMatch: 'full' },
  {
    path: 'help',
    loadComponent: () =>
      import('./modules/help/instructions/instructions.component').then(
        (m) => m.InstructionsComponent,
      ),
  },
  {
    path: 'history',
    loadComponent: () =>
      import('./modules/history/history.component').then(
        (m) => m.HistoryComponent,
      ),
  },
  {
    path: 'shop',
    loadChildren: () =>
      import('./modules/shop/shop.routes').then((routes) => routes.routes),
  },
];
```

Рис. 3.17 –AppRoutes

Корінь додатку в даному випадку також виконує роль контейнера, із встановленим заголовком сторінки та боковим меню для навігації між модулями. Саме у заголовку і знаходиться кнопка для увімкнення голосового інтерфейсу.

Для демонстрації роботи голосового інтерфейсу було створено такі фронтенд модулі:

- Help (navigate to help) (рис.3.18)

Проста сторінка із списком доступних у додатку глобальних команд, яка є першою сторінкою, що зустрічає користувача.

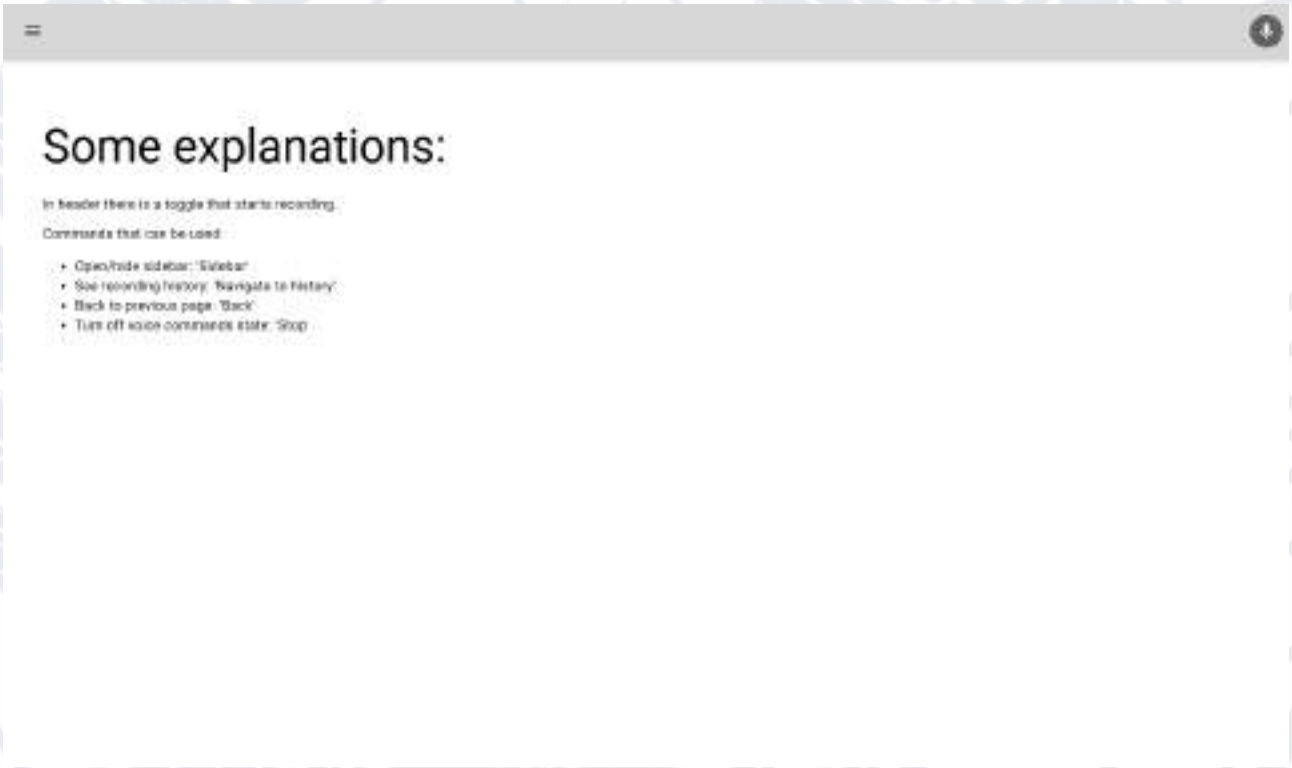


Рис. 3.18– сторінка help

- History (navigate to history)рис.3.19

Сторінка є зручним інструментом для налагодження роботи додатку, оскільки саме тут здебільшого і можна побачити результат транскрипції.

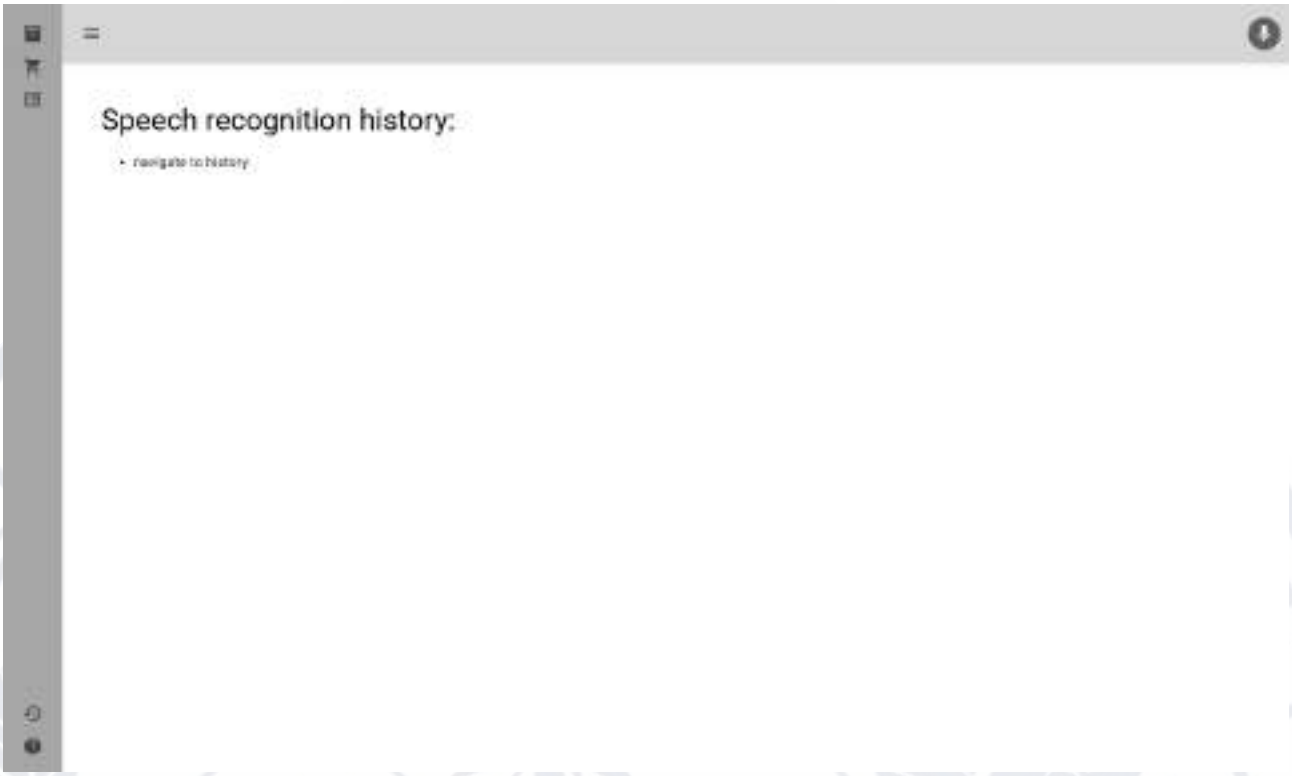


Рис. 3.19 – сторінка history

- Shop (navigate to shop)

В даному випадку Shop – це низка компонентів та відповідних сторінок. Конфігурація маршрутизації для них:

- "products" – список товарів
- "products/:id" – сторінка деталей про товар
- "cart" – кошик, в якому зберігаються усі додані товари
- "orders" – список замовлень
- "orders/:id" – деталі замовлення
- "" – порожній шлях, з нього відбувається переадресація на список продуктів

Сторінка із списком продуктів будується з даних, отриманих з серверної частини додатку у форматі карток товарів. Щоб додати товар до кошику потрібно перейти на сторінку деталей товару, скориставшись кнопкою Details. Кількість карток в одному рядку змінна та підлаштовується під розширення.(рис. 3.20)

Доступні команди:

- Select – увімкнути навігацію за картками.
- Next – обрати наступний товар.
- Previous – обрати попередній товар.
- Details – перейти на сторінку товару.

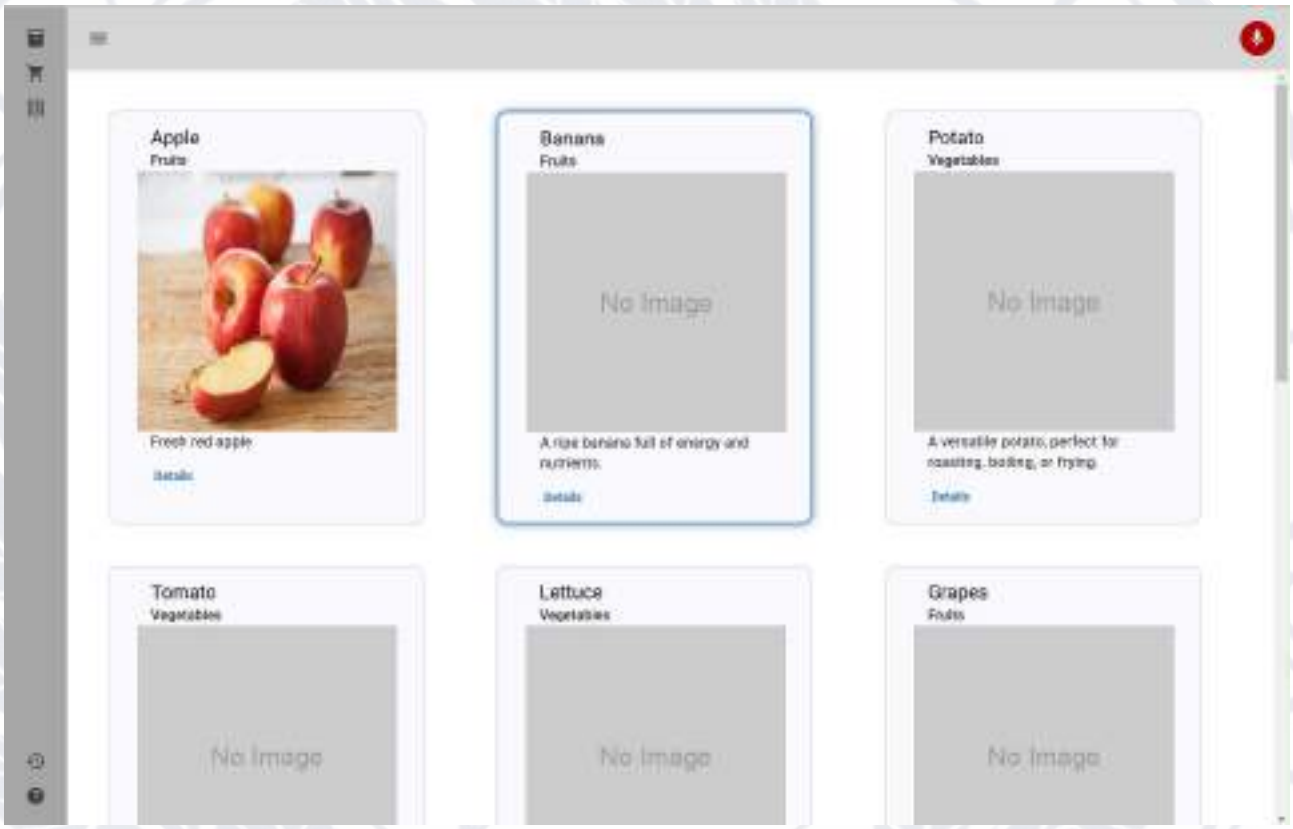


Рис. 3.20 – сторінка Products list

Сторінка деталей товару дозволяє переглянути деталі про товар (його ціну, наявність у залишку) та обрати кількість товару для додавання у корзину. Також реалізовано валідацію поля кількість, щоб унеможливити замовлення більшої кількості за можливу.

Доступні команди:

- Increase – збільшити кількість товару у полі форми
- Decrease – зменшити кількість товару у полі форми
- Add – додати обрану кількість товару до кошика

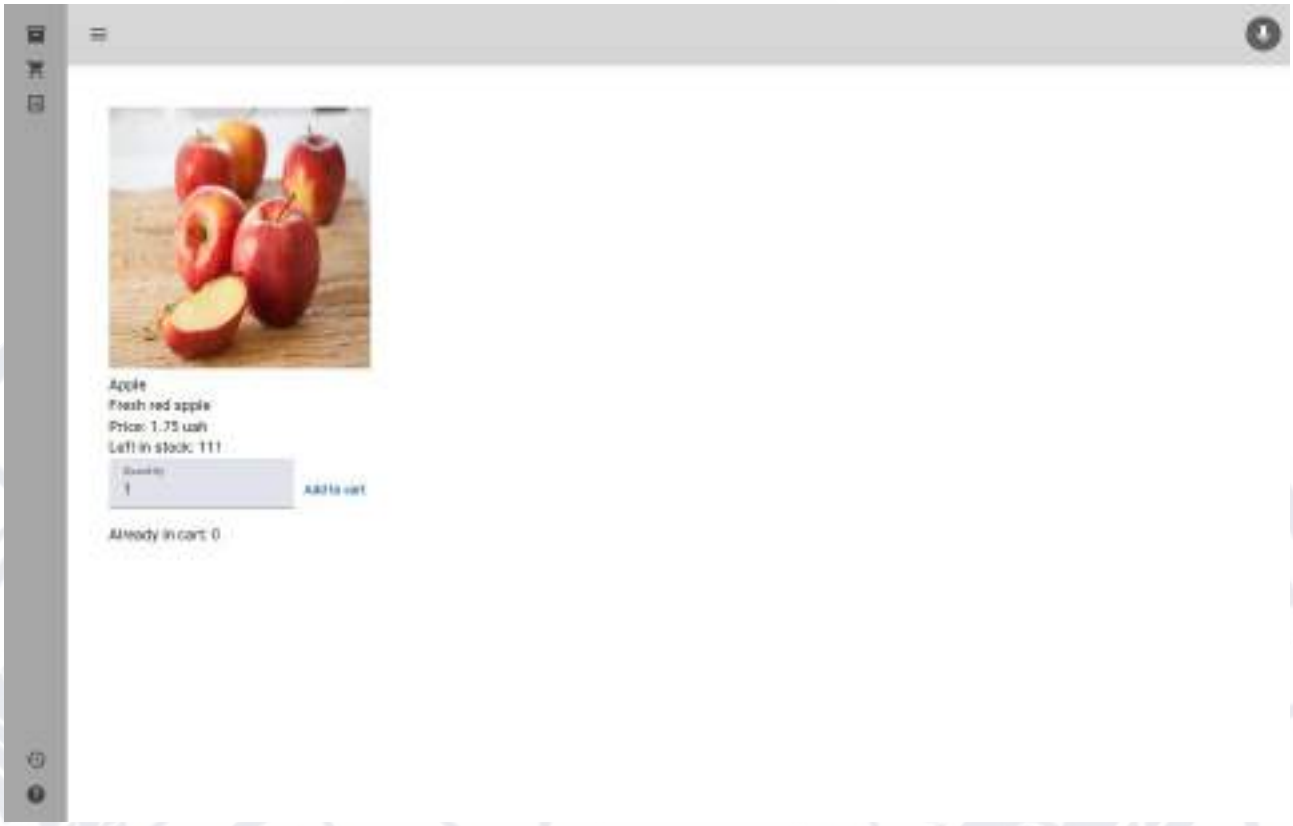


Рис. 3.21 – сторінка Product details

Сторінка кошику дозволяє переглянути список обраних товарів та побачити фінальну вартість замовлення. При натисненні на кнопку Delete – товар вилучається з кошику, а при натисненні на кнопку Checkout – створюється відповідне замовлення у базі даних. (рис.3.22)

Доступні команди:

- Select – обрати товар
- Next – обрати наступний товар
- Previous – обрати попередній товар
- Delete – видалити обраний товар
- Checkout – створити замовлення із обраного списку товарів

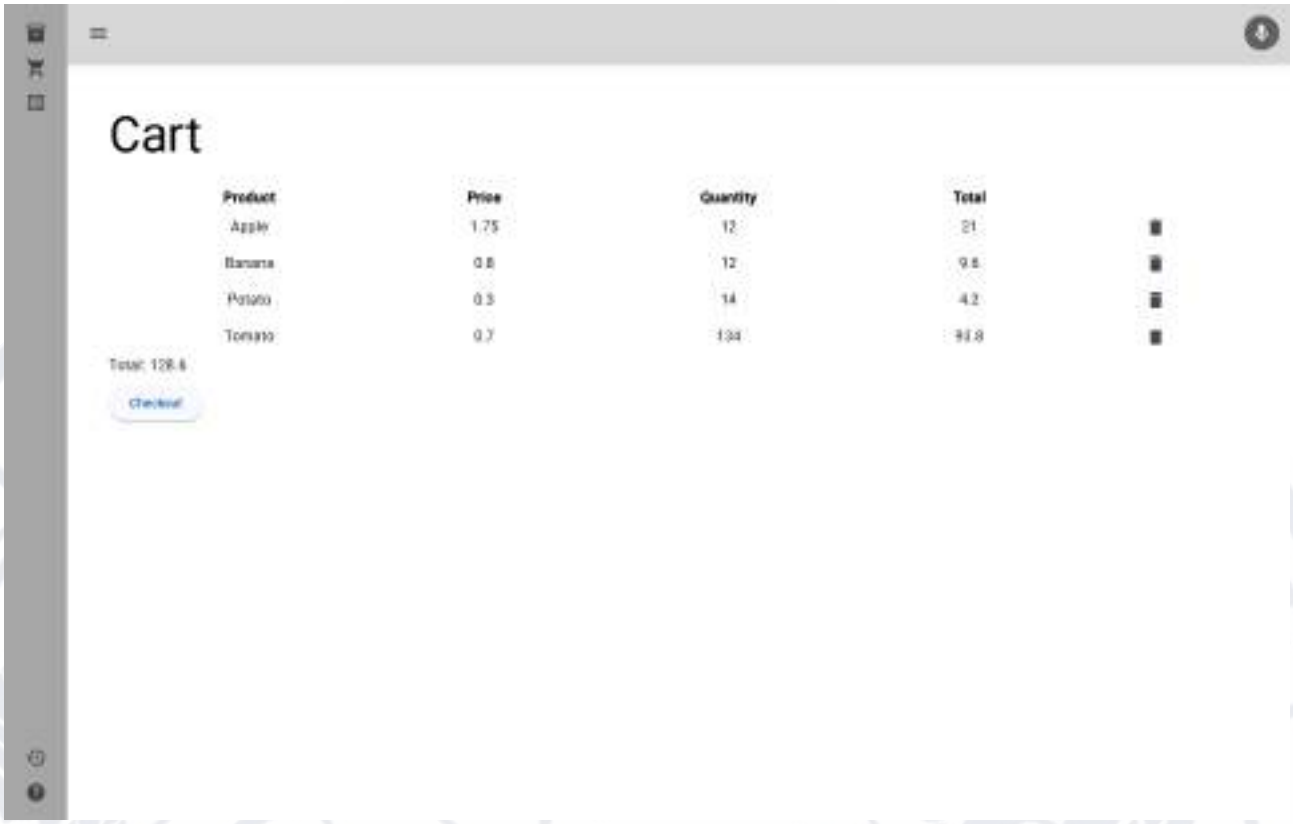


Рис. 3.22 – сторінка Cart

Сторінка списку замовлень дозволяє переглянути список існуючих замовлень.

Доступні команди:

- Select – обрати замовлення
- Next – обрати наступне замовлення
- Previous – обрати попереднє замовлення
- Delete – видалити обране замовлення
- Details – перейти на сторінку деталей про замовлення.

Рис. 3.23 – сторінка Orders list

Сторінка деталей про замовлення дозволяє візуалізувати дані про створене замовлення.

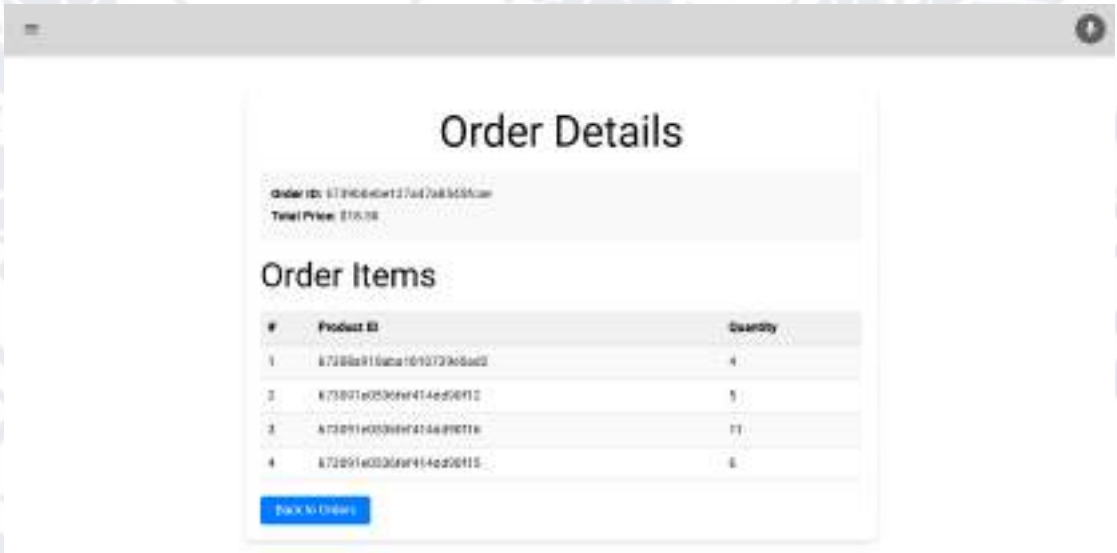


Рис. 3.24 – сторінка Order details

ВИСНОВКИ

У рамках даної роботи було розглянуто актуальність побудови вебсайтів із використанням голосового інтерфейсу та методи їх побудови.

Були проаналізовані обрані технології та інструменти для розробки веб-застосунків, зокрема Angular, NestJS і MongoDB. Особлива увага приділена їх перевагам, таким як швидкість розробки, гнучкість та масштабованість, а також можливостям інтеграції та підтримки модульності. Розглянуто переваги обраного серверу баз даних MongoDB, включаючи його ефективність у роботі з великими обсягами даних і простоту управління документно-орієнтованими структурами.

Було створено та протестовано додаток із використанням технології управління голосовим інтерфейсом. При побудові серверної частини додатку було створено гнучку структуру з дотриманням принципів MVC. При розробці клієнтської частини додатки використано кілька патернів та дотримано принцип OCP. В майбутньому можна буде розширити функціонал додатку, використовуючи при цьому нові технології машинного навчання для аналізу команд схожих за сенсом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft Azure. Voice Service APIs [Електронний ресурс]. Режим доступу до ресурсу: https://azure.microsoft.com/en-us/services/cognitive-services/text-to-speech/
2. IBM Watson Speech to Text [Електронний ресурс]. Режим доступу до ресурсу: https://www.ibm.com/products/speech-to-text
3. Microsoft. Cortana [Електронний ресурс]. Режим доступу до ресурсу: https://support.microsoft.com/uk-ua/topic/cortana-%D1%82%D0%B0-%D0%BA%D0%BE%D0%BD%D1%84%D1%96%D0%B4%D0%B5%D0%BD%D1%86%D1%96%D0%B9%D0%BD%D1%96%D1%81%D1%82%D1%8C-47e5856e-3680-d930-22e1-71ec6cdde231
4. Amazon [Електронний ресурс]. Alexa Skills Kit Documentation. Режим доступу до ресурсу: https://developer.amazon.com/en-US/alexa/alexa-skills-kit
5. Angular [Електронний ресурс]. Режим доступу до ресурсу: https://angular.dev/
6. NestJS [Електронний ресурс]. Режим доступу до ресурсу: https://nestjs.com/
7. Web Speech API [Електронний ресурс]. Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API
8. Refactoring guru [Електронний ресурс]. Режим доступу до ресурсу: https://refactoring.guru/uk
9. Mongoose [Електронний ресурс]. Режим доступу до ресурсу: https://docs.nestjs.com/techniques/mongodb
10. GOF patterns [Електронний ресурс]. Режим доступу до ресурсу: https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns
11. Material UI [Електронний ресурс]. Режим доступу до ресурсу: https://material.angular.io/
12. You don’t know JS [Електронний ресурс]. Режим доступу до ресурсу: https://github.com/getify/You-Dont-Know-JS
13. Learning JavaScript Design Patterns [Електронний ресурс]. Режим доступу до ресурсу: https://addyosmani.com/resources/essentialjsdesignpatterns/book/
14. Swagger [Електронний ресурс]. Режим доступу до ресурсу: https://swagger.io/
15. Postman [Електронний ресурс]. Режим доступу до ресурсу: https://www.postman.com/
16. Don’t Overcomplicate Your Angular App [Електронний ресурс]. Режим доступу до ресурсу: https://golosay.medium.com/dont-overcomplicate-your-angular-app-e4e8886e518f
17. Host Decorators Are Dead… Use Host Element Binding Instead [Електронний ресурс]. Режим доступу до ресурсу:

https://medium.com/@mrbriantreese/angular-tutorial-host-element-binding-92664f87eae5
18. Common web application architectures [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/ru-ru/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures
19. What is REST [Електронний ресурс]. Режим доступу до ресурсу: https://restfulapi.net/
20. Clean Architecture [Електронний ресурс]. Режим доступу до ресурсу: https://github.com/jasontaylordev/CleanArchitecture
21. Introducing Whisper [Електронний ресурс]. Режим доступу до ресурсу: https://openai.com/index/whisper/?ref=refind
22. The Future of Speech Recognition: Where Will We Be in 2030? [Електронний ресурс]. Режим доступу до ресурсу: https://thegradient.pub/the-future-of-speech-recognition/?ref=refind
23. AI learns how to fool text-to-speech. That's bad news for voice assistants [Електронний ресурс]. Режим доступу до ресурсу: https://thenextweb.com/news/ai-learns-how-to-fool-text-to-speech-thats-bad-news-for-voice-assistants?ref=refind
24. An Edtech User's Glossary to Speech Recognition and AI in the Classroom [Електронний ресурс]. Режим доступу до ресурсу: https://www.edsurge.com/news/2021-09-02-an-edtech-user-s-glossary-to-speech-recognition-and-ai-in-the-classroom?ref=refind
25. Neuroscientists decode brain speech signals into actual sentences [Електронний ресурс]. Режим доступу до ресурсу: https://www.theguardian.com/science/2019/jul/30/neuroscientists-decode-brain-speech-signals-into-actual-sentences?ref=refind
26. Building a Layered Architecture in NestJS & Typescript: Repository Pattern, DTOs, and Validators [Електронний ресурс]. Режим доступу до ресурсу: https://medium.com/@patrick.cunha336/building-a-layered-architecture-in-nestjs-typescript-repository-pattern-dtos-and-validators-08907a8ac4cb
27. Building Microservices with NestJS, TCP and Typescript [Електронний ресурс]. Режим доступу до ресурсу: https://medium.com/itnext/building-microservices-with-nestjs-tcp-and-typescript-dda33aad8b89
28. MongoDB [Електронний ресурс]. Режим доступу до ресурсу: https://medium.com/@ananthvishnu/mongodb-b95e8364ceed
29. What Is MongoDB? Working, Architecture, Features, and Use Cases [Електронний ресурс]. Режим доступу до ресурсу: https://www.spiceworks.com/tech/cloud/articles/what-is-mongodb/
30. Hello, Angular 19 [Електронний ресурс]. Режим доступу до ресурсу: https://medium.com/@sehban.alam/hello-angular-19-77fb1691bbb5

31. Angular Signals, Reactive Context, and Dynamic Dependency Tracking [Електронний ресурс]. Режим доступу до ресурсу: https://medium.com/@eugeniyoz/angular-signals-reactive-context-and-dynamic-dependency-tracking-d2d6100568b0
32. Stop Using @ViewChild/Children Decorators! Use Signals Instead [Електронний ресурс]. Режим доступу до ресурсу: https://itnext.io/stop-using-viewchild-children-decorators-use-signals-instead-5bcfd7d26e55
33. Top design patterns for frontend [Електронний ресурс]. Режим доступу до ресурсу: https://dev.to/superviz/top-design-patterns-for-frontend-1bk5
34. Exploring the Power of the Web Speech API: Revolutionizing User Interaction [Електронний ресурс]. Режим доступу до ресурсу: https://gili842.medium.com/exploring-the-power-of-the-web-speech-api-revolutionizing-user-interaction-9be2d1d74324
35. Git concepts in less than 10 minutes [Електронний ресурс]. Режим доступу до ресурсу: https://opensource.com/article/22/11/git-concepts
36. 5 Git configurations I make on Linux [Електронний ресурс]. Режим доступу до ресурсу: https://opensource.com/article/22/9/git-configuration-linux
37. How to rename a branch, delete a branch, and find the author of a branch in Git [Електронний ресурс]. Режим доступу до ресурсу: https://opensource.com/article/22/5/git-branch-rename-delete-find-author
38. Webstorm [Електронний ресурс]. Режим доступу до ресурсу: https://www.jetbrains.com/webstorm/
39. Material design blog [Електронний ресурс]. Режим доступу до ресурсу: https://m3.material.io/blog
40. 10 fundamental web accessibility tips for front-end developers [Електронний ресурс]. Режим доступу до ресурсу: https://www.frontendmentor.io/articles/10-fundamental-web-accessibility-tips-for-frontend-developers-rUurADGxCt
41. A Complete Guide To Accessible Front-End Components [Електронний ресурс]. Режим доступу до ресурсу: https://www.smashingmagazine.com/2021/03/complete-guide-accessible-front-end-components/
42. Веб-доступність. Що варто знати кожному Front-end розробнику і дизайнеру [Електронний ресурс]. Режим доступу до ресурсу: https://dou.ua/lenta/articles/web-accessibility/

ДОДАТОК А

JSON конфігурація колекції запитів у Postman

{ "info": { "_postman_id": "696be41e-5a85-4b75-aa9f-5ed83e7f3337", "name": "Diploma", "description": "Collection for testing the Ecommerce API with Products and Orders endpoints", "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json" }, "item": [{ "name": "Products", "item": [{ "name": "Create Product", "request": { "method": "POST", "header": [{ "key": "Content-Type", "value": "application/json", "type": "text" }], "body": { "mode": "raw", "raw": "{\n \"name\": \"Apple\",\n \"price\": 1.2,\n \"stock\": 100,\n \"description\": \"A fresh and juicy apple\",\n \"category\": \"Fruits\"\n}" }, "url": { "raw": "http://localhost:3000/products", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["products"] } }, "response": [] }, { "name": "Get All Products", "request": { "method": "GET", "header": [], "url": { "raw": "http://localhost:3000/products", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["products"] } }, "response": [] }, { "name": "Get Product By ID", "request": { "method": "GET", "header": [], "url": { "raw": "http://localhost:3000/products/:id", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["products", ":id"], "variable": [{ "key": "id", "value": "replace_with_product_id" }] } }, "response": [] }, { "name": "Update Product", "request": { "method": "PUT", "header": [{ "key": "Content-Type", "value": "application/json", "type": "text" }], "body": { "mode": "raw", "raw": "{\n \"name\": \"Updated Apple\",\n \"price\": 1.75,\n \"description\": \"Fresh red apple, new price\",\n \"category\": \"Fruits\",\n \"stock\": 120\n}" }, "url": { "raw": "http://localhost:3000/products/:id", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["products", ":id"], "variable": [{ "key": "id", "value": "replace_with_product_id" }] } }, "response": [] }, { "name": "Delete Product", "request": { "method": "DELETE", "header": [], "url": { "raw": "http://localhost:3000/products/:id", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["products", ":id"], "variable": [{ "key": "id", "value":

"673091e0536fef414dd90f11" }] } }, "response": [] }] }, { "name": "Orders", "item": [{ "name": "Create Order", "request": { "method": "POST", "header": [{ "key": "Content-Type", "value": "application/json", "type": "text" }], "body": { "mode": "raw", "raw": "{\n \"items\": [\n { \"productId\": \"replace_with_product_id_1\", \"quantity\": 2 },\n { \"productId\": \"replace_with_product_id_2\", \"quantity\": 3 }\n]\n}" }, "url": { "raw": "http://localhost:3000/orders", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["orders"] } }, "response": [] }, { "name": "Get All Orders", "request": { "method": "GET", "header": [], "url": { "raw": "http://localhost:3000/orders", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["orders"] } }, "response": [] }, { "name": "Get Order By ID", "request": { "method": "GET", "header": [], "url": { "raw": "http://localhost:3000/orders/:id", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["orders", ":id"], "variable": [{ "key": "id", "value": "replace_with_order_id" }] } }, "response": [] }, { "name": "Delete Order By ID", "request": { "method": "DELETE", "header": [], "url": { "raw": "http://localhost:3000/orders/:id", "protocol": "http", "host": ["localhost"], "port": "3000", "path": ["orders", ":id"], "variable": [{ "key": "id", "value": "replace_with_order_id" }] } }, "response": [] }] }] }