ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО «ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД «МІЖРЕГІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ ПЕРСОНАЛОМ»

Факультет комп'ютерно-інформаційних технологій

Кафедра комп'ютерних інформаційних систем і технологій

Кучер Андрій Євгенович

КВАЛІФІКАЦІЙНА РОБОТА

на тему: **«Веб-інформаційна система управління заявками у торговельній мережі»**

Група: ІК-9-22-Б1ІПЗ(4.0ддс)

Спеціальність: 121 «Інженерія програмного забезпечення»

Рівень вищої освіти: перший (бакалаврський)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів, текстів інших авторів мають посилання на відповідне джерело.

_____ Кучер А. Є.
(підпис)

Науковий керівник

_____ Знаковська Є. А.
(підпис)

Допущено до захисту ЕК

Завідувач кафедри _____ Сергій КАВУН , д.е.н., професор
(підпис)

Київ 2026

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 108 с., 9 рис., 2 додатки, 25 джерел.

МУЛЬТИАГЕНТНІ СИСТЕМИ, ДЕЦЕНТРАЛІЗОВАНА ТОРГІВЛЯ, АСИНХРОННА ВЗАЄМОДІЯ, PYTHON, FASTAPI, SPADE, XMPP, SQLITE.

Об’єктом дослідження є процеси децентралізованого керування ресурсами та електронної комерції в розподілених мережах з використанням мультиагентних технологій.

Метою роботи є розробка вебплатформи для децентралізованої торгівлі, в якій процес пошуку товарів та оформлення замовлень виконується через асинхронну взаємодію мережі незалежних інтелектуальних агентів для створення автономного середовища.

Методи розробки базуються на мові програмування Python з використанням фреймворків FastAPI (для вебінтерфейсу), SPADE (для агентного середовища), протоколу XMPP та локальної бази даних SQLite.

Результатом роботи є розробка розподіленої торговельної системи, архітектурні рішення якої забезпечують автоматичне перенаправлення замовлень та транзакційний захист даних у випадку виникнення локального дефіциту чи мережевих збоїв.

MULTI-AGENT SYSTEMS, DECENTRALIZED TRADE, ASYNCHRONOUS INTERACTION, PYTHON, FASTAPI, SPADE, XMPP, SQLITE.

The object of research is the processes of decentralized resource management and e-commerce in distributed networks using multi-agent technologies.

The aim of the work is to develop a web platform for decentralized e-commerce, where the process of searching for products and checkout is performed through asynchronous interaction of a network of independent intelligent agents to create an autonomous environment.

The development methods are based on the Python programming language using the FastAPI framework (for the web interface), SPADE (for the agent environment), the XMPP protocol, and the local SQLite database.

The result of the work is the development of a distributed trading system, the architectural solutions of which provide automatic order redirection and transactional data protection in case of local shortages or network failures.

Зміст

ВСТУП

Використання децентралізованих підходів стрімко впроваджується в різноманітні процеси сучасного бізнесу та інформаційних технологій. Причиною цього можна вважати переваги, які вони надають електронній комерції. Серед них є можливість відмовитися від єдиної централізованої бази даних, яка часто стає критичним вузлом під час великого навантаження або паралельного опитування багатьох розподілених складів. Процес синхронізації даних у класичних системах складний та призводить до появи застарілих даних. У зв'язку з цим, одним із методів розв’язання цієї проблеми можна вважати застосування технологій мультиагентних систем.

Тема побудови розподілених торговельних майданчиків за допомогою взаємодії незалежних інтелектуальних агентів завжди є актуальною задачею. Отримання такої моделі має багато практичних цінностей. Одна із них — це можливість створити автономне та гнучке середовище, де кожен склад або магазин самостійно контролює власні ресурси. Об’єктом розробки вебплатформи для децентралізованої торгівлі є використання спеціалізованих агентних платформ та комунікаційних протоколів. Головною складовою до розробки кваліфікаційної роботи є вивчення принципів побудови мультиагентних систем на базі фреймворку SPADE та реалізація асинхронного шлюзу між вебінтерфейсом і мережею агентів.

Метою дипломної роботи є розробка вебплатформи для децентралізованої торгівлі з використанням мультиагентної архітектури та асинхронної взаємодії. Система має забезпечувати надійний пошук товарів та оформлення замовлень через незалежних агентів. Додатково слід застосувати методи транзакційного контролю для збереження цілісності даних у базі даних при виникненні мережевих збоїв.

Для виконання поставленої задачі необхідно:

− проаналізувати джерела літератури та сучасні архітектурні підходи в електронній комерції;

− виконати детальний аналіз та обґрунтувати вибір технологічного стеку (FastAPI, SPADE, XMPP);

− виконати аналіз та спроєктувати дворівневу архітектуру розподіленої системи;

− розробити асинхронний комунікаційний шлюз для трансляції HTTP-запитів із контролем сесій через thread_id;

− реалізувати алгоритм пріоритетного локального пошуку, що оптимізує використання внутрішніх ресурсів системи та автоматично активує резервний канал партнера у випадку дефіциту товарів;

− забезпечити стійкість та базовий захист системи за допомогою модулів автентифікації та транзакційного відкату баз даних;

− виконати програмну розробку та тестування створеного прототипу системи.

Методи дослідження. В роботі застосовані методи об'єктно-орієнтованого та асинхронного програмування, архітектурні підходи побудови мультиагентних систем на базі протоколу XMPP, а також методи транзакційного керування реляційними базами даних.

Практичне значення одержаних результатів. Сфер використання цієї системи безліч - роздрібна онлайн-торгівля, розподілені логістичні мережі, B2B-платформи, системи управління ланцюгами постачання та електронні маркетплейси. Наприклад, розроблені агентні модулі можуть динамічно кооперуватися для покриття великих замовлень з різних складів без участі центрального сервера. Розроблена система є досить гнучкою та практичною до використання.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ РОЗРОБКИ СИСТЕМИ

1.1. Аналіз предметної області

В останні роки сфера роздрібної торгівлі зазнала помітних змін, що значною мірою пов’язані з активним переходом бізнесу до онлайн-середовища [1]. Сучасна електронна комерція (e-commerce) уже не обмежується лише інтернет-магазином як окремим вебресурсом [2] [3]. Фактично йдеться про складну інформаційну інфраструктуру, у межах якої безперервно взаємодіють торговельні платформи, склади, постачальники та кінцеві споживачі.

Причини активного розвитку систем управління в галузі роздрібної торгівлі мають комплексний характер.

Насамперед увагу привертає висока динамічність ринкового середовища [4]. Інформація щодо вартості товарів або їх залишків на складах змінюється практично безперервно. Через значний потік одночасних запитів централізовані системи не завжди здатні синхронізувати дані між усіма вузлами в реальному часі. Як наслідок, користувачі можуть отримувати неактуальні відомості про наявність продукції, а сама система - формувати помилки типу «stale data» або некоректно оформлювати замовлення.

Ще одним фактором стала розподіленість сучасної інфраструктури складів і постачальників. Більшість торговельних платформ функціонує одразу з кількома географічно віддаленими вузлами та зовнішніми партнерськими сервісами [5]. За таких умов обмін даними між компонентами системи відбувається постійно. І що більшою стає кількість складів та клієнтських звернень, то складніше підтримувати актуальність залишків товару та стабільну роботу центральних компонентів системи.

Не менш суттєво зросли й вимоги користувачів. Сьогодні очікується миттєве підтвердження замовлення, швидке оновлення інформації про доступність товару та можливість контролю доставки практично в режимі реального часу. Для перевірки таких даних система часто змушена одночасно взаємодіяти з кількома сервісами або базами даних. У ході аналізу вдалося

помітити, що навіть незначні затримки мережі чи перевантаження серверів безпосередньо впливають на швидкість оформлення замовлень і загальну якість обслуговування користувачів [6].

За подібних умов класичні централізовані підходи до побудови інформаційних систем уже не завжди демонструють достатній рівень масштабованості та відмовостійкості. Якщо навантаження на центральний сервер зростає або виникають перебої в каналах зв'язку, продуктивність усієї системи може істотно погіршуватись [7]. Саме тому сучасний ринок електронної комерції дедалі більше потребує гнучких архітектурних рішень, здатних ефективно працювати в середовищі постійних змін та великої кількості паралельних процесів.

1.2 Аналіз архітектурних підходів та існуючих рішень

Архітектура програмного забезпечення є основою для будь-якої інформаційної системи і визначає структуру та взаємодію компонентів системи [8]. Розгляд архітектури при проектуванні системи важливий з наступних міркувань:

- визначення технічних вимог та необхідних ресурсів, що передбачає визначення технології, інструментів та обсягу робіт;
- виявлення вузьких місць та невідповідність вимогам. На етапі проектування це може бути уточнення певних технічних рішень та визначення помилок, збоїв, що можуть виникати виходячи з архітектури рішення;
- забезпечення можливості масштабованості та стійкості системи.

При вирішенні завдань розробки при сучасному проектуванні розрізняють різні архітектурні парадигми, які пропонують власні способи організації логіки та взаємодію між компонентами системи [9].

На сьогоднішній день основна більшість сучасних веб-інформаційних систем будуються на основі клієнт-серверної архітектури. Вона передбачає чітке розділення ролей і в реалізація умовно розкладається на клієнтську частину, яка безпосередньо взаємодіє з користувачем та серверну, що відповідає за обробку даних та бізнес-логіку. Така будова дозволяє зосередити

найбільш складні та ресурсозатратні операції з обробки даних на стороні сервера. Проте внутрішня організація серверної частини може реалізуватися під різними архітектурними підходами.

В залежності від способу декомпозиції бізнес-логіки та управління даними в рамках цієї парадигми можна виділити три основних еволюційних етапів серверної архітектури: монолітна , мікросервісна архітектура та подійно-оріентовані системи [9].

Централізована (монолітна) архітектура. Це традиційний підхідв якому компоненти системи (інтерфейс, бізнес-логіка, робота с даними) об'єднуються в єдиний модуль, що реалізує весь функціонал системи. Реалізація компонентів в єдиному модулі передбачає:

- Простоту та легкість розробки адже всі дані зосереджені в одному модулі.
- Продуктивність та швидкодію.
- В єдиному модулі дані не потрібно без потреби перетворювати до певного виду для обробки.

Враховуючи переваги як один з методів рішення монолітний підхід має свої обмеження та недоліки, які слід враховувати при виборі архітектури майбутньої системи.

До основних недоліків можна віднести:

1) Складність масштабування. У разі зростання навантаження виникає потреба дублювати всю систему, що може призвести до надмірного перевантаження використовуваних обчислювальних ресурсів

2) Висока зв'язність коду проекту. Локальна помилка в одній частині проекту може призвести до повної відмови всього додатку

3) Складність оновлення. Застосунок реалізований на конкретному технологічному стеку та з використанням мови при оновленні версій бібліотек може потребувати повного переписування деяких частин коду.

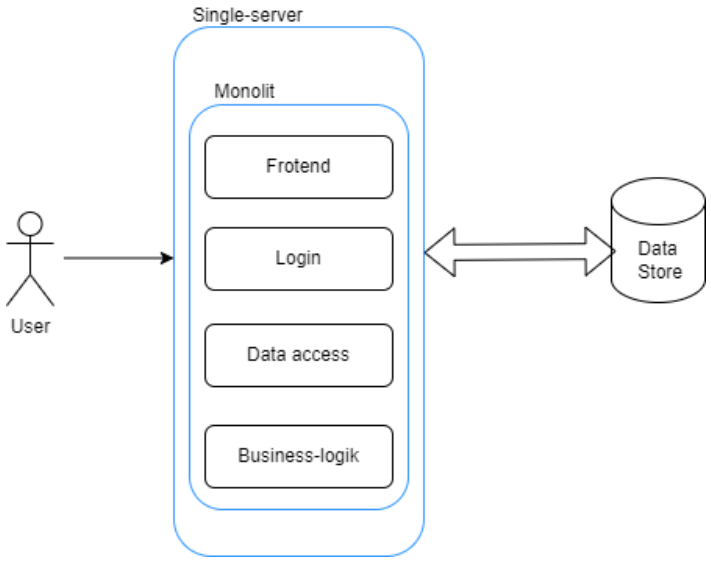


Рис 1.1 - Загальна схема монолітної архітектури

Таким чином, монолітна архітектура може бути ефективною для невеликих локальних систем, однак при роботі з великою кількістю незалежних вузлів її використання ускладнює масштабування та адаптацію системи до змін.

При вирішенні зазначених проблем монолітного підходу в сучасній розробці використовують мікросервісну архітектуру. При такому підході масштабний застосунок реалізується як набір невеликих та незалежних сервісів. Взаємодія між сервісами передбачає встановлення чітко визначеного інтерфейсу API. Сервіс, в такому випадку, реалізує конкретну функцію в бізнес-логіці.

Такий підхід має певні переваги:

1) Незалежне масштабування. Можливість надання більших обчислюваних ресурсів для того сервісу, який більш навантажений в даний момент

2) Ізоляція збоїв та висока відмовостійкість. У разі відмови одного з сервісу додаток продовжує працювати без функції, яка покладалася на сервіс, що вийшов з ладу

3) Технологічна гнучкість. Використання для написання сервісу того технологічного стеку та мови програмування , які найбільш підходять для вирішення поставлених задач.

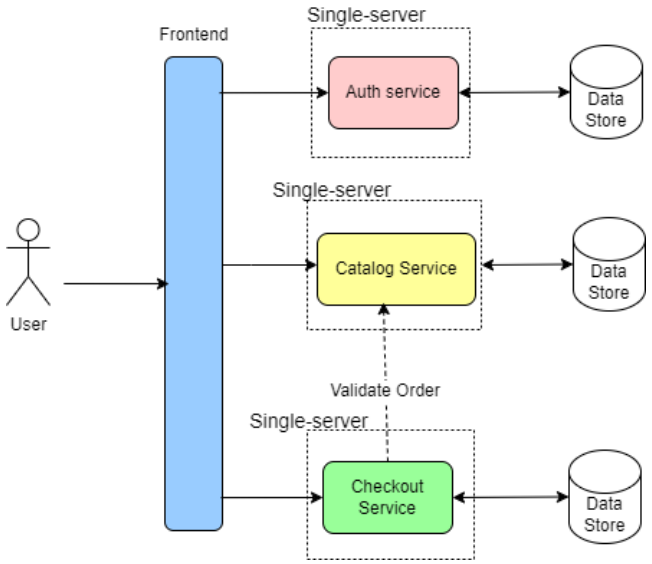


Рис 1.2 - Загальна схема мікросервісної архітектури

Необхідно зазначити і певні недоліки, які можуть проявлятися при проектуванні та мати суттєвий характер:

1) Складність проектування та декомпозиція. На відміну від моноліту, в певних випадках, достатньо складно розділити систему на ізольовані сервіси. Помилки на цьому етапі можуть призвести до заплутаної та неефективної архітектури.

2) Накладні витрати на обробку даних. Пов’язано з тим, що сервіси проектуються як ізольовані сутності, а для взаємодії вони повинні дотримуватись одного формату даних (наприклад JSON) , що накладає певні затримки у часі для конвертації з одного формату в інший дані (наприклад, при мережевій передачі).

3) Синхронна залежність. Найбільш чітко цей недолік проявляється при обміні даними між сервісами за допомогою HTTP-протоколу. Певний сервіс змушений очікувати відповіді від іншого сервісу, що у разі збоїв або високого навантаження уповільнює роботу всього додатка.

Отже, мікросервісний підхід забезпечує кращу гнучкість у порівнянні з монолітом, проте потребує складнішої координації між сервісами та додаткових механізмів управління взаємодією.

Для подолання проблеми синхронної залежності та зменшення затримок при взаємодії мікросервісів у сучасних високонавантажених системах застосовують подійно-орієнтовану архітектуру (Event-DrivenArchitecture). Цей підхід повністю змінює принцип обміну даними між компонентами системи, переводячи його у повністю асинхронний режим.

Головна особливість такої реалізації полягає в тому, що сервіси більше не викликають один одного напряму. Для цього в архітектуру впроваджується спеціальний посередник - брокер повідомлень найпопулярніші з яких ApacheKafka або RabbitMQ [10] .

Таке архітектурне рішення передбачає, що у випадку виникнення певної дії (події) в одному з сервісів він публікує повідомлення про це в брокер та продовжує свою роботу при цьому не очікуючи миттєвої відповіді від інших сервсів. Інші сервіси, яким потрібна ця інформація, підписані на відповідні події та обробляють їх автономно.

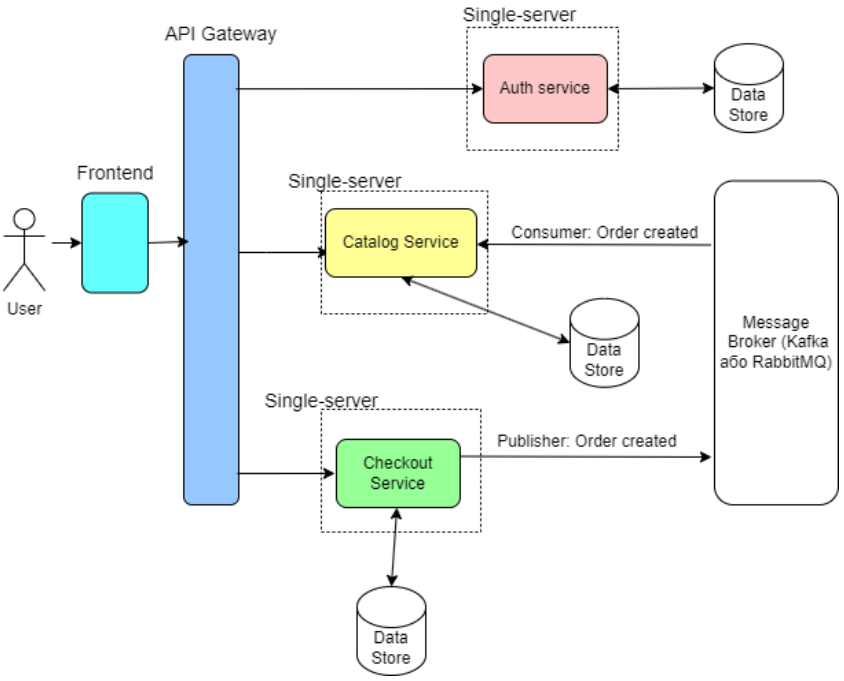


Рис 1.3 - Узагальнена схема подійно-орієнтованої архітектури

Подієво-орієнтована архітектура як асинхронне рішення для микросервісної архітектури дозволяє вирішувати задачі де важливим стають: можливість масштабованості системи, швидка відповідь клієнту та довга фонова робота. Це можуть бути задачі офрмлення замовлення, аналітики, логістики. В той же час для задач де відповідь повинна бути миттєвою (наприклад, перевірка балансу, авторизація, отримання точної ціни товару) доцільно використовувати синхронні мікросервіси.

Розглядані системи, починаючи з мікросервісів, відносять до розподілених систем через фізичну розподіленності компонентів таких систем (компоненти можуть знаходитись на різних вузлах). Хоча слід зазначити, що розподілені системи є достатньо широким класом до якого відносять такі сучасні рішення як сервісно-орієнтована архітектура (SOA), безсерверна архітектура (Serverless architecture)та інші. Проте, попри фізичну розподіленість, такі системи найчастіше залишаються логічно централізованими. Під централізацією, в даному випадку, слід розуміти зосередженність на єдиній бізнес-логіцірозробника і виокремлені сервіси є лише реактивними і пасивними виконавцями жорстко запрограмованого алгоритму. Натомість існує клас задач для яких така централізація і жорсткість існування алгоритму призводить до неефективного рішення або не існує взагалі. Ці задачі характеризуються загалом високою динамічністю, слабоструктурованістю та наявністю елементів невизначеності. Характерними прикладами таких задач є ройовий інтелект літальних апаратів, динамічний розподіл ресурсів в розподілених мережах, однорангові ринки та смарт-контракти. Тому вирішення подібних задач покладається на клас децентралізованих систем, що передбачає автономіність логіки управління на рівні кожного окремого вузла.

Клас децентралізованих систем включає кілька видів, що орієнтовані під конкретні задачі. Наприклад, однорангові мережі P2P використовують для обміну файлами чи ресурсами між рівноправними учасниками мережі (

наприклад, торрент). До іншого виду децентралізованих систем відносять технології розподіленого реєстру (Blockchain). Вони забезпечують консенсус та безпеку транзакцій у середовищі без взаємної довіри [11] .

Іншим представником цього класу є мультиагентні системи (МАС), в яких децентралізація проявляється на рівні прийняття рішень. У цьому випадку процес керування потоками даних та ресурсів набуває інтелектуального характеру та супроводжується гнучкою координацією дій між усіма учасниками мережі [12]. Учасниками мережі виступають агенти – сутності, які мають власну поведінку. На відміну від розгляданих мікросервісівагентна поведінка характеризується проактивністю, тобто здатністю діяти в залежності від власних цілей і внутрішнього стану агента [13]. Класичні агенти створюються і діють в єдиному програмному середовищі. Незалежно від конкретної програмної реалізації (будь то класична платформа JADE чи сучасні розподілені рішення), архітектура будь-якої МАС базується на стандартах FIPA [12]. Згідно з цими стандартами, функціонування системи забезпечується двома обов'язковими сервісами управління:

- AMS (AgentManagementSystem) - сервіс, що керує життєвим циклом агентів, їх створенням, автентифікацією та унікальними адресами;
- DF (DirectoryFacilitator) - аналог "жовтих сторінок", де агенти реєструють свої послуги, щоб інші могли знайти їх для взаємодії (наприклад, магазин шукає доступні склади).

Взаємодія між агентами здійснюється за допомогою спеціальної мови FIPA-ACL (AgentCommunicationLanguage). Вона відрізняється від звичайних API-запитів тим, що кожне повідомлення містить не просто дані, а виражає комунікативний намір через стандартні типи повідомлень (перформативи).

У загальній архітектурі МАС використовуються такі базові типи перформативів:

- REQUEST - має імперативний (поведінковий) характер і використовується, коли один агент вимагає від іншого виконати певну дію;

- PROPOSE - виступає як офіційна пропозиція угоди або певних умов взаємодії, які інший агент може розглянути;
- ACCEPT-PROPOSAL / REJECT-PROPOSAL - використовуються для прийняття або, відповідно, відхилення раніше запропонованих умов;
- INFORM - служить для прямої передачі фактологічної інформації або сповіщення інших учасників про зміну стану середовища.

Крім самого типу повідомлення, пакет обов'язково містить метадані, необхідні для координації: унікальний ідентифікатор діалогу (conversation-id), ідентифікатор потоку обробки (thread) для підтримки паралельних розмов, мову кодування вмісту (language) та онтологію (ontology), яка визначає правила інтерпретації даних учасниками мережі.

Загальну схему внутрішньої структури інтелектуального агента та його взаємодії із середовищем наведено на рисунку 1.4.

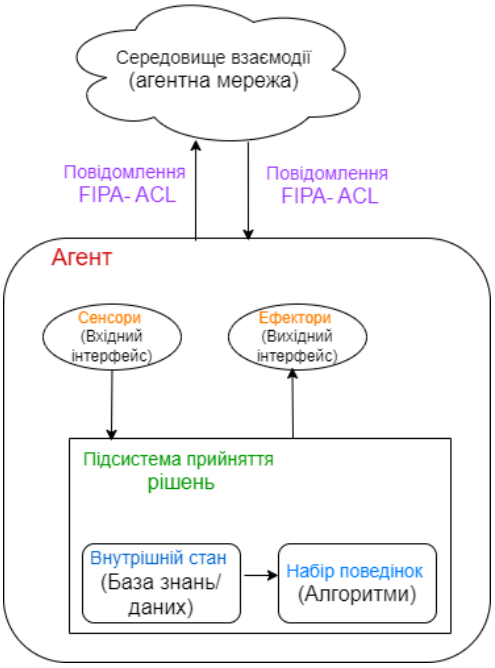


Рис 1.4 - Архітектурна схема внутрішньої структури та контуру взаємодії інтелектуального агента

Представлена на рисунку 1.4 структура відображає класичний кібернетичний контур взаємодії інтелектуального агента із зовнішнім

середовищем. По суті мова йде про базову модель за якою агент отримує інформацію, аналізує її та виконує відповідну дію.

Функціонування такого контуру забезпечується трьома основними підсистемами, кожна з яких виконує власну роль у процесі роботи агента.

1) **Підсистема сприйняття (сенсори)** відповідає за моніторинг середовища та отримання вхідних повідомлень від інших агентів мережі. Саме через сенсори агент дізнається про нові події або запити. Крім цього сенсори виконують первинний аналіз отриманих пакетів даних і виділяють службові метадані, наприклад ідентифікатори потоків або діалогів (thread).

2) **Підсистема прийняття рішень** фактично є центральною частиною агента. Тут поєднується внутрішній стан агента (локальні дані, поточні залишки товарів, статуси тощо) та набір моделей поведінки які визначають як саме агент повинен реагувати на певні події. Частина алгоритмів при цьому може бути жорстко визначеною, а частина — більш адаптивною. Саме на цьому рівні формується логіка відповіді на зовнішні запити або команди. Наприклад агент може прийняти рішення про передачу запиту іншому агенту, відмову у виконанні операції чи підтвердження резервування товару.

3) **Підсистема дії (ефектори)** забезпечує вже зворотний вплив агента на середовище. Для цього агент формує та надсилає вихідні повідомлення згідно стандартів FIPA-ACL, а також виконує зміни у власних локальних структурах даних. Наприклад оновлює кількість товару після списання або змінює статус заявки.

Після розгляду окремих архітектурних підходів можна перейти до невеликого порівняльного аналізу. Він передбачає розгляд систем за критеріями автономності, способу взаємодії між компонентами, адаптивності та механізмів вирішення конфліктів між учасниками системи.

Для більш наочного представлення відмінностей між архітектурними рішеннями була складена таблиця 1.1.

Проведений аналіз показує що монолітна архітектура залишається відносно простою у реалізації та підтримці, особливо для невеликих

проєктів.Проте при роботі в умовах розподіленої торговельної мережі її гнучкість вже стає недостатньою. Ускладнюється масштабування, а будь-які зміни можуть впливати одразу на всю систему.

Мікросервісний підхід у цьому плані виглядає більш гнучким і дозволяє краще масштабувати окремі компоненти.Однак разом із цим з’являються і нові складності пов’язані з координацією сервісів, синхронізацією даних та підтримкою стабільної взаємодії між вузлами системи. Особливо це помітно при великій кількості міжсервісних запитів.

У свою чергу мультиагентний підхід дозволяє організувати більш автономну взаємодію між окремими компонентами системи. Кожен агент працює як окрема незалежна сутність зі своєю логікою та локальними даними. Через це система стає більш адаптивною до змін середовища і краще підходить для задач пов’язаних з динамічним пошуком та обробкою заявок.

Для систем електронної комерції архітектурний підхід впливає не лише на продуктивність. Не менш важливою є здатність системи працювати в умовах високого навантаження, розподіленості компонентів та постійної зміни даних між різними вузлами.

Таблиця 1.1 - Порівняльний аналіз архітектурних підходів

Критерій порівняння	**Монолітна архітектура**	**Мікросервісна архітектура**	**Мультиагентна система (MAS)**
Ступінь автономності	Відсутня (єдина система)	Середня (незалежні сервіси)	Висока (автономні агенти)
Тип взаємодії	Внутрішні виклики функцій	Синхронні/асинхронні API-запити	Асинхронні переговори (FIPA-ACL)
Реакція на зміни	Потребує перезапуску всієї системи	Потребує оновлення сервісу	Адаптивна (агенти змінюють поведінку)
Управління конфліктами	Блокування на рівні БД	Складні транзакційні протоколи	Механізм м'якого резервування та TTL
Поведінка вузлів	Пасивна	Пасивна	Проактивна та інтелектуальна

Серед найбільш важливих критеріїв при цьому можна виділити масштабованість, автономність окремих компонентів, спосіб взаємодії між вузлами а також стійкість системи до відмов окремих елементів. Саме ці фактори часто визначають наскільки ефективно система буде працювати у реальних умовах.

1.3 Обґрунтування вибору технологій та засобів реалізації

З урахуванням обраної архітектури системи виникла необхідність сформувати технологічний стек, здатний забезпечити як високу продуктивність веб-інтерфейсу, так і можливість побудови розподіленого мультиагентного середовища. У такому випадку вибір технологій безпосередньо впливає не лише на швидкодію системи, а й на стабільність взаємодії між окремими компонентами.

Першочерговим етапом стало визначення основної мови програмування, оскільки саме вона задає доступний набір бібліотек, інструментів та підходів до реалізації архітектури. У ході аналізу було обрано Python. Таке рішення обумовлене кількома причинами.

Насамперед Python забезпечує підтримку асинхронного програмування через бібліотеку asyncio [14]. Це дозволяє ефективно обробляти велику кількість повідомлень між агентами без блокування основних процесів системи. Для мультиагентного середовища така можливість є критично важливою. Крім того, значна кількість готових бібліотек для роботи з мережевими протоколами, базами даних та асинхронною взаємодією дає змогу зосередитися безпосередньо на проектуванні логіки системи, а не на реалізації низькорівневих механізмів.

Традиційно одним із найбільш відомих фреймворків для побудови розподілених мультиагентних систем вважається JADE, реалізований мовою Java. Проте в процесі аналізу було виявлено низку обмежень, пов’язаних із його використанням у сучасному вебсередовищі. Насамперед йдеться про складність інтеграції з актуальними вебтехнологіями, високий поріг входження та достатньо «важку» архітектуру самого фреймворку. Саме тому для реалізації

агентного шару системи було обрано SPADE (Smart Python Agent Development Environment). Даний фреймворк створювався як сучасна альтернатива JADE та поєднав підходи до розробки агентів із перевагами екосистеми Python [15]. SPADE розвивається як сучасна платформа для мультиагентних систем, зокрема у версії SPADE 3, яка підтримує нове покоління розподілених MAS [16] .

Вибір SPADE для побудови агентного шару пояснюється кількома особливостями.

- Фреймворк підтримує мову комунікації ACL (Agent Communication Language), що є важливим для реалізації контекстної взаємодії між агентами.
- Для обміну повідомленнями використовується протокол XMPP [17]. Це дозволяє агентам швидко передавати дані навіть у випадку розміщення на різних вузлах мережі та спрощує проходження мережевих обмежень і брандмауерів.
- SPADE підтримує асинхронну модель виконання, завдяки чому агенти можуть паралельно виконувати декілька задач без блокування ресурсів системи.
- Фреймворк містить готові поведінкові шаблони (Cyclic, OneShot, Periodic), які значно спрощують реалізацію складної логіки взаємодії між агентами.
- У версії SPADE 4.1.2 передбачена можливість використання вбудованого XMPP-сервера, що суттєво полегшує локальне тестування системи [18] .

Для реалізації взаємодії між браузерним клієнтом і агентним середовищем виникла потреба у швидкому та стабільному API-шарі на основі REST-архітектури [19]. Його завданням стало забезпечення роботи серверної частини застосунку та передача запитів до агентної системи. З цією метою було використано FastAPI [20].

Важливий моментом є те, що FastAPI поєднує високу продуктивність із достатньо простою структурою розробки. Серед основних переваг цього фреймворку можна виділити:

- високу швидкодію, що є важливою під час передачі запитів між користувачем і агентами;
- вбудовану підтримку Swagger, яка значно спрощує тестування API та перевірку роботи ендпоінтів взаємодії з агентами.
- підтримку REST-архітектури, що забезпечує простоту та масштабованість взаємодії між компонентами системи.

Окрему увагу під час проектування системи було приділено ізоляції середовищ виконання вебчастини та агентного шару. На практиці вдалося виявити конфлікт версій спільних залежностей, зокрема бібліотеки шаблонізації Jinja2. Причина полягала в тому, що FastAPI орієнтується на сучасні версії пакетів, тоді як SPADE має жорсткі обмеження щодо сумісності залежностей для забезпечення стабільної асинхронної XMPP-комунікації.

Рішенням стало логічне та фізичне розділення API-шару й агентної частини системи. У результаті було реалізовано окремий проміжний Agent API-шар на базі бібліотеки aiohttp, яка не створювала конфліктів із залежностями SPADE [21]. Подібне розділення значно спростило тестування компонентів системи та дозволило використовувати незалежні віртуальні середовища (venv) для кожного модуля. Завдяки цьому оновлення вебінтерфейсу більше не створює ризику порушення логіки взаємодії агентів за стандартом FIPA-ACL.

Для локального зберігання даних агентів, користувацьких сесій та службової інформації було обрано SQLite. Вбудована підтримка цієї СУБД у Python дала можливість швидко організувати роботу з даними як на стороні клієнтської частини [22].

Під час побудови користувацького інтерфейсу було використано комбінований підхід, що поєднує серверний рендеринг і клієнтську логіку. Основні інструменти , що були при цьому використані :

- **Jinja2** застосовується на стороні FastAPI для генерації динамічних HTML-сторінок. Це дозволяє передавати початкові параметри сесії, зокрема ідентифікатор користувача або стан авторизації, ще на етапі завантаження сторінки [23].
- **JavaScript (Vanilla JS)** використовується для асинхронного оновлення даних на стороні клієнта. Оскільки MAS функціонує в асинхронному режимі, саме JavaScript забезпечує обробку відповідей від агентів — змін цін, оновлення статусів замовлень або повідомлень про резерв товару — без необхідності повного перезавантаження сторінки.

1.4 Висновки до першого розділу

У першому розділі кваліфікаційної роботи виконано комплексний аналіз предметної області електронної комерції та детально розглянуто сучасні архітектурні підходи до побудови інформаційних систем. На основі аналізу монолітної, мікросервісної та подійно-орієнтованої архітектур було сформовано обґрунтування розробки системи на базі децентралізованого мультиагентного підходу (МАС) за стандартами FIPA, оскільки класичні централізовані рішення мають обмеження при синхронізації даних на розподілених складах. Визначено, що використання автономних інтелектуальних агентів дозволяє ефективно координувати потоки даних через асинхронні переговори мовою FIPA-ACL. Для практичної реалізації платформи розглянуто та обрано оптимальний технологічний стек на базі мови Python (бібліотека asyncio), фреймворку SPADE 4.1.2 з вбудованим XMPP-сервером, REST-фреймворку FastAPI та СУБД SQLite. Важливим інженерним результатом розділу стало вирішення проблеми конфлікту версій залежностей Jinja2 шляхом обґрунтування логічного розділення середовищ виконання та впровадження

проміжного шару Agent API на базі aiohttp, що формує надійну базу для подальшого проєктування системи.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ВЕБ-ІНФОРМАЦІЙНОЇ СИСТЕМИ КЕРУВАННЯ ЗАЯВКАМИ

2.1 Функціональні та нефункціональні вимоги

Функціональні вимоги (табл. 2.1) описують конкретні дії, поведінку та функції, які має виконувати система.

Таблиця 2.1 - Функціональні вимоги

ФВ1	Користувач може авторизуватись за логіном і паролем перед входом в систему
ФВ2	Користувач може виконувати пошук товарів
ФВ3	Система повинна відображати найвигіднішу пропозицію товару
ФВ4	Система повинна підтримувати формування користувацького кошика
ФВ5	Перед оформленням заявки система повинна виконувати повторну перевірку актуальності даних
ФВ6	Система повинна підтримувати взаємодію з декількома постачальниками
ФВ7	Агент - координатор повинен виконувати автоматичний вибір найбільш вигідної пропозиції
ФВ8	Система повинна підтримувати часткове виконання заявки
ФВ9	Користувач повинен отримувати повідомлення про зміну умов замовлення

Нефункціональні вимоги (табл. 2.2) описують наскільки добре працює система, звертаючись до атрибутів якості, як продуктивність, безпека та масштабованість

Таблиця 2.2 - Нефункціональні вимоги

НВ1	Авторизація повинна бути захищеною (паролі хешуються)
НВ2	Система повинна підтримувати асинхронну взаємодію між компонентами

НВ3	Компоненти системи повинні працювати в ізольованих середовищах
НВ4	Обмін повідомленнями між агентами повинен виконуватись через XMPP
НВ 5	Система повинна підтримувати роботу у сучасних браузерах
НВ6	Дані про заявки повинні зберігатись у локальній базі даних SQLite

2.2. Сценарії варіантів використання

Сценарій 1. Пошук товару

Актор:Користувач

Мета: Отримати актуальні пропозиції товару

Основний сценарій:

1. Користувач вводить назву товару у форму пошуку.
2. FastAPI передає запит до агентного шлюзу MAS API.
3. Координатор StoreAAgent виконує опитування складів.
4. Система повертає найвигіднішу пропозицію та загальну кількість товару.
5. Користувач може додати товар до кошика.

Сценарій 2. Верифікація актуальності товару перед оформленням заявки

Актор:Користувач

Мета: Перевірити актуальність ціни та доступності товару перед оформленням заявки.

Основний сценарій:

1. Користувач переходить до оформлення заявки після формування кошика.
2. FastAPI передає параметри товару до агентного шлюзу MAS API.
3. Шлюз формує повідомлення з типом операції "verify" та надсилає його координатору StoreAAgent.

4. Координатор повторно опитує склади та перевіряє актуальність ціни і кількості товару.

5. Якщо умови не змінилися, система повертає статус "ok" і дозволяє продовжити оформлення заявки.

6. Якщо ціна або кількість змінились, система повертає статус "alert" та оновлені параметри товару.

7. Користувач підтверджує або скасовує оформлення заявки.

Сценарій 3. Оформлення та виконання заявки

Актор:Користувач

Мета: Оформити заявку та виконати списання товару зі складів або зовнішнього постачальника.

Основний сценарій:

1. Після успішної верифікації користувач підтверджує оформлення заявки.

2. Агентний шлюз викликає роут handle_checkout() та надсилає повідомлення з типом операції "sell" координатору StoreAAgent.

3. Координатор визначає найбільш вигідні джерела товару та формує повідомлення на списання товару.

4. Для локальних складів використовується перформативrequest, а для зовнішнього партнера StoreBAgent — propose.

5. Після підтвердження операції склади списують необхідну кількість товару зі своїх локальних сховищ.

6. У випадку успішного виконання система формує підтвердження заявки та зберігає її у базі даних SQLite.

7. Користувач отримує оновлений статус заявки у веб-інтерфейсі.

Сценарій 4. Керування історією замовлень та видалення заявок

Актор: Користувач

Мета: Переглянути журнал покупок або видалити помилкові записи з історії. **Основний сценарій:**

1. Користувач переходить до розділу історії замовлень.

2. Система відображає список усіх раніше оформлених заявок поточного користувача, підтягуючи їхній актуальний статус та деталі з бази даних.

3. Користувач може обрати конкретне замовлення та натиснути кнопку його видалення.

4. Система видаляє обраний запис із бази даних, перевіряючи, що це замовлення належить саме цьому користувачу.

5. Користувач також може обрати повне очищення історії замовлень.

6. Система повністю видаляє всі записи про замовлення цього користувача з локальної бази даних.

7. Веб-інтерфейс оновлюється, відображаючи актуальний стан журналу замовлень.

Найбільш наочно сценарії реалізованого проекту відображає діаграма сценаріїв використання (Use Case Diagram) [24]

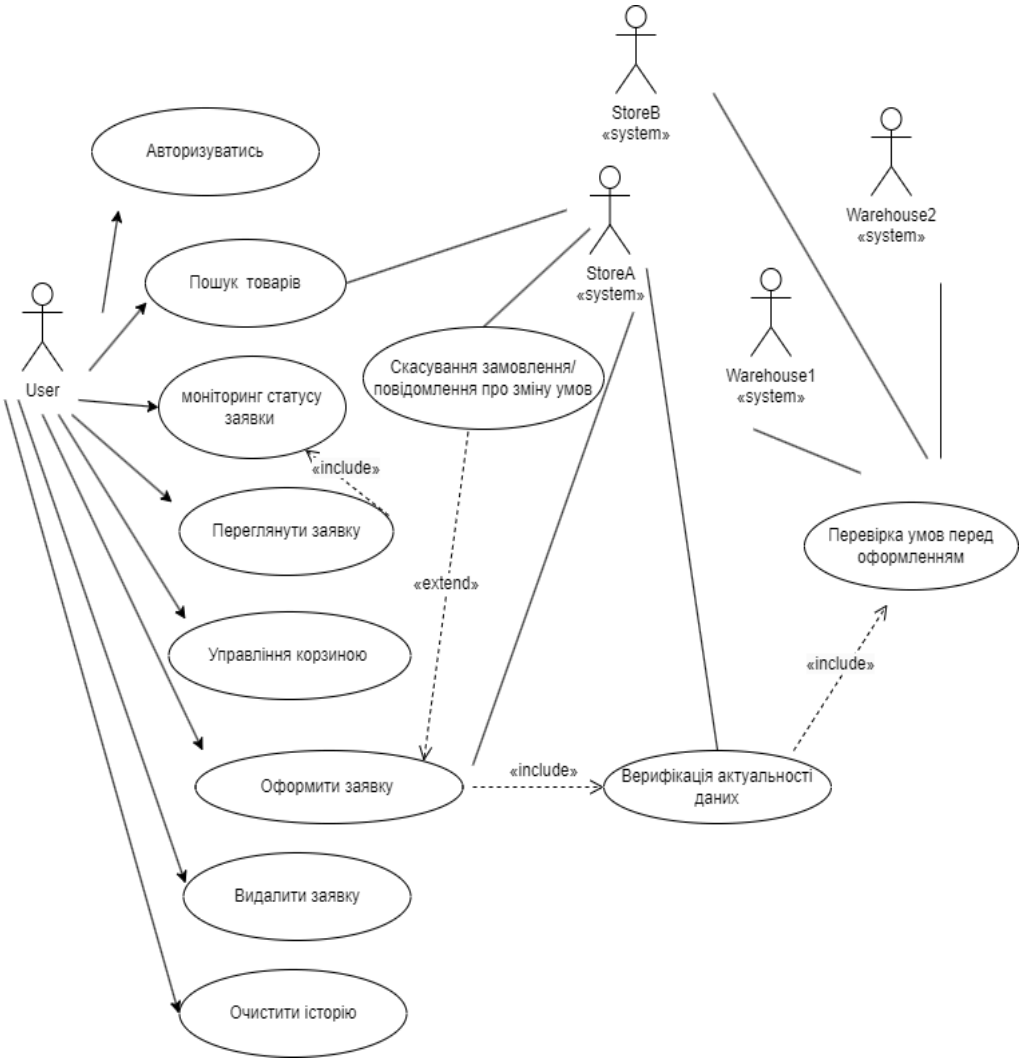


Рис 2.1- Діаграма сценаріїв використання (Use Case Diagram)

2.3 Обґрунтування вибору архітектура системи

Архітектура програмного забезпечення відіграє важливу роль у побудові структури програмного забезпечення, визначення організації її компонентів та взаємозвязок між ними.

З вище проведеного аналізу предметної області стає зрозумілим, що найбільш використовувана для розробки різноманітних веб-додатків та застосунків вважається клієнт-серверна архітектура. Вибір саме цієї моделі завдячує низькою переваг:

− чіткому розподіленню ролей між клієнтом та сервером. Сервер виконує всі ресурсномісткі операції та обробку даних в той же час клієнт залишається

максимально полегшеним та зосередженим на відображені інтерфейсу користувача;

− використання гнучких стандартизованих протоколів обміну даними (наприклад HTTP/REST), що дозволяє забезпечити єдиний механізм обміну між різними компонентами системи;

− можливість масштабування існуючої системи нарощуванням серверної частини або змінювання логіки обробки запитів без значної переробки клієнтської частини.

З огляду на можливий динамічний характер взаємодії учасників торговельної мережі , постійної зміни складу учасників взаємодії (у випадку розширення мережі або у випадку нестабільності їх роботи) та переорієнтування запитів між ними ці обставини накладають складність на реалізацію для клієнт-серверної архітектури.

Одна з причин складності такої реалізації полягає у парадигмі централізованих систем, які повинні контролювати кожен стан системи, а у випадку збільшення учасників взаємодії може виникнути перевантаження керуючого вузла. Також слід враховувати , що учасники можуть мати різноманітні ролі поведінки тому реалізація такої системи ускладняється як реалізацією самої поведінки учасника так і перевірки виконання необхідних умов.

Розглянуті задачі та повязані з ними обмеження найбільш повно вирішуються переходом з централізованої моделі до розподілених систем. У таких системах обчилювальні ресурси та логіка прийняття рішень не зосереджена на одномоу сервері, а рознесена між багатьма автономними вузлами, які взаємодіють через мережу.

Розподілені системи виокремлюють цілий класс систем, які мають наступні характеристики:

1) Прозора відмовостійкість.Такі системи продовжують функціонувати у випадку виходу з ладу окремих компонент.

2)Масштабованість. Можливість достатньо швидко розширувати такі системи у випадку долучення нових компонент не змінюючи значно центрального коду

3) Відкритість. Це здатність системи взаємодіяти з іншими компонентами системи використовуючи стандартні протоколи, незалежно від внутрішньої реалізації.

4) Паралелизм. Це можливість одночасного виконання багатьох процесів, що впливає на швидкість відповіді системи.

Як видно з розглянутих загальних характеристик децентралізовані системи найбільше підходять для вирішення перерахованих труднощів централізованих систем, якими є клієнт-серверні системи. Водночас загальні характеристики стосуються цілого класу розподілених систем але в контексті динамічної торгівлі найбільш повно ці характеристики можуть бути реалізовані за допомогою мультиагентних систем. Основними причинами є той факт, що агенти які утворюють МАС, характенизуютьсяавтономністю у прийнятті рішень. Агент як сутність пред прийняттям рішення може ініціювати переговори з іншими учасниками, керуватись своїми стратегіями для реалізації власних цілей. Це дає можливість побудувати систему зі складною комунікацію для вирішення складних завдань в динамічній обстановці, навіть з елементами невизначеності.

Доцільно розглянути деякі властивості агентів, що прямо впливають на ефективність такого підходу:

1) Аватономність передбачає дію агента без будь-якого зовнішнього впливу. Це пояснюється самовільної дії агента в залежності від власного внутрішного стану (наприклад, агент складу вирішує чи може він зарезервувати товар).

2) Соціальна поведінка передбачає здатність агента взаємодіяти з іншими агентами. Це досягається використанням спеціальної мови комунікації (FIPA-ACL), що дозволяє не просто обмінюватись повідомленнями а "домовлятися" розуміючи контекст та координувати власні дії

3) Реактивність передбачає можливість агента сприймати зміни у середовищі, в якому він працює (наприклад, виникнення нового замовлення) і своєчасно реагувати на них

4) Проактивність передбачає можливість агента мати ініціативу для досягнення поставленої цілі.

Таким чином, мультиагентна система відповідає найбільшою мірою характеру взаємодії учасників динамічного ринку, яким є торговельна мережа зі своїми прямими та сторонніми постачальниками.

Отже враховуючи сформовані функціональні та не функціональні вимоги до проєкту та динамічний характер дії учасників торговельної мережі можна зробити висновок, що архітенктура розроблюваної системи повинна задовільняти вимогам як централізованої так і децентрадізованої моделі. Це можливо тільки у випадку гібридної моделі.

У разі реалізації такої моделі взаємодію користувача з системою (один з магазинів, що виконує пошук та формування заявки на товар) покладається на клієнт-серверну модель тоді як складна взаємодія учасників ринку (торгівельної мережі) - на мультиагентну систему.

Звісно, реалізація взаємозвязку легкого кліента (браузера) з сервером доцілно реалізувати через HTTP методи запитів (наприклад GET, PUT та інші), що найкраще реалізовано в рамках REST взаємодії. Рівень агентів, враховуючи вибраний фреймворк SPADE, реалізує взаємодію між агентами через сервер XMPP. Важливо відмітити, що характер взаємодії на цих двох рівнях різний. Це пояснюється тим, шо HTTP - запити є безстановними, тобто кожен новий запит виконується окремо та не запамятовується сервером [25]. На протривагу цьому, взаємодія агентів за допомогою мови комунікації FIPA-ACL поверх протоколу XMPP дозволяє зберігати історію та контекст діалогу, що необхідно при реалізації багатостадійних переговорів між агентами в рамках одного діалогу (наприклад, багатокрокове узгодження ціни або залишків на різних складах).

Отже, реалізація гібридної системи буде потребувати узгодження рівнів веб-кліента та мультиагентноїсистеми, це стає передумовою створення проміжного рівня API, що буде транслювати запит користувача до мультагентної системи, та відповідь від мультагентної системи назад.

В наступному підпункті буде розглянуто більш детально реалізовану архітектуру проєкту.

2.4 Склад та призначення компонентів системи

Проведений аналіз архітектурних підходів показує, що спроєктована система має гібридну структуру, у якій поєднуються принципи класичної клієнт-серверної моделі та мультиагентного підходу. Подібне поєднання дозволяє відокремити рівень взаємодії з користувачем від складної логіки розподіленого пошуку та координації між вузлами системи. У результаті вдається зменшити навантаження на вебчастину застосунку та перенести обробку бізнес-процесів до агентного середовища.

Відповідно до визначеної архітектури систему поділено на декілька функціональних рівнів.

1. **Рівень представлення**

Цей рівень забезпечує безпосередню взаємодію з користувачем: приймає запити, відображає результати пошуку та забезпечує роботу інтерфейсу. Реалізація виконана у вигляді легкого клієнта на основі шаблонізатора Jinja2 та JavaScript.

До основних функцій рівня представлення належать:

− візуалізація даних через динамічне формування сторінок із результатами пошуку, станом кошика та статусами замовлень;

− організація асинхронної комунікації за допомогою функцій fetch, що дозволяє надсилати запити до API-сервера без повного перезавантаження сторінки;

− забезпечення інтерактивності користувацького інтерфейсу шляхом обробки відповідей від агентного шару, зокрема повідомлень про зміну вартості товару, резервування або його відсутність.

Фактично цей рівень виконує роль точки контакту між користувачем і всією розподіленою системою.

2. **Рівень шлюзування**

Рівень шлюзування виконує функцію проміжного середовища між вебінтерфейсом та агентною платформою. Основне його призначення полягає у стабільній передачі даних між HTTP-середовищем і внутрішньою мультиагентною системою.

У процесі проектування стало очевидно, що безпосереднє поєднання класичного вебсервера та агентного середовища може призводити до конфліктів асинхронних циклів виконання. Саме тому рівень шлюзування було реалізовано як дворівневу структуру:

1) API-сервер виступає зовнішнім шлюзом системи. Він приймає HTTP-запити від користувача, виконує первинну перевірку вхідних даних та ініціює передачу запитів до агентної платформи. Для взаємодії з наступним підрівнем використовується бібліотека requests, що дозволило ізолювати контексти виконання та уникнути конфліктів між асинхронною моделлю агентного середовища і вебфреймворком. Крім передачі запитів, API-сервер також відповідає за подальшу обробку результатів: запис резервів у базу даних та синхронізацію статусів замовлень відповідно до фінальної відповіді агентів.

2) Внутрішній агентний шлюз функціонує як окремий компонент системи, що працює на порту 8001. Його завданням є приймання HTTP-запитів від API-сервера та їх подальше перетворення у структуровані повідомлення внутрішнього протоколу SPADE. Саме цей компонент запускає сценарії пошуку товару або перевірки його наявності, передаючи агентам необхідні параметри - ідентифікатор товару, кількість, а також контекст поточного запиту.

Подібна організація шлюзового рівня забезпечує повне логічне відокремлення вебчастини системи від специфічного агентного середовища. Це суттєво спрощує підтримку та тестування окремих компонентів.

3. **Рівень агентної логіки**

Рівень агентної логіки формує інтелектуальне ядро системи. Саме тут зосереджена основна бізнес-логіка, пов’язана з розподіленим пошуком товарів, резервуванням ресурсів та координацією між учасниками торговельної мережі.

У межах цього рівня функціонують автономні агенти магазину, складів та зовнішнього постачальника. Для взаємодії між собою вони використовують повідомлення зі структурованими метаданими. Завдяки цьому кожен агент здатний визначати контекст поточної взаємодії — запит, команду або інформаційне повідомлення - а також відстежувати окремі сценарії обробки за допомогою thread_id.

Окремо варто зазначити, що кожен агент має власне локальне сховище даних. Такий підхід відповідає принципам розподіленої архітектури та дозволяє забезпечити автономність вузлів системи навіть у випадку тимчасової недоступності інших компонентів мережі.

Реалізоване архітектурне рішення з врахуванням розгляданих особливостей будови більш наглядно демонструє загальна схема на рис 2.1

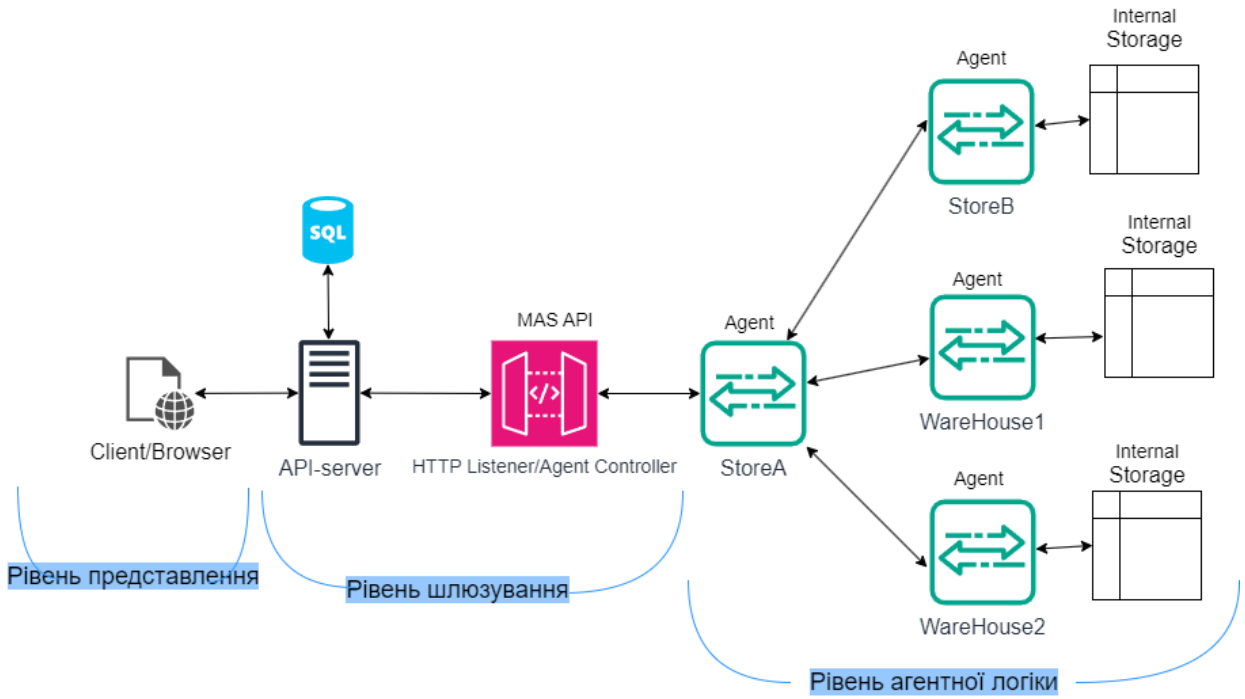


Рис 2.2 - Загальнаархітектурна схема веб-інформаційної системи

Практична реалізація зазначеної архітектури системи потребувала організації окремого запуску та взаємодії між її компонентами. Найбільш наочно ці аспекти відображає діаграма розгортання. На ній відображено основні програмні модулі системи, способи їх запуску та взаємозв’язок між ними.

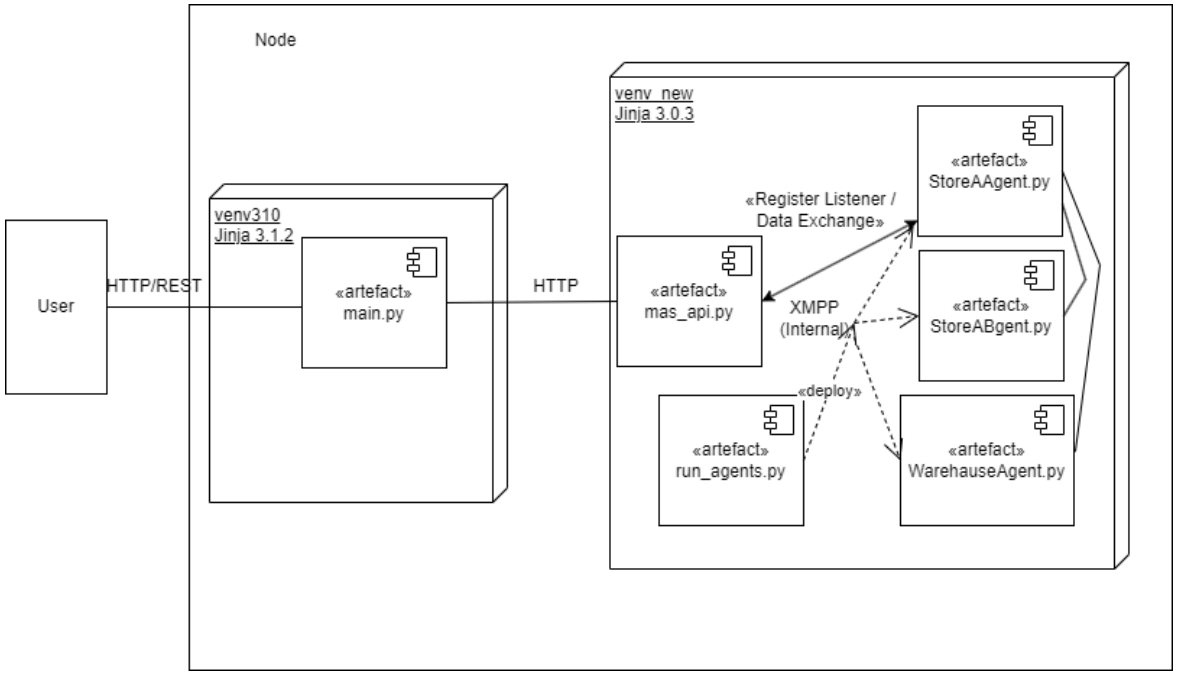


Рис 2.3 - Діаграма розгортання

Як видно з діаграми, окремі компоненти системи запускаються в різних віртуальних середовищах Python. Це дозволило розділити веб-рівень API-серверу відмультиагентного середовища SPADE. Передувало цьому також наявність конфліктів між версіями бібліотек Jinja2для API-серверу (реалізованому на FastAPI) та агентної платформи SPADE.

2.5 Обґрунтування та опис алгоритму взаємодії компонентів

Для забезпечення коректного функціонування розподіленої системи та координації дій між веб-інтерфейсом і автономними агентами було розроблено наскрізний алгоритм взаємодії компонентів системи.Особливістю даного рішення стало розділення логіки на два окремі сценарії роботи: децентралізований пошук пропозицій та двоетапна процедура оформлення заявки з перевіркою актуальності даних.

Динаміку роботи системи та інформаційні потоки між розглянутими раніше рівнями можна умовно поділити на три основні етапи.

1. Ініціалізація запиту та трансляція (Рівень представлення → Рівень шлюзування)

Процес взаємодії із системою починається на рівні представлення коли користувач вводить назву необхідного товару у форму пошуку вебінтерфейсу.Після цього клієнтський JavaScript-скрипт перехоплює подію, формує асинхронний HTTP-запит та через функцію fetch() передає параметри пошуку до відповідного маршруту API-сервера, реалізованого на FastAPI.

На стороні API-сервера виконується первинна перевірка отриманих даних. Далі використовуючи бібліотеку requests сервер формує GET-запит, а у випадку оформлення замовлення — POST-запит до внутрішнього агентного шлюзу (MAS API, порт 8001).Такий підхід дозволяє ізолювати вебконтекст користувача від асинхронних циклів виконання агентного середовища.

2. Агентний пошук та багаторівневе опитування (Рівень шлюзування → Рівень агентної логіки)

Після надходження HTTP-запиту внутрішній агентний шлюз (MAS API) виконує його обробку, витягує назву товару та формує внутрішню подію для платформи SPADE.Далі керування передається агенту-координатору Магазину А (StoreAAgent), який організовує подальший процес пошуку.

Логіка роботи агента побудована за каскадним принципом.

Спочатку виконується опитування внутрішніх складів. Через XMPP-протокол агент надсилає запити локальним агентам складів (WarehouseAgent 1 та WarehouseAgent 2).Кожен із них перевіряє власне локальне сховище та повертає відповідь з інформацією про наявність товару, доступну кількість і поточну ціну.

Якщо ж товар на внутрішніх складах не знайдено запускається другий етап пошуку. Тоді StoreAAgent переходить до взаємодії з агентом Магазину Б, який виступає зовнішнім резервним постачальником.

Магазин Б працює як окремий автономний вузол зі своєю внутрішньою логікою прийняття рішень.Він може підтвердити запит або повернути відмову (Refuse) залежно від поточного стану своїх ресурсів, внутрішніх правил чи інших умов. Через це система повинна коректно реагувати на ситуації коли зовнішній партнер не погоджується виконувати угоду.

Окремо варто розглянути процес оформлення замовлення (Checkout).У цьому випадку використовується дворівневий механізм синхронізації та уникнення конфліктів.

На першому етапі формується замовлення на основі результатів первинного пошуку.Під час пошуку StoreAAgent повертає інформацію у форматі «кількість товару за найкращою ціною / загальна кількість на всіх складах». Разом із результатом фактично фіксується ідентифікатор постачальника, який надав найкращу пропозицію.

Спираючись на отримані дані користувач визначає необхідну кількість товару, додає її до кошика та ініціює оформлення заявки.

Після цього запускається етап повторної перевірки та трансляції даних.Клієнтський запит, який містить назву товару, необхідну кількість та інформацію про вибраного постачальника, через FastAPI-сервер передається до внутрішнього агентного шлюзу, а потім - агенту StoreA.

Оскільки за час формування кошика ситуація на складах може змінитися, StoreAAgent виконує повторну перевірку у конкретного агента складу. При цьому уточнюється актуальна наявність товару та перевіряється відповідність ціни тій, яка була отримана під час попереднього пошуку.

У ході аналізу було встановлено що саме цей етап є одним із найбільш критичних для запобігання конфліктам між одночасними замовленнями різних користувачів.

Якщо виявляється що кількість товару або його вартість змінилися, система не скасовує процес оформлення автоматично.У такому випадку StoreAAgent формує зворотне повідомлення для вебінтерфейсу, яке викликає появу діалогового вікна confirm.

Користувач отримує інформацію про нові умови замовлення і може погодитися з ними або відмовитися від продовження операції.Якщо підтвердження отримано відповідь знову передається агенту StoreA для продовження виконання сценарію.

Фінальний етап передбачає безпосереднє списання товару.Якщо дані залишилися актуальними або користувач підтвердив оновлені умови через вікно confirm, агент StoreA формує остаточний запит на списання товару до відповідного агента складу.

Після отримання такої команди складський агент оновлює власне локальне сховище та виконує списання необхідної кількості товару.Все відбувається вже на рівні агентного середовища.

Паралельно StoreAAgent надсилає результат виконання на FastAPI-сервер.Далі сервер фіксує завершення взаємодії між агентами, виконує запис транзакції до бази даних db.sqlite та оновлює статус заявки для користувача.

Додаткова перевірка перед фінальним оформленням замовлення дозволяє суттєво знизити ризик виникнення конфліктів у випадку одночасних звернень до одного й того самого товару.Завдяки цьому вдається уникнути ситуацій коли продукція вже була зарезервована або придбана іншим користувачем, але інформація про це ще не встигла потрапити до клієнта.

Як вже зазначалося раніше, як під час первинного пошуку так і під час верифікації замовлення агент StoreA сортує отримані відповіді за критерієм ціни – від нижчої до вищої.Саме на основі цього формується подальша логіка підбору товару.

а) якщо потрібна кількість товару повністю покривається пропозицією з найкращою ціною, то заявка буде сформована виключно на основі цього постачальника.

б) якщо необхідна кількість перевищує доступний обсяг товару за найвигіднішою ціною, формується збірна заявка. У такому випадку система

спочатку використовує найбільш вигідну пропозицію, а решту кількості добирає з інших доступних джерел із вищою, але все ще прийнятною ціною.

3. Сервісні сценарії керування заявками (взаємодія з базою даних та журналами подій)

Окрім основних бізнес-процесів пов'язаних із пошуком товарів та оформленням замовлень, у системі передбачено набір сервісних операцій для роботи із заявками в особистому кабінеті користувача.Вони відображають подальший життєвий цикл заявки вже після завершення взаємодії з агентною мережею.

Моніторинг та перегляд стану заявки

Користувач у будь-який момент може переглянути поточний стан створеної заявки.У цьому випадку система не виконує звернення до XMPP-мережі агентів, а отримує необхідні дані безпосередньо з локальної бази даних db.sqlite. Завдяки цьому зменшується навантаження на агентне середовище.І одночасно пришвидшується отримання результату для користувача. Під час такого запиту можуть відображатися різні статуси заявки: success, pending, alert або інші значення які були сформовані під час її обробки.

Видалення заявки

Якщо користувач вирішив відмовитися від замовлення або отримав повідомлення про зміну умов та не бажає продовжувати оформлення, він може ініціювати видалення заявки. Після надходження такого запиту відповідний маршрут FastAPI виконує обробку операції та видаляє запис про транзакцію з локальної бази даних.Сама процедура є відносно простою. За необхідності додатково може формуватися службове повідомлення для шлюзу з метою логування події або коригування подальших дій системи.При цьому запуск повного циклу агентних переговорів не потрібний.

Очищення історії

Окремо реалізовано можливість очищення історії заявок.Користувач може масово приховувати завершені записи або повністю видаляти закриті транзакції з локальної бази даних системи.

Таким чином завершується повний цикл керування заявками у межах розробленого вебдодатку — від створення та обробки замовлення до його подальшого перегляду, видалення чи архівування.

2.6 Розробка протоколу обміну даними та структур повідомлень

Розглянуті раніше структура системи та алгоритми взаємодії її компонентів передбачають необхідність організації стабільного зв’язку між усіма логічними рівнями архітектури. Формування механізмів комунікації здійснювалося з урахуванням поділу системи на окремі функціональні шари, кожен із яких виконує власну роль у процесі обробки запитів та координації дій між компонентами.

У результаті в системі було виокремлено два основні типи комунікаційної взаємодії.

Як зазначалося вище, спроектована веб-інформаційна система управління заявками реалізує два ізольованих процеси : децентралізованого пошуку прорпозицій та двоетапну процедуру оформлення заявки з перевіркою актуальності даних. Тому нижче розглянемо більш детально послідовність дій (динаміки взаємодії) окремо для кожного з цих процесів та структури даних, що використовуються для організації звязку між логічними архітектурними рівнями.

2.6.1. Динаміка взаємодії та структури даних у процесі пошуку товарів

Перед початком пошуку товарів у системі передбачена попередня авторизація клієнта через стандартне введення логіна та пароля. Це потрібно для входу до приватної частини системи пошуку де вже доступний функціонал роботи із товарами та заявками.Сам механізм авторизації реалізований у спрощеному вигляді, оскільки основний акцент у проєкті робився саме на взаємодії агентів та процесах розподіленого пошуку.

Для користувача створено досить простий веб-інтерфейс. Його основна задача - показати процес пошуку товарів, формування кошика та подальшого

оформлення заявки. Через це частина елементів інтерфейсу реалізована без надлишкової складності.

Сам процес децентралізованого пошуку з точки зору користувача виглядає синхронним, тобто після натискання кнопки пошуку він просто очікує результат. Але всередині системи ситуація інша. На рівні логічних шарів та агентного середовища виконуються асинхронні і розподілені процеси обміну повідомленнями між окремими агентами. Частина запитів може виконуватись паралельно.

Для кращого відображення хронології взаємодії між компонентами системи та визначення меж відповідальності окремих файлів і модулів була розроблена діаграма послідовності (Sequence Diagram) (рис.А.1 додаток А) . Саме вона дозволяє більш наочно показати порядок передачі повідомлень між веб-клієнтом, API-рівнем та агентним середовищем.

Опис діаграми послідовностівіпроцесу пошуку товару

Авторизований користувач на головній сторінці веб-інтерфейсу Index.html вводить назву товару у пошуковий рядок та натискає кнопку «Знайти».Після цього спрацьовує JavaScript-функція яка формує HTTP-запит до API-сервера. Для користувача цей процес виглядає досить простим і майже миттєвим, хоча всередині системи запускається вже цілий ланцюг взаємодій.

API-сервер main.py виступає початковою точкою маршруту пошуку.Саме він приймає HTTP-запит від клієнта та перенаправляє його на порт агентного шлюзу. Далі робота переходить вже до внутрішнього агентного середовища.

Агентний шлюз, частина якого реалізована як асинхронний веб-сервер, приймає REST-запит і перетворює його у внутрішній об’єкт повідомлення. При цьому до повідомлення додається унікальний ідентифікатор сесії `thread`, який потім використовується для відстеження конкретного сценарію взаємодії. Після цього повідомлення передається у асинхронну XMPP-мережу агентів.

Запит надходить агенту-координатору. Саме він відповідає за подальший розподіл повідомлень між іншими агентами системи. Спочатку координатор розсилає запити агентам складів. Якщо ж потрібний товар там відсутній —

виконується звернення до партнерського магазину B або зовнішнього постачальника.

Після отримання відповідей від агентів складів та постачальників координатор аналізує отримані пропозиції і визначає найбільш вигідний варіант. Далі формується відповідь, до якої знову додається унікальний ідентифікатор thread, після чого повідомлення надсилається назад до агентного шлюзу mas_api.py.

Агентний шлюз приймає відповідь через підписаного слухача повідомлень і використовуючи ідентифікатор сесії визначає до якого саме HTTP-запиту належить отриманий результат.Після цього формується відповідь для API-сервера.

API-сервер після отримання даних виконує їх обробку та повертає готовий результат фронтенду користувача.У підсумку користувач бачить вже сформований список знайдених товарів або повідомлення про їх відсутність.

Нижче наведено більш детальний покроковий розгляд структури повідомлень та даних які використовуються під час виконання пошуку товарів.

Крок1. Клієнтський HTTP-запит (Frontend → → API-сервер)

Користувач у пошуковому полі <inputid="search"> вводить назву потрібного товару, наприклад milk та натискає кнопку “Знайти”. Після цього спрацьовує асинхронна JavaScript-функція search(), яка зчитує введене значення та формує HTTP-запит до бекенда, тобто API-сервера.

На цьому етапі логіка ще досить проста - фронтенд лише передає введений користувачем рядок далі у систему. Дані передаються у вигляді Query-параметра прямо в URL запиту.Такий підхід був обраний через простоту реалізації та зручність тестування через браузер або Postman.

Транспортний формат запиту використовується у вигляді HTTP GET-запиту.

```
GET /search?query=milk HTTP/1.1
```

```
Host: localhost:8000
Accept: application/json
```

Крок 2 . Міжсервісний REST-запит (API-сервер → → агентний шлюз на aiohttp)

Бекенд сервер на FastAPI(файл main.py, порт 8000) приймає вхідний HTTP-запит який надійшов від фронтенду. Після цього управління передається сервісному шару MASSearchService , який по суті виконує роль внутрішнього маршрутизатора між веб-рівнем та агентною частиною системи.

Далі за допомогою бібліотеки requests формується вже новий HTTP GET-запит на порт 8001, де працює окремий шлюз мультиагентної системи.На відміну від фронтенду тут взаємодія вже відбувається всередині самої системи, без прямої участі користувача.

Цікаво що навіть попри використання асинхронного середовища SPADE та aiohttp, на цьому етапі було вирішено залишити саме синхронний виклик через requests. Це дозволило уникнути конфліктів між різними event loop і трохи спростило логіку обміну повідомленнями між компонентами.

На даному етапі дані все ще передаються у стандартному для веб-середовища форматі JSON-параметрів, але вже фактично ізольовані від клієнтської частини внутрішньою мережею системи.Фронтенд більше напряму не взаємодіє з агентним середовищем.

Транспортний формат внутрішнього запиту реалізований через HTTP REST JSON.

```
GET /search?query=milk HTTP/1.1
Host: localhost:8001
Accept: application/json
```

Крок 3 Передача запиту в агентське середовище (агентний шлюз на aiohttp → → MAS)

Внутрішній мікросервіс mas_api.py, який працює на порту 8001 та реалізований на базі aiohttp, приймає HTTP-запит у функцію handle_search(). Саме на цьому етапі відбувається одна з ключових трансформацій у всій архітектурі системи.

Якщо раніше використовувались звичайні веб-абстракції типу HTTP та JSON, то тепер запит переходить уже у формат взаємодії мультиагентного середовища.Відбувається конвертація даних у стандарти FIPA-ACL, які використовуються агентами SPADE для внутрішнього обміну повідомленнями.

Для цього вбудований шлюзовий агент ClientAgent інкапсулює отримане строкове значення пошуку у спеціальний контейнер повідомлення бібліотеки SPADE.По суті звичайний рядок із назвою товару перетворюється у повноцінне агентне повідомлення з набором метаданих.

Далі повідомлення доповнюється службовою інформацією: адресою отримувача, типом взаємодії, ідентифікатором потоку thread та іншими параметрами які потрібні для подальшого маршрутизаційного процесу в XMPP-середовищі.

Структура повідомлення всередині агентної мережі (XMPP / FIPA-ACL):

```
{
  "performative": "query",
  "sender": "client@localhost",
  "receiver": "storea@localhost",
  "thread": "d3b07384-d113-483a-a5cf-81ff076f8e21",
  "body": "milk"
}
```

Примітка:

Додавання нового поля "performative": "query" пов’язане з тим, що агенти працюють у власному ізольованому середовищі і тому повинні правильно розуміти контекст отриманого повідомлення.Значення query одразу сигналізує агенту-отримувачу (StoreAAgent), що перед ним саме інформаційний запит а не команда резервування чи списання товару.

Тобто агент після отримання такого повідомлення розуміє що потрібно просто виконати пошук у локальному каталозі та повернути результат.Жодних транзакцій або змін стану товару при цьому не запускається. Це важливо, оскільки один і той самий агент у системі може обробляти різні типи команд.

Окремо варто звернути увагу на поле "thread": "UUID".Саме воно використовується для ідентифікації конкретної сесії взаємодії між компонентами системи.

Проблема полягає в тому що агентне середовище працює асинхронно. Після відправлення повідомлення шлюз mas_api.py не блокується і не очікує відповідь у тому самому потоці виконання.Він продовжує обробляти інші запити та слухати нові повідомлення від агентів.

Через це при надходженні відповіді від агентів складів або постачальників необхідно точно визначити до якого саме клієнтського HTTP-запиту належить отриманий результат. Для цього і використовується унікальний UUID-ідентифікатор thread. Асинхронний слухач шлюзу зчитує це поле та співставляє відповідь із потрібною сесією пошуку.

Крок 4 внутрішня взаємодія агентів (StoreAAgent ↔ WarehouseAgent / StoreBAgent)

Після отримання повідомлення з перформативом query головний агент-координатор StoreAAgent (файл StoreAAgent.py) запускає внутрішній процес збору інформації про товар.На цьому етапі система вже повністю переходить у режим внутрішньої агентної взаємодії.

Оскільки архітектура побудована за децентралізованим принципом, StoreAAgent не звертається до якоїсь єдиної централізованої бази даних. Замість цього він починає опитувати інші незалежні агентні сутності які мають власні локальні сховища даних.

Спочатку надсилаються запити агентам локальних складів WarehouseAgent. Якщо ж потрібний товар там не знайдено або кількість недостатня - додатково виконується звернення до агента партнерської мережі StoreBAgent. Через це пошук може проходити у декілька етапів.

Для організації такої взаємодії всередині XMPP-середовища формується внутрішній каскад повідомлень.Кожен агент отримує окремий запит зі своїм контекстом взаємодії та ідентифікатором `thread`, щоб потім результати можна було правильно співставити між собою.

Структура внутрішнього запиту від головного агента до агента складу:

```
{
  "performative": "query",
```

```
  "sender": "storea@localhost",
  "receiver": "w1@localhost",
  "thread": "d3b07384-d113-483a-a5cf-81ff076f8e21",
  "body": "milk"
}
```

Агенти складів або партнерського магазину після отримання повідомлення виконують перевірку власних локальних сховищ даних (DATA_STORAGE / my_data) та формують відповідь головному агенту StoreAAgent. На цьому етапі тип взаємодії вже змінюється.

Перформатив query, який використовувався для запиту інформації, замінюється на inform. Це відповідає логіці стандарту FIPA-ACL, де inform означає передачу знайдених даних або результату виконання операції.

Структура повідомлення-відповіді від агента складу або магазину B має такий вигляд:

```
{
  "performative": "inform",
  "sender": "w1@localhost",
  "receiver": "storea@localhost",
  "thread": "d3b07384-d113-483a-a5cf-81ff076f8e21",
  "body": {
    "qty": 90,
    "price": 45.00
  }
}
```

Варто звернути увагу що поле "thread" не змінюється протягом усього сценарію взаємодії. Саме це дозволяє пов’язати всі повідомлення між собою навіть якщо вони надходять у різний час або від різних агентів.

Особливо важливим це є для шлюзу mas_api.py, оскільки він працює асинхронно і може одночасно очікувати відповіді на велику кількість подібних запитів. Відповідно без thread було б складно зрозуміти до якого саме HTTP-запиту належить конкретна відповідь від агента складу.

Після отримання всіх відповідей StoreAAgent виконує збір результатів, аналізує отримані ціни та обирає найкращу пропозицію (best_price).Далі формується фінальна відповідь яка вже повертається назад до шлюзу мультиагентної системи.

Крок 5 Формування та відправка відповіді користувачу (MAS → → агентний шлюз на aiohttp → → Frontend)

Після завершення пошуку агент StoreAAgent формує фінальну відповідь для шлюзу.Всередину повідомлення він повторно додає ідентифікатор сесії thread, а також змінює перформатив на inform, оскільки тепер уже передається готовий результат а не запит на пошук. Після цього повідомлення відправляється назад до шлюзу mas_api.py.

На стороні mas_api.py працює спеціальний слухач який підписаний на агентну XMPP-мережу.Він постійно очікує нові повідомлення від агентів та після отримання відповіді зчитує поле thread. Саме через цей ідентифікатор шлюз визначає до якого саме клієнтського HTTP-запиту належить отриманий результат.

Далі дані конвертуються у JSON-формат та повертаються HTTP-відповіддю назад на порт 8000, де працює FastAPI-сервер.На цьому етапі агентне середовище вже фактично завершує свою роботу.

API-сервер прослуховуючи власний порт отримує JSON-відповідь, виконує остаточну обробку даних і надсилає результат користувачу на фронтенд.По суті для браузера це виглядає як звичайна HTTP-відповідь хоча всередині системи перед цим відбулося декілька асинхронних етапів обміну між агентами.

Фінальний JSON-формат відповіді який повертається у браузер:

```
{
  "results": [
    {
      "name": "milk",
      "status": "found",
      "source": "warehouse",
      "display_qty": "90 / 250",
      "price": 45.00
    }
  ]
}
```

Фронтенд після отримання відповіді від сервера на сторінці Index.html через функцію search() виконує await response.json() та розпаковує отримані

дані.Після цього відповідний JavaScript-код формує елементи інтерфейсу і відображає користувачу знайдені товари або іншу службову інформацію.

У випадку якщо товар повністю відсутній на складах, у постачальників або партнерського магазину, чи користувач просто неправильно ввів назву — система повертає повідомлення "Товар не знайдений".На рівні агентної логіки при цьому StoreAAgent формує статус "not_found" або повертає порожній масив результатів.

Далі вже спрацьовує відповідний сценарій на стороні інтерфейсу користувача.Фронтенд перевіряє отриману відповідь і показує повідомлення про відсутність товару без перезавантаження сторінки. Це виглядає простіше для користувача і не потребує повторного відкриття сторінки.

2.6.2. Динаміка взаємодії та структури даних у процесі оформлення заявок

Процес оформлення заявки в системі пов’язаний не лише з простим записом товару у базу даних. Перед цим виконується повторна перевірка актуальності вибраної пропозиції клієнта, оскільки після етапу пошуку ціна або кількість товару могла вже змінитися.

Після підтвердження актуальності даних система виконує запис заявки у локальну базу даних, яка використовується для подальшого зберігання інформації про замовлення та його статус. Одночасно з цим запускається процедура узгодженого списання або резервування товару у відповідного постачальника.

Динаміка взаємодії між компонентами системи, порядок обміну JSON-повідомленнями а також життєвий цикл зміни статусів заявки були відображені на окремо розробленій діаграмі послідовності процесу оформлення заявки.

Для кращого відображення хронології взаємодії між компонентами системи та визначення меж відповідальності окремих файлів і модулів була розроблена діаграма послідовності (Sequence Diagram) (рис.А.2 додаток А) .

Саме вона дозволяє більш наочно показати порядок передачі повідомлень між веб-клієнтом, API-рівнем та агентним середовищем.

Опис діаграми послідовності для процесу оформлення заявки

Взаємодія між веб-інтерфейсом користувача, локальною базою даних SQLite та багатоагентною системою під час оформлення замовлення реалізована за двоетапною схемою.Така організація процесу дозволяє зменшити ризик конфліктів у випадку зміни ціни або кількості товару на віддалених складах та забезпечити цілісність даних під час виконання операцій.

Етап 1. Опосередкована верифікація параметрів замовлення

Процес розпочинається на стороні веб-інтерфейсу, коли користувач натискає кнопку оформлення замовлення.Після цього клієнтський сценарій формує асинхронний HTTP POST-запит до ендпоінта /checkout основного API-сервера FastAPI, який працює на порту 8000.

На стороні FastAPI запускається цикл перевірки для кожної позиції із кошика користувача.Для цього викликається метод verify_item сервісу MASSearchService, який надсилає HTTP POST-запит /verify до шлюзу інтеграції агентів (aiohttp, порт 8001).

У даному сценарії агентний шлюз виконує роль проміжної ланки між вебчастиною та агентною мережею.Компонент ClientAgent генерує унікальний ідентифікатор сесії thread_id за допомогою UUID4 та формує XMPP-повідомлення типу inform з метаданими request_type="verify".

Після цього повідомлення передається координатору мережі - агенту StoreAAgent.

Отримавши запит StoreAAgent запускає внутрішню логіку get_sorted_offers.Спочатку виконується опитування локальних агентів складів (WarehouseAgent) за допомогою перформативу query.

Якщо потрібний товар на локальних складах відсутній відбувається каскадне переключення на резервний канал.Тоді формується додатковий запит до партнерського агента StoreBAgent.

Після отримання всіх відповідей (XMPP-повідомлення типу inform) координатор виконує сортування знайдених пропозицій за критерієм мінімальної ціни та формує підсумковий результат.Далі відповідь повертається назад на агентний шлюз.

На стороні шлюзу фоновий компонент ResponseListener перехоплює повідомлення, знаходить відповідний запит за значенням thread_id та передає дані HTTP-серверу для формування JSON-відповіді.

Якщо під час перевірки виявляється що поточна ціна товару стала вищою за ту яку користувач бачив під час додавання до кошика, FastAPI призупиняє подальше оформлення заявки.У такому випадку оновлюються дані сесії та фронтенду повертається статус "alert" (PRICE_CHANGED).

Після цього користувачеві відображається модальне вікно з інформацією про нові умови замовлення та пропозицією підтвердити або скасувати подальше виконання операції.

Етап 2. Каскадне децентралізоване списання та транзакційна фіксація

Після отримання підтвердження від користувача, або у випадку якщо ціна взагалі не змінювалася, запускається фінальна стадія оформлення покупки.Фронтенд повторно ініціює відповідний запит до сервера.

На стороні FastAPI витримується невелика пауза приблизно у 2 секунди для стабілізації потоків після попередньої перевірки.Далі викликається метод checkout_item, який надсилає HTTP POST-запит /checkout на шлюз (порт 8001).

Отримавши запит шлюз генерує новий thread_id та передає координатору StoreAAgent команду на фактичне списання товару з використанням request_type="sell".

Після цього агент StoreAAgent запускає асинхронний метод handle_sell.У межах цього методу послідовно обробляється список доступних постачальників (offers), який був сформований під час попереднього етапу пошуку та верифікації.

Для кожного джерела агент надсилає окремий XMPP-запит з перформативом request, який містить інструкцію виду {"action":"reduce","qty":take_qty}.Таким чином кожен агент отримує команду на списання лише тієї кількості товару яка потрібна саме від нього.

Локальні агенти складів (WarehouseAgent) або партнерський магазин StoreBAgent приймають такий запит, перевіряють власні залишки у локальних структурах даних (DATA_STORAGE або STORE_B_DATA) і виконують фактичне зменшення кількості товару.

Варто зазначити що в агенті партнерського магазину реалізовано ймовірнісний механізм відмови (rejection_chance = 0.20).Це було зроблено для моделювання реальних умов роботи, наприклад проблем зв'язку, зміни внутрішніх правил постачальника або інших непередбачуваних ситуацій.

У випадку відмови або недостатньої кількості товару агент повертає відповідь зі статусом rejected.

Усі отримані відповіді повертаються до StoreAAgent через повідомлення типу inform.Після цього координатор збирає результати по кожній окремій мікроугоді та формує підсумковий звіт.

Далі шлюзу повертається фінальний статус "processed", який також містить прапорець final_success.Його значення дорівнює True, якщо замовлення вдалося повністю зібрати з доступних джерел.

Агентний шлюз передає отримані результати назад на FastAPI-сервер.Після цього вебсервер переходить до етапу збереження даних та відкриває блок безпечної обробки винятків try/except.

Якщо хоча б частина товарів була успішно придбана (is_anything_bought = True), виконується запис усіх результатів у базу даних db.sqlite.При цьому зберігаються як успішні операції, так і відхилені із зазначенням конкретного складу або постачальника.

Для цього використовується серія запитів INSERT INTO orders.

Після успішного завершення операцій викликається conn.commit(), кошик користувача очищується а фронтенд отримує команду на відображення результату оформлення замовлення.

Якщо ж під час роботи з базою даних або мережею виникла критична помилка сервер переходить до блоку except.У такому випадку виконується виклик conn.rollback(), який повністю скасовує всі незбережені зміни та дозволяє уникнути порушення узгодженості даних у системі.

Етап перевірки перед оформленням заявки

На етапі попередньої перевірки агентний шлюз надсилає повідомлення агенту StoreAAgent для повторної перевірки відповідності ціни та доступної кількості товару результатам первинного пошуку. По суті система ще раз перевіряє чи не змінились умови поки користувач формував кошик.

Для цього у body повідомлення передаються параметри які користувач вибрав у кошику.Наприклад назва товару, ціна яка була показана користувачу та потрібна кількість товару.

```
{
  "performative": "inform",
  "request_type": "verify",
  "sender": "client@localhost",
  "receiver": "storea@localhost",
  "thread": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "body": {
    "product": "milk",
    "client_price": 45.00,
    "client_qty": 1
  }
}
```

Етап фінального оформленням заявки

Після успішної перевірки або після підтвердження користувачем нових умов при status="alert" агентний шлюз через роут handle_checkout() формує вже фінальне повідомлення з типом операції sell .Саме це повідомлення запускає процедуру фактичного списання товару.

Тобто для StoreAAgent це вже не просто перевірка а команда на виконання операції reduce для відповідного складу або постачальника.

```
{
  "performative": "inform",
```

```
    "request_type": "sell",
    "sender": "client@localhost",
    "receiver": "storea@localhost",
    "thread": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
    "body": {
      "product": "milk",
      "client_price": 45.00,
      "client_qty": 1
    }
  }
```

Етап списання у складів/ надання пропозиції партнеру StoreBAgent

Після визначення найбільш вигідного постачальника координатор StoreAAgent формує та надсилає цільову команду на списання товару.Для спрощення реалізації та уніфікації механізму взаємодії використовується однаковий перформатив request як для локальних агентів складів (W1, W2), так і для зовнішнього партнера StoreBAgent.

Таким чином незалежно від типу постачальника подальша логіка обробки повідомлень залишається практично однаковою.Це дозволило уникнути дублювання окремих сценаріїв у коді.

При зверненні до StoreBAgent додатково враховується логіка автономного агента.На відміну від локальних складів він може повернути статус rejected не лише через відсутність потрібної кількості товару, а й через власні внутрішні обмеження або змодельовану ймовірність відмови.

У поточній реалізації така ймовірність становить 20 %.Це використовується для імітації реальних умов роботи розподіленого середовища, де зовнішній партнер не завжди може підтвердити виконання операції.

Приклад повідомлення для агента складу або партнерського магазину на зменшення залишків товару:

```
  {
    "performative": "request",
    "sender": "storea@localhost",
    "receiver": "w1@localhost",
    "thread": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
    "body": {
      "action": "reduce",
      "product": "milk",
      "qty": 1
    }
```

```
    }
```

2.7 Висновки до другого розділу

Проаналізувавши вимоги до автоматизації керування заявками торгівельної мережі, було прийнято рішення розробити веб-інформаційну систему на основі гібридної архітектури. Створення такої системи дозволило поєднати зручний для користувача веб-інтерфейс із децентралізованим мультиагентним середовищем SPADE, де кожен склад або магазин взаємодіє як автономний вузол.

У ході проєктування було деталізовано базові сценарії використання системи, включаючи процеси пошуку товарів, двоетапної верифікації даних перед замовленням та оформлення заявок. Виходячи з цього, було обґрунтовано розподіл шлюзового рівня на зовнішній API-сервер (FastAPI) для прийому HTTP-запитів та внутрішній агентний шлюз (MAS API) для їх трансляції в асинхронні повідомлення агентів. Така дволанкова структура забезпечує стабільну роботу системи без конфліктів асинхронних потоків і мінімізує ризики помилок при одночасному оформленні замовлень, що формує надійну базу для програмної реалізації проєкту.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ВЕБ-ІНФОРМАЦІЙНОЇ СИСТЕМИ

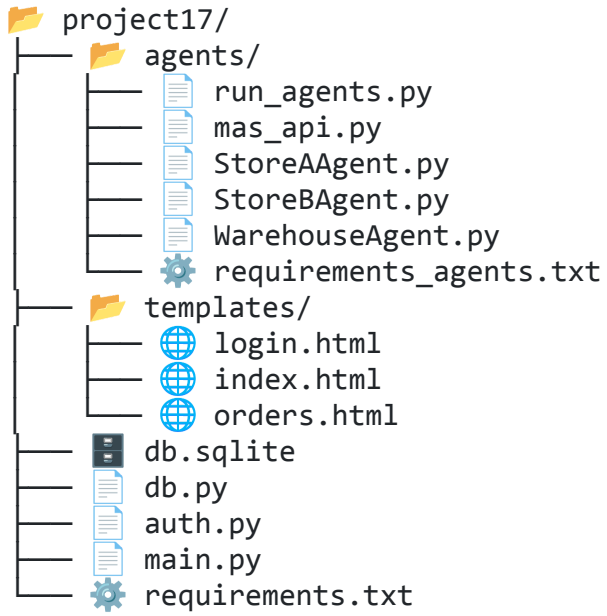
3.1 Структура та склад проекту

Перед переходом до безпосереднього опису реалізації окремих компонентів системи доцільно спочатку розглянути загальну структуру проєкту та спосіб розподілу файлів за їх функціональним призначенням. Це дозволяє краще зрозуміти як саме організована взаємодія між різними частинами системи.

Програмний комплекс побудований за модульним принципом. Через це логіка веб-рівня, комунікаційного шлюзу та децентралізованої мережі агентів

ізольована одна від одної і може розвиватись незалежно. Такий підхід також спрощує підтримку коду та внесення змін у окремі частини системи без значного впливу на інші модулі.

Основне дерево каталогів та файлів проекту має наступний вигляд:



Як можна побачити з наведеної структури проєкт поділений на два окремих функціональних шари які взаємодіють між собою через шлюзовий механізм.Таке розділення дозволяє відокремити класичну веб-логіку від мультиагентного середовища.

Кореневий веб-рівень включає основний запускний файл main.py, який реалізований на базі FastAPI. Також тут знаходиться локальна база даних db.sqlite, модуль конфігурації db.py та підсистема авторизації auth.py, що забезпечує доступ користувачів через форму login.html.

Саме цей рівень відповідає за взаємодію з користувачем, обробку HTTP-запитів та роботу з локальною БД. Керування бібліотеками та залежностями виконується окремо через файл requirements.txt.

Окремо виділений ізольований агентний рівень (/agents).Він являє собою вже самостійну частину мультиагентної системи де працюють агенти StoreAAgent, StoreBAgent та WarehouseAgent. Усі вони функціонують у межах асинхронного середовища SPADE.

Комунікаційний шлюз mas_api.py, який побудований на aiohttp являє собою сервер, що прослуховує порт 8001. Він виступає проміжною ланкою між FastAPI та XMPP-середовищем агентів. По суті, він працює як своєрідний «міст», перетворюючи HTTP-запити у внутрішні XMPP-повідомлення для агентної мережі.

Важливу роль у всій структурі виконує файл run_agents.py. Саме він запускає всю агентну мережу і фактично виступає оркестратором системи. Під час запуску скрипт автоматично реєструє агентів на XMPP-сервері, ініціює їх внутрішні цикли поведінки (CyclicBehaviour).

Також окрему роль відіграє файл requirements_agents.txt, який знаходиться всередині каталогу агентів. Його наявність дозволяє розгортати агентний блок окремо від основного веб-сайту, наприклад на іншому сервері або як окремий мікросервіс. Через це вся система виходить більш гнучкою з точки зору масштабування та подальшого розділення компонентів.

3.2 Опис реалізації агентів

3.2.1. Детальний опис та логіка роботи агента-координатора (StoreAAgent)

Агент-координатор StoreAAgent у даній системі виконує одну з найважливіших ролей. Можна сказати, що саме через нього проходить майже вся логіка роботи мультиагентного середовища. Він приймає повідомлення від агентного шлюзу, обробляє їх, звертається до складів або резервного магазину, збирає відповіді та повертає фінальний результат. По суті саме цей агент об'єднує роботу всіх інших компонентів системи в єдиний процес.

Після запуску StoreAAgent переходить у режим постійного очікування повідомлень. Для цього використовується CyclicBehaviour, завдяки якому агент не завершує свою роботу після виконання однієї операції. Це досить важливий момент, адже система повинна бути готовою приймати нові запити постійно. Тобто агент працює за простим принципом: отримав повідомлення, обробив його та знову повернувся в режим очікування.

Коли надходить новий запит, координатор спочатку перевіряє його тип. Для цього використовується параметр request_type. У поточній реалізації передбачено три основні режими роботи: search, verify та sell. Саме від цього значення залежить яка частина логіки буде виконуватись далі.

Далі агент намагається отримати інформацію з тіла повідомлення:

```
request_data = json.loads(msg.body)
product_id = request_data.get("product")
client_price = request_data.get("client_price")
client_qty = request_data.get("client_qty", 1)
```

Тут координатор отримує назву товару, ціну яку бачить користувач та необхідну кількість. Ці параметри потім використовуються практично на всіх наступних етапах роботи. Проте в коді передбачений і запасний варіант. Якщо повідомлення прийшло не у вигляді JSON, агент просто бере назву товару безпосередньо з поля body. Це дозволяє уникнути зайвих помилок та робить систему більш стійкою до різних форматів повідомлень.

Після цього координатор починає пошук доступних пропозицій через метод get_sorted_offers(). Спочатку він звертається до локальних складів w1@localhost та w2@localhost. Кожному складу надсилається окремий запит із перформативом query:

```
req = Message(to=w, body=product, thread=thread_id)
req.set_metadata("performative", "query")
await behaviour.send(req)
```

Потім агент просто чекає відповіді від складів. Тут використовується асинхронне очікування, тому система не блокується повністю під час роботи. Якщо склад повідомляє, що товар є і його кількість більша за нуль, така пропозиція додається до списку результатів. При цьому додатково перевіряється thread повідомлення. Це потрібно для того, щоб випадково не обробити відповідь від іншого запиту, який міг виконуватись паралельно.

Після того як відповіді отримані, результати сортуються за ціною:

```
results.sort(key=lambda x: x["price"])
```

Все робиться досить просто. На першому місці опиняється найдешевший варіант, а отже саме його система буде розглядати як основну пропозицію для користувача. Таким чином покупцю автоматично показується найбільш вигідна ціна.

Якщо ж після опитування складів з'ясовується, що потрібного товару немає взагалі, координатор не завершує роботу одразу. У такому випадку він переходить до резервного сценарію та звертається до StoreBAgent. По суті тут реалізовано каскадний пошук. Тобто система спочатку намагається знайти товар на власних складах і лише якщо це не вдалося - звертається до зовнішнього партнера. Завдяки цьому навіть відсутність товару на локальних складах ще не означає, що користувач не зможе його придбати.

Подальша робота агента вже залежить від режиму роботи.

У режимі search координатор формує відповідь для каталогу товарів. Якщо товар знайдено, користувач отримує інформацію про найкращу ціну, джерело постачання та доступну кількість товару. Якщо ж товар відсутній, повертається статус not_found.

Режим verify використовується перед оформленням заявки. Тут система ще раз перевіряє чи не змінились умови з моменту пошуку товару. Наприклад, користувач міг додати товар у кошик кілька хвилин тому, а за цей час хтось інший уже викупив частину залишків або змінилась ціна.

Для перевірки використовуються такі умови:

```
price_match = (best["price"] == client_price)
qty_ok = (best["qty"] >= client_qty)
```

Якщо ціна залишилась тією ж самою і товару вистачає, агент повертає статус "ok". Якщо ж хоча б одна з умов більше не виконується, повертається статус "alert". У такому випадку користувач бачить повідомлення про зміну умов і вже сам вирішує чи продовжувати оформлення заявки.

Найбільша частина логіки знаходиться саме у режимі sell. Тут координатор уже не просто перевіряє інформацію, а виконує реальне списання товару. Після пошуку доступних пропозицій агент послідовно перебирає всіх

знайдених постачальників. Для кожного з них визначається кількість товару, яку можна використати для поточного замовлення:

```
take_qty = min(remaining_qty, shop["qty"])
```

Далі формується окремий запит на списання товару. Якщо постачальник підтверджує операцію, необхідна кількість зменшується і система переходить до наступного кроку. Якщо ж відповіді немає або постачальник відмовився виконувати операцію, координатор просто переходить до іншого джерела.

Цікавим моментом є те, що одне замовлення може збиратися одразу з кількох постачальників. Наприклад, на одному складі може бути лише частина необхідної кількості товару. У такому випадку система автоматично шукає решту на інших складах. Для користувача це виглядає як одна покупка, хоча всередині координатор фактично збирає її з декількох джерел.

Після завершення всіх операцій агент формує підсумкову відповідь для шлюзу. У відповідь включається список виконаних операцій, а також прапорець final_success, який показує чи вдалося повністю виконати заявку. Саме на основі цієї інформації веб-рівень надалі приймає рішення щодо commit або rollback транзакції у базі даних.

Отже, StoreAAgent фактично виконує роль центрального керуючого елемента всієї мультиагентної системи. Він приймає запити, шукає товари, перевіряє актуальність даних, координує списання між різними постачальниками та забезпечує узгоджену роботу всіх інших агентів.

3.2.2 Детальний опис та логіка роботи агентів зовнішнього постачальника та складів (StoreBAgent та WarehouseAgent)

Агент StoreBAgent у даній системі виконує роль зовнішнього резервного постачальника. Якщо StoreAAgent можна вважати центральним координатором усієї системи, то StoreBAgent більше схожий на окремого незалежного учасника, який має власні залишки товарів та самостійно приймає рішення щодо можливості продажу. По суті він моделює роботу стороннього магазину, до якого система звертається тоді, коли на локальних складах не вдалося знайти необхідний товар або виникає потреба у додатковому постачальнику.

Для зберігання інформації про товари використовується проста структура даних у пам'яті програми:

```
STORE_B_DATA = {
    "milk": {"available": True, "quantity": 5, "price": 50},
    "oil": {"available": True, "quantity": 2, "price": 20},
}
```

Тут для кожного товару зберігається його кількість, ціна та ознака доступності. Звичайно, у реальному магазині для цього використовувалася б база даних, проте для демонстрації роботи мультиагентної системи такого підходу цілком достатньо.

Після запуску агент переходить у режим постійного очікування повідомлень за допомогою CyclicBehaviour. Тобто він безперервно працює у фоновому режимі та чекає поки інші агенти звернуться до нього із новим запитом. Якщо повідомлення не надходить, агент просто продовжує очікування.

Коли приходить новий запит, StoreBAgent спочатку визначає тип повідомлення та намагається отримати необхідні параметри з його тіла. Зокрема його цікавить назва товару та кількість, яка запитується. Якщо повідомлення надійшло не у форматі JSON, використовується резервний варіант обробки, коли назва товару береться безпосередньо з поля body. Це дозволяє агенту працювати з різними форматами повідомлень без необхідності жорстко контролювати структуру кожного запиту.

Подальша логіка роботи залежить від типу отриманого повідомлення. По суті можна виділити два основні режими роботи.

Перший режим використовується під час звичайного пошуку товару. У цьому випадку координатор просто хоче дізнатися чи є товар у наявності та яка його поточна ціна. Якщо потрібний товар знайдено, агент повертає інформацію про його кількість, вартість та джерело постачання. Якщо ж товар відсутній, повертається повідомлення про відмову. Саме таким чином StoreBAgent бере участь у каскадному пошуку товарів, коли локальні склади не можуть запропонувати потрібну позицію.

Другий режим роботи використовується під час безпосереднього оформлення заявки. У цьому випадку агент може приймати повідомлення як з перформативом propose, так і з перформативом request. Для цього в коді використовується універсальна перевірка:

```
if performative in ["propose", "request"]:
```

На перший погляд може здатися, що ці два перформативи мають різне призначення і потребують окремої логіки. Проте в даній реалізації було прийнято дещо інше рішення. Координатор фактично надсилає команду на виконання мікроугоди та списання певної кількості товару, тому для StoreBAgent більш важливим є сам факт необхідності виконання операції, ніж конкретний тип перформативу.

Завдяки такому підходу агент залишається більш гнучким і може коректно працювати незалежно від того, який саме механізм взаємодії використовує координатор. Це також спрощує подальший розвиток системи, оскільки логіка виконання угоди залишається однаковою для обох варіантів повідомлень.

Після отримання такого запиту агент перевіряє декілька умов. Насамперед контролюється наявність товару та достатність його кількості для виконання заявки. Однак цим перевірка не обмежується. У системі додатково реалізований механізм випадкової відмови:

```
rejection_chance = 0.20
is_willing_to_sell = random.random() > rejection_chance
```

По суті це імітація поведінки реального зовнішнього постачальника. У практичних умовах сторонній магазин може відмовити через власну політику, внутрішні обмеження або інші причини. Тому навіть за наявності товару угода не завжди буде підтверджена.

Якщо всі перевірки пройдено успішно, агент виконує реальне списання товару:

```
stock["quantity"] -= qty_requested
```

Після цього кількість товару на складі зменшується. Якщо залишок стає рівним нулю, товар автоматично позначається як недоступний для подальших

продажів. Далі формується відповідь зі статусом success, яка повертається координатору.

Якщо ж товару недостатньо або спрацьовує механізм відмови, агент повертає повідомлення зі статусом rejected. Цікавим моментом є те, що StoreBAgent не ігнорує запит повністю, а надсилає явну відповідь про відмову. Завдяки цьому координатору не потрібно чекати завершення таймауту і він може швидше перейти до інших варіантів обробки заявки.

Отже, StoreBAgent виступає незалежним зовнішнім постачальником у мультиагентній системі. Він підтримує пошук товарів, бере участь у процесі оформлення заявок, контролює власні залишки та самостійно приймає рішення щодо підтвердження або відхилення угоди. Саме завдяки цьому компоненту система отримує можливість працювати навіть у тих випадках, коли локальні склади не можуть повністю задовольнити запит користувача.

Агент WarehouseAgent у даній системі виконує роль локального складу, на якому безпосередньо зберігаються товари. На відміну від StoreAAgent, який займається координацією всієї мережі, складський агент не приймає складних рішень і не займається пошуком постачальників. Його завдання значно простіше - зберігати інформацію про власні залишки товарів, повідомляти їх актуальний стан та виконувати списання після отримання відповідної команди.

У проекті одночасно працює декілька таких агентів, зокрема w1@localhost та w2@localhost. Кожен із них має власні залишки товарів та власні ціни. Завдяки цьому один і той самий товар може коштувати по-різному на різних складах, що надалі дозволяє координатору обирати найбільш вигідну пропозицію для користувача.

Після запуску агент отримує доступ до своїх локальних даних через загальну структуру DATA_STORAGE:

```
self.my_data = DATA_STORAGE.get(str(self.jid), {})
```

По суті саме тут визначається з якими товарами буде працювати конкретний склад. Наприклад, агент w1 бачить лише свої залишки, а агент w2

працює вже зі своїм набором даних. Завдяки цьому досягається ізоляція складів один від одного і кожен агент відповідає лише за власну частину інформації.

Після ініціалізації WarehouseAgent переходить у режим постійного очікування повідомлень через CyclicBehaviour. Тобто він працює практично як невеликий фоновий сервіс, який чекає поки координатор або інший агент звернеться до нього із новим запитом. Якщо повідомлень немає, агент просто продовжує очікування.

Коли надходить новий запит, склад спочатку аналізує його тип через значення performative. Від цього залежить яку саме операцію необхідно виконати. По суті в агента існує два основні режими роботи.

Перший режим використовується під час оформлення заявки та списання товару. У цьому випадку координатор надсилає повідомлення з перформативом request, в якому міститься назва товару та кількість, яку необхідно списати зі складу.

Для цього агент спочатку розпаковує отримані дані:

```
data = json.loads(msg.body)
product_name = data.get("product")
qty_to_reduce = data.get("qty", 1)
```

Після цього виконується перевірка чи існує товар на даному складі та чи достатньо його кількості для виконання операції. Якщо залишків достатньо, відбувається реальне списання товару:

```
info["qty"] -= qty_to_reduce
```

Тобто склад фактично змінює власний стан і зменшує кількість доступної продукції. Саме тому цей режим можна вважати транзакційним, оскільки він вже впливає на дані системи, а не просто повертає інформацію.

Після успішного списання агент формує відповідь зі статусом success та повідомляє координатору про успішне завершення операції. Якщо ж за час оформлення заявки товар встиг закінчитися або його кількість стала меншою за необхідну, склад повертає повідомлення про помилку. Таким чином координатор може своєчасно перейти до іншого постачальника або скасувати операцію.

Другий режим роботи використовується значно частіше. Це звичайний інформаційний запит, який надходить під час пошуку товарів або перевірки актуальності даних перед оформленням заявки. У цьому випадку координатору не потрібно нічого списувати. Він просто хоче дізнатися чи є товар у наявності та скільки він коштує на конкретному складі.

Після отримання такого запиту агент перевіряє власне сховище даних та додатково отримує інформацію про товар із каталогу PRODUCT_CATALOG. Якщо товар знайдено, формується відповідь такого типу:

```
response_body = {
    "status": "available",
    "product": product_name,
    "qty": info["qty"],
    "price": info["price"],
    "description": catalog_info["description"],
    "warehouse_id": str(self.agent.jid)
}
```

Тут передається не тільки ціна та кількість товару, а й текстовий опис та ідентифікатор складу, який сформував відповідь. Надалі саме ці дані використовуються координатором для формування списку доступних пропозицій.

Якщо ж потрібного товару на складі немає або його залишок дорівнює нулю, агент повертає відповідь зі статусом not_found. При цьому склад не намагається самостійно знайти альтернативу або звернутися до інших агентів. Такі рішення приймає виключно StoreAAgent, який відповідає за координацію всієї мережі.

Отже, WarehouseAgent можна розглядати як локального виконавця в мультиагентній системі. Він не займається бізнес-логікою, не приймає рішень щодо маршрутизації запитів та не взаємодіє безпосередньо з користувачем. Його головне завдання полягає у підтримці актуального стану власних залишків та швидкому виконанні запитів від координатора. Саме завдяки

такому розподілу обов'язків кожен агент виконує свою окрему функцію, а система залишається достатньо простою для розширення та підтримки.

3.3 Детальний опис та логіка роботи API-серверу

API-сервер, реалізований на базі FastAPI, виконує роль центральної точки взаємодії між користувачем, локальною базою даних та мультиагентною системою. Саме через нього проходять усі HTTP-запити від веб-інтерфейсу, а також здійснюється обмін даними з агентним шлюзом. По суті FastAPI виступає своєрідним посередником між звичайним веб-застосунком та агентним середовищем SPADE.

Після запуску сервер виконує ініціалізацію основних компонентів системи. Підключаються маршрути авторизації, механізм сесій SessionMiddleware та шаблонізатор Jinja2, який використовується для формування HTML-сторінок. Завдяки використанню сесій сервер може зберігати інформацію про авторизованого користувача та його корзину між різними HTTP-запитами.

Одним із основних завдань сервера є організація пошуку товарів. Коли користувач вводить назву товару у пошуковий рядок, FastAPI не виконує пошук самостійно. Замість цього він звертається до агентного шлюзу через сервіс MASSearchService:

```
response = requests.get(
    "http://localhost:8001/search",
    params={"query": query}
)
```

Далі отримана відповідь повертається на фронтенд у вигляді JSON-структури. Такий підхід дозволяє повністю відокремити веб-рівень від внутрішньої логіки мультиагентної системи. Для користувача це виглядає як звичайний пошук товарів, хоча фактично за ним стоїть взаємодія з агентами складів та координатором.

Після знаходження потрібного товару користувач може додати його до корзини для подальшого оформлення. У даному проекті корзина реалізована досить просто і зберігається безпосередньо у веб-сесії користувача. По суті це

тимчасове сховище, яке дозволяє накопичувати інформацію про майбутню заявку до моменту її остаточного підтвердження.

Для кожної доданої позиції у сесії зберігається кількість товару, його поточна ціна та джерело постачання, яке було визначене агентною мережею під час пошуку:

```
cart[pid] = {
    "quantity": 1,
    "price": data.get("price"),
    "source": data.get("source")
}
```

На перший погляд така структура може здатися досить простою, проте її цілком достатньо для реалізації механізму швидкого замовлення. Після додавання товару користувачу вже не потрібно повторно виконувати пошук або заново вказувати параметри заявки, оскільки вся необхідна інформація тимчасово зберігається у сесії.

Важливо, що корзина в даному проекті не записується у базу даних одразу після кожної зміни. Натомість усі дії виконуються безпосередньо у сесії користувача. Завдяки цьому система не створює зайвого навантаження на SQLite та не виконує непотрібних транзакцій до моменту фактичного оформлення покупки.

Для керування корзиною передбачено окремий набір маршрутів. Користувач може збільшувати або зменшувати кількість товару через маршрут update-cart, а також повністю видаляти окремі позиції за допомогою remove-from-cart. По суті на цьому етапі відбувається лише коригування локального стану корзини без будь-якого впливу на залишки товарів у агентній мережі.

Лише після натискання кнопки оформлення заявки запускається повноцінна процедура взаємодії з мультиагентною системою та перевірки актуальності умов покупки.

Найбільша частина логіки реалізована у маршруті checkout, який відповідає за оформлення замовлення. Саме тут відбувається взаємодія між

веб-рівнем, мультиагентною системою та базою даних. Після натискання користувачем кнопки оформлення сервер спочатку перевіряє чи не є корзина порожньою. Якщо ж у ній присутні товари, запускається процедура попередньої верифікації через агентну мережу.

На цьому етапі сервер послідовно звертається до агентного шлюзу та повторно перевіряє актуальність кожної позиції:

```
real_data = mas_service.verify_item(
    product_id,
    item_data.get("price")
)
```

Це необхідно тому, що між моментом пошуку товару та оформленням заявки міг пройти певний час. За цей період ціна могла змінитися або частина товару вже могла бути придбана іншим користувачем.

Цікавим моментом реалізації є механізм м'якої перевірки зміни ціни. Якщо під час повторної перевірки система виявляє, що товар став дорожчим, покупка не завершується автоматично. Натомість користувач отримує повідомлення про зміну умов:

```
if best_system_price > client_price:
```

При цьому нова ціна одразу оновлюється у корзині користувача. Завдяки цьому після підтвердження нових умов процедура оформлення може бути продовжена без повторного виникнення тієї ж помилки.

Якщо етап верифікації завершився успішно, сервер переходить до другої фази - реального оформлення заявки. Для кожного товару з корзини виконується виклик маршруту checkout агентного шлюзу:

```
sell_result = mas_service.checkout_item(
    product_id,
    fallback_price,
    item_data["quantity"]
)
```

Саме на цьому етапі координатор StoreAAgent запускає процедуру каскадного списання товарів між складами та зовнішнім постачальником.

FastAPI вже не бере участі у виборі джерел постачання, а лише очікує фінальний результат виконання операції.

Після отримання відповіді сервер аналізує інформацію про всі виконані мікроугоди. Особливістю реалізації є те, що кожна операція записується в базу даних окремим рядком. Наприклад, якщо одна заявка була сформована одразу з двох різних складів, у таблиці orders буде створено два окремих записи із зазначенням джерела постачання та фактичної кількості отриманого товару.

Для збереження результатів використовується SQLite. Після успішної обробки всіх позицій виконується підтвердження транзакції:

```
conn.commit()
```

Якщо ж під час оформлення виникає критична помилка, сервер виконує відкат змін:

```
conn.rollback()
```

Завдяки цьому забезпечується цілісність даних навіть у випадку збою окремих компонентів системи.

Після завершення всіх операцій корзина користувача автоматично очищується, а фронтенд отримує підсумковий звіт про виконання заявки. У звіті міститься інформація про статус замовлення та перелік виконаних операцій, які надалі можуть бути відображені у журналі замовлень користувача.

Окрім роботи із замовленнями, сервер також надає функціональність перегляду історії покупок. Для цього використовується маршрут orders, який отримує записи з таблиці orders та передає їх у шаблон orders.html. Користувач може переглядати свої попередні заявки, видаляти окремі записи або повністю очищати історію замовлень.

Отже, FastAPI-сервер виконує роль центрального веб-рівня системи. Він забезпечує авторизацію користувачів, роботу з корзиною, взаємодію з мультиагентним середовищем, збереження результатів у базі даних та формування веб-інтерфейсу. Саме через нього відбувається об'єднання всіх інших компонентів проекту в єдину працюючу систему.

3.4 Детальний опис та логіка роботи агентського шлюзу MAS API

Проміжний рівень MAS API у даному проекті виконує роль спеціального «моста» між звичайним FastAPI-сервером та мультиагентною системою SPADE. У проекті виникла проблема: веб-сервер працює через HTTP-запити, а агенти SPADE взаємодіють між собою зовсім по-іншому - через XMPP-повідомлення. Через це FastAPI не може напряму працювати з агентами так само просто, як із базою даних або звичайним REST API. Саме тому у системі був створений окремий проміжний рівень - MAS API, який приймає HTTP-запити від основного сервера, перетворює їх у повідомлення для агентів, чекає відповіді та повертає результат назад у вигляді звичайного JSON.

Цей рівень побудований на основі асинхронного фреймворку aiohttp і працює як окремий сервер на локальному порту 8001. Після запуску система створює HTTP-застосунок:

```
app=web.Application()
```

Далі сервер реєструє основні роути , через які FastAPI взаємодіє з мультиагентною системою:

```
app.add_routes([
web.get("/search", handle_search),
web.post("/verify", handle_verify),
web.post("/checkout", handle_checkout)
])
```

Кожен із цих маршрутів відповідає за свою окрему задачу. Наприклад, /search використовується для пошуку товарів та отримання найкращих цін, /verify перевіряє актуальність товару перед покупкою, а /checkout запускає реальне списання товару через агентів.

Основною частиною цього рівня є спеціальний агент-посередник ClientAgent. Саме через нього всі запити потрапляють у мультиагентну систему. Під час запуску агент створює словник results, у якому тимчасово зберігаються відповіді від інших агентів:

```
self.results= {}
```

Коли від FastAPI приходить новий HTTP-запит, система спочатку створює унікальний thread_id:

```
thread_id=str(uuid.uuid4())
```

Це потрібно для того, щоб не переплутати відповіді від агентів. Наприклад, якщо на сайт одночасно зайде декілька користувачів і всі почнуть шукати товари або оформляти замовлення, система повинна розуміти, яка саме відповідь належить конкретному запиту.

Після цього ClientAgent формує повідомлення для координатора StoreAAgent:

```
msg=Message(to="storea@localhost")
msg.body=str(query)
msg.thread=thread_id
```

У повідомлення також додаються метадані:

```
msg.set_metadata("performative", "inform")
msg.set_metadata("request_type", request_type)
```

Завдяки параметру request_type координатор розуміє, що саме потрібно зробити: звичайний пошук товару, перевірку перед покупкою чи реальне списання товару.

Для відправки повідомлення використовується поведінка SenderBehaviour, яка працює у режимі одноразової дії (OneShotBehaviour). Її задача дуже проста — надіслати повідомлення агенту:

```
awaitself.send(self.msg)
```

Після цього MAS API переходить у режим очікування відповіді від мультиагентної системи. Для цього запускається окрема циклічна поведінка ResponseListener, яка постійно слухає нові повідомлення:

```
msg=awaitself.receive(timeout=1)
```

Коли координатор або інший агент надсилає відповідь, ResponseListener зчитує її та зберігає у словнику за відповідним thread_id. У цей же час метод ask() у циклі перевіряє, чи з’явилась потрібна відповідь у словнику. Як тільки відповідь приходить, система забирає її через .pop(thread_id). Це дозволяє не тільки отримати результат, а й одразу видалити його з пам’яті, щоб словник не накопичував старі відповіді. Після цього дані перетворюються назад у Python-об’єкт через json.loads() і повертаються у функцію обробки роута.

Завдяки цьому система правильно зв’язує відповідь агентів із потрібним HTTP-запитом користувача.

Наприклад, коли FastAPI викликає ендпоінт /search, MAS API отримує HTTP-запит і передає його у мультиагентну систему через метод:

```
result=await client.ask(query)
```

Після цього відповідь від агентів перетворюється у звичайний JSON-об’єкт і повертається назад FastAPI-серверу.

Схожим чином працюють і ендпоінти /verify та /checkout. Вони отримують JSON-дані від FastAPI, упаковують їх у повідомлення для агентів та передають координатору:

```
body_payload=json.dumps({
"product": product,
"client_price": price,
"client_qty": qty
})
```

Після завершення обробки результат повертається назад через:

```
returnweb.json_response(result)
```

Варто також зазначити, що для ендпоінта /search реалізовано додаткове форматування відповіді. Якщо координатор знаходить товар, MAS API формує масив results, у який підставляються назва товару, статус, найкраща ціна та поле display_qty, що показує співвідношення доступного товару до загальної кількості у системі.

Таким чином MAS API бере на себе всю взаємодію зі SPADE-агентами та XMPP-повідомленнями. Основний FastAPI-сервер працює з мультиагентною системою так, ніби це звичайний HTTP-сервіс. Завдяки такому підходу вдалося об’єднати веб-застосунок та мультиагентну систему в одну спільну архітектуру, де кожен рівень виконує свою окрему задачу.

3.5 Висновки до третього розділу

У третьому розділі виконано повну програмну реалізацію системи та описано її модульну структуру. Було детально розглянуто роботу веб-сервера на FastAPI, який відповідає за інтерфейс користувача, роботу з кошиком та

збереження історії замовлень у базі даних SQLite. Також описано реалізацію проміжного шлюзу MAS API на базі aiohttp, який успішно пов'язує веб-сайт із мультиагентною системою, перетворюючи HTTP-запити в асинхронні повідомлення для агентів. Крім того, задокументовано логіку роботи самих агентів у середовищі SPADE: координатора StoreAAgent, локальних складів WarehouseAgent та зовнішнього постачальника StoreBAgent. Створений програмний комплекс підтвердив свою працездатність і повністю готовий до практичного використання.

ВИСНОВКИ

В процесі виконання роботи була досягнута мета, яка полягала у розробці вебплатформи для децентралізованої торгівлі на основі взаємодії мережі незалежних інтелектуальних агентів.

Для досягнення поставленої мети було виконано наступні сформовані завдання роботи:

- було проведено аналіз специфіки побудови сучасних систем електронної комерції;
- виконано аналіз особливостей і призначення розподілених архітектур та технологій мультиагентних систем;
- проведено аналіз особливостей організації логістичних процесів та керування залишками товарів;
- розглянуто існуючі програмні аналоги та обґрунтовано доцільність відмови від централізованих баз даних;
- обґрунтовано вибір програмних засобів розробки (FastAPI, SPADE, XMPP, SQLite);
- спроєктовано дворівневу архітектуру та модель взаємодії агентного середовища;
- дійснено розробку системи, контроль транзакцій та опис прикладів використання основних функцій платформи.

Розроблена децентралізована платформа реалізована у вигляді вебзастосунку на базі Python-фреймворків FastAPI та SPADE з підтримкою транзакційного захисту даних і асинхронного зв'язку через протокол XMPP. Система автоматично перенаправляє замовлення на склади партнерів у разі локального дефіциту товарів, що дозволяє оптимізувати витрати на центральну інфраструктуру та підвищити відмовостійкість торгової мережі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1.Заяць О. І., Капко Я. Є. Сучасні тенденції розвитку електронної комерції. Економіка і суспільство. - 2024. - № 28. - URL: https://economyandsociety.in.ua/index.php/journal/article/download/2893/2817/ (дата звернення: 06.06.2026).

2. Гринько Т. В., Патлаха В. В. Е-комерція в Україні: сучасні тренди у роздрібній торгівлі. Екологічні науки. - 2024. - № 11. - С. 6-13. - URL: https://eco-science.net/wp-content/uploads/2024/11/11.24._topic_Tetyana-V.-Grynko-Viktoria-V.-Patlakha-6-13.pdf (дата звернення: 06.06.2026).

3. Іпполітова І. Я. Перспективи розвитку електронної торгівлі в Україні в умовах цифровізації економіки. Економіка і суспільство. - 2023. - Вип. 47. -URL: https://economyandsociety.in.ua/index.php/journal/article/view/2106 (дата звернення: 06.06.2026).

4. Марусей Т. В. Основні тенденції розвитку ринку електронної комерції в Україні. Економіка і суспільство. - 2018. - Вип. 14. - С. 1011-1015. - URL: https://economyandsociety.in.ua/journals/14_ukr/144.pdf (дата звернення: 06.06.2026).

5. Кут М. Характеристики інфраструктури електронної торгівлі: виклики та перспективи розвитку. Економічний простір. - 2024. - № 191. - С. 46-50. - URL: https://prostir.pdaba.dp.ua/index.php/journal/article/view/1531 (дата звернення: 06.06.2026).

6. Шостак Л. В., Ліпич Л. Г., Павлова С. В. Вплив електронної комерції на інновації бізнес-моделей. Економіка та управління. - 2025. - № 12. - URL: https://journals.kymu.kyiv.ua/index.php/economy/article/view/263 (дата звернення: 06.06.2026).

7. Назаркевич М. А., Возний Я. В. Архітектурні підходи до розробки масштабованих веб-застосунків. Cybersecurity and Digital Resilience. - 2024. - № 24. - С. 341-350. - URL: https://www.csecurity.kubg.edu.ua/index.php/journal/article/view/613 (дата звернення: 06.06.2026).

8. Морозов А., Вакалюк Т., Кубрак Ю., Зосімович Д. Аналіз факторів впливу на архітектури програмних систем. Information Technology. - 2022. - № 1. - С. 44-52. - URL: https://journals.politehnica.dp.ua/index.php/it/article/download/82/70/71 (дата звернення: 06.06.2026).

9. Скворцов В. Х. Порівняння монолітної та мікросервісної архітектури у розробці вебзастосунків. Радіоелектроніка та молодь у XXI столітті: матеріали 28-го Міжнар. молодіж. форуму, 16-18 квітня 2024 р. Харків : ХНУРЕ, 2024. - Т. 4. - С. 152-154. (дата звернення: 06.06.2026).

10. Будьонний Д. Ю. Аналіз брокерів повідомлень RabbitMQ та Apache Kafka. Наука онлайн: міжнародний електронний науковий журнал. - 2018. - № 6. - URL: https://nauka-online.com/publications/information-technology/2018/6/analiz-brokerov-soobshhenij-rabbitmq-i-apache-kafka/ (дата звернення: 04.06.2026).

11. Tanenbaum A. S., van Steen M. Distributed Systems: Principles and Paradigms. 3rd ed. CreateSpace Independent Publishing Platform, 2017. - 596 p.

12. Wooldridge M. An Introduction to MultiAgent Systems. 2nd ed. Chichester: John Wiley & Sons, 2009. - 484 p.

13. Russell S., Norvig P. Artificial Intelligence: A Modern Approach. 4th ed. Pearson, 2020. - 1136 p.

14. Документація бібліотеки asyncio: офіційний сайт Python. [Електронний ресурс]. - Режим доступу: https://docs.python.org/3/library/asyncio.html. - Дата доступу: 05.06.2026.

15. Radhakrishnan G., Chithambaram V., Shunmuganathan K. L. Comparative Study of JADE and SPADE Multi Agent System. International Journal of Advanced Research (IJAR). - 2018. - Vol. 6, Issue 11. - P. 1035-1042. - DOI: 10.21474/IJAR01/8090.

16. Palanca J., Terrasa A., Julián V., Carrascosa C. SPADE 3: Supporting the New Generation of Multi-Agent Systems. IEEE Access. - 2020. - Vol. 8. - P. 182537-182549. - DOI: 10.1109/ACCESS.2020.3030837.

17. XMPP Standards Foundation. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120. - URL: https://xmpp.org/rfcs/rfc6120.html (дата звернення: 18.05.2026).

18. SPADE (Smart Python Agent Development Environment) документація. [Електронний ресурс]. - Веб-сайт. - Режим доступу: https://spade-mas.readthedocs.io/en/latest/readme.html. - Дата доступу: 22.05.2026.

19. Richardson L., Ruby S. RESTful Web Services. Sebastapol: O'Reilly Media, 2007.-448 p.

20. FastAPI документація. [Електронний ресурс]. - URL: https://fastapi.tiangolo.com/. Дата звернення: 20.05.2026.

21. aiohttp документація. [Електронний ресурс]. - URL: https://docs.aiohttp.org/. - Дата звернення: 19.05.2026.

22. sqlite3 - DB-API 2.0 interface for SQLite databases: Python 3.14.5 documentation. [Електронний ресурс]. - Режим доступу: https://docs.python.org/3/library/sqlite3.html. - Дата доступу: 05.06.2026.

23. Jinja Documentation: official website. [Електронний ресурс]. - Режим доступу: https://jinja.palletsprojects.com/. - Дата доступу: 05.06.2026.

24. UML 2.5.1 Specification. Object Management Group (OMG). - 2017. - URL: https://www.omg.org/spec/UML/2.5.1/ (дата звернення: 06.06.2026).

25. An overview of HTTP: MDN Web Docs. [Електронний ресурс]. - Режим доступу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview. - Дата доступу: 05.06.2026.

ДОДАТОК Б

Лістинг програмного коду

Папка tempates/ файл **index.html**

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Головна — MAS Панель</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body>

<div class="container-fluid">
    <div class="row">

        <div class="col-md-2 bg-success text-white min-vh-100 p-3">
            <h5 class="mb-1 text-center">MAS Панель</h5>
            <p class="text-center small text-white-50 mb-4">Вітаємо, {{ user
}}</p>

            <div class="nav flex-column nav-pills">
                <a href="/" class="nav-link w-100 text-start text-white active
fw-bold bg-dark mb-2">Головна</a>
                <a href="/orders" class="nav-link w-100 text-start text-white
mb-2">Заявки</a>
            </div>
            <hr>
            <div class="text-center"><a href="/logout" class="btn btn-sm
btn-outline-light">Вийти</a></div>
        </div>

        <div class="col-md-10 p-4">
            <h2 class="mb-4">Вітрина децентралізованих закупівель</h2>

            <div class="card p-4 shadow-sm mb-4">
                <h5> Децентралізований пошук товарів</h5>
                <div class="input-group mb-3" style="max-width: 500px;">
                    <input type="text" id="search" class="form-control"
placeholder="Пошук товару...">
```

```
                    <button class="btn btn-primary"
onclick="search()">Знайти</button>
                </div>
                <div id="results" class="mt-2"></div>
            </div>

            <div class="card p-4 shadow-sm">
                <h5> Поточна корзина (Специфікація MAS)</h5>
                <div id="cart" class="my-3 p-2 border rounded bg-light"></div>
                <button class="btn btn-success" style="max-width: 250px;"
onclick="processCheckout()">Оформити замовлення</button>
            </div>
        </div>

    </div>
</div>

<script>
// Автоматична загрузка корзини при старті сторінки
window.onload = function() {
    loadCart();
};

//  ФУНКЦИЯ ПОШУКУ
async function search() {
    const query = document.getElementById("search").value;
    if (!query.trim()) {
        alert("Введіть назву товару");
        return;
    }
    const response = await fetch(`/search?query=${query}`);
    const data = await response.json();

    const resultsDiv = document.getElementById("results");
    resultsDiv.innerHTML = "";

    if (!data.results || data.results.length === 0) {
        resultsDiv.innerHTML = "<div class='alert alert-warning small mt-2'>Нічого
не знайдено в мережі агентів</div>";
        return;
    }
```

```
    data.results.forEach(product => {
        const item = document.createElement("div");
        item.className = "p-2 border rounded my-2 bg-light d-flex
justify-content-between align-items-center shadow-sm";

        let statusText = "";
        let infoText = "";

        const isAvailable = product.status === "found" ||
                            product.status === "confirmed" ||
                            product.status === "available" || !product.status;

        if (isAvailable) {
            let sourceName = "склад";
            if (product.source) {
                sourceName = product.source.split("@")[0];
            }
            statusText = `в наявності (${sourceName})`;
            infoText = ` [${product.display_qty || '1/1'} | ${product.price || 0}
грн]`;
        } else {
            statusText = "немає в наявності";
        }


    item.innerHTML = `
            <div style="display: flex; justify-content: space-between;
align-items: center; width: 100%; margin: 8px 0; font-family: Arial, sans-serif;">
                <div>
                    <strong style="text-transform:
capitalize;">${product.name}</strong>
                    — ${statusText}<span style="color: #28a745; font-weight:
bold;">${infoText}</span>
                </div>
                <div>
                    ${isAvailable ?
                        `<button onclick="addToOrder('${product.name}',
${product.price || 0}, '${product.source || 'w1@localhost'}')"
                                style="margin-left: 10px; padding: 2px 8px;
cursor: pointer; background-color: #28a745; color: white; border: none; rounded:
4px;">
                            Купити
```

```
                        </button>`
                        : ""
                    }
                    <!-- Відміна позиції  -->
                    <button onclick="this.closest('.p-2').remove()"
                            style="margin-left: 10px; padding: 2px 6px; cursor:
pointer; background-color: #dc3545; color: white; border: none; font-weight:
bold;"
                            title="Скасувати">
                        ×
                    </button>
                </div>
            </div>
        `;
        resultsDiv.appendChild(item);
    });
}

// ДОДАВАННЯ ДО КОРЗИНИ
async function addToOrder(productName, price, source) {
    await fetch("/add-to-cart", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({
            product_id: productName,
            price: price,
            source: source
        })
    });
    loadCart();
}

// ФУНКЦІЯ ВІДОБРАЖЕННЯ КОРЗИНИ з КНОПКАМИ + ТА -
async function loadCart() {
    const response = await fetch("/cart-data");
    const data = await response.json();

    const cartDiv = document.getElementById("cart");
    cartDiv.innerHTML = "";

    if (!data.cart || data.cart.length === 0) {
```

```
        cartDiv.innerHTML = "<p class='text-muted small mb-0'>Корзина
порожня</p>";
        return;
    }

    data.cart.forEach(item => {
        const el = document.createElement("div");
        el.className = "d-flex align-items-center justify-content-between
border-bottom py-2";
        el.innerHTML = `
            <div>
                <b style="text-transform: capitalize;">${item.name}</b> — <span
class="text-success fw-bold">${item.price} грн</span>
            </div>
            <div class="d-flex align-items-center gap-2">
                <button class="btn btn-sm btn-outline-secondary py-0 px-2"
onclick="updateCart('${item.name}', 'dec')">-</button>
                <span class="fw-bold">${item.quantity} шт.</span>
                <button class="btn btn-sm btn-outline-secondary py-0 px-2"
onclick="updateCart('${item.name}', 'inc')">+</button>
                <button class="btn btn-sm btn-danger py-0 px-2 ms-3"
onclick="removeFromCart('${item.name}')">X</button>
            </div>
        `;
        cartDiv.appendChild(el);
    });
}

// ВИДАЛЕННЯ З КОРЗИНИ
async function removeFromCart(productId) {
    await fetch("/remove-from-cart", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ product_id: productId })
    });
    loadCart();
}

// ЗМІНА КІЛЬКОСТІ (+ и -)
async function updateCart(productId, action) {
    await fetch("/update-cart", {
        method: "POST",
```

```
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ product_id: productId, action: action })
    });
    loadCart();
}

// ДВУХФАЗНА ПРОВЕРКА ФУНКЦИЯ ЧЕКАУТА
async function processCheckout() {
    const cartResponse = await fetch("/cart-data");
    const cartData = await cartResponse.json();

    if (!cartData.cart || cartData.cart.length === 0) {
        alert("Ваша корзина порожня!");
        return;
    }


    const response = await fetch('/checkout', { method: 'POST' });
    const result = await response.json();

    // ОБРАБОТКА ЗМІНИ ЦІНИ (при сценарію довгого очікування)
    if (result.status === "alert" && result.type === "PRICE_CHANGED") {
        let userAgreed = confirm(
            `Умови змінилися децентралізованою мережею!\n\n` +
            `Поки ви оформляли замовлення, найкраща ціна змінилася.\n` +
            `Стара ціна: ${result.old_price} грн\n` +
            `Нова ціна: ${result.new_price} грн\n\n` +
            `Ви згодні продовжити оформлення за новою ціною?`
        );

        if (userAgreed) {
            console.log("Користувач погодився. На бэкенде нужно вызвать чекаут
заново.");

            await processCheckout();
        } else {
            alert("Оформлення скасовано користувачем.");
        }
        return;
    }

    // ФИНІЛ: УСПІШНИЙ ЗАКАЗ
```

```
    if (result.status === "success" || result.status === "confirmed" ||
result.message === "Обробка замовлення завершена") {
        alert("Вітаємо! Ваше замовлення успішно опрацьовано MAC та зафіксовано в
базі.");
        window.location.href = "/orders";
    } else {
        alert(result.message || "Помилка при оформленні замовлення агентами");
        loadCart();
    }
}
</script>

</body>
</html>
```

Пака tempates/ файл **orders.html**

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Мої Заявки — MAS Панель</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css"
rel="stylesheet">
    <style>

        .custom-table {
            width: 100%;
            border-collapse: collapse;
            font-family: Arial, sans-serif;
            font-size: 14px;
            box-shadow: 0 4px 6px rgba(0,0,0,0.05);
            border-radius: 8px;
            overflow: hidden;
        }
        .custom-table th {
            background-color: #f8f9fa;
            color: #495057;
            padding: 12px 16px;
            font-weight: 600;
            border-bottom: 2px solid #dee2e6;
            text-align: left;
```

```
        }
        .custom-table td {
            padding: 14px 16px;
            border-bottom: 1px solid #efefef;
            color: #333;
        }
        .custom-table tr:hover {
            background-color: #f1f3f5;
        }
        .status-badge {
            display: inline-block;
            padding: 4px 8px;
            border-radius: 4px;
            font-size: 12px;
            font-weight: 500;
        }
        .status-ok { background-color: #e6f4ea; color: #137333; border: 1px solid
#ceead6; }
        .status-alert { background-color: #fef7e0; color: #b06000; border: 1px
solid #feebc8; }
        .status-error { background-color: #fce8e6; color: #c5221f; border: 1px
solid #fad2cf; }
    </style>
</head>
<body>

<div class="container-fluid">
    <div class="row">

        <div class="col-md-2 bg-success text-white min-vh-100 p-3">
            <h5 class="mb-1 text-center">MAS Панель</h5>
            <p class="text-center small text-white-50 mb-4">Вітаємо, {{ user
}}</p>

            <div class="nav flex-column nav-pills">
                <a href="/" class="nav-link w-100 text-start text-white
mb-2">Головна</a>
                <a href="/orders" class="nav-link w-100 text-start text-white
active fw-bold bg-dark mb-2">Заявки</a>
            </div>
            <hr>
```

```
            <div class="text-center"><a href="/logout" class="btn btn-sm
btn-outline-light">Вийти</a></div>
        </div>

        <div class="col-md-10 p-4">
            <div class="d-flex justify-content-between align-items-center mb-4">
                <h3 style="color: #495057; margin: 0;">Історія замовлень (Аудит
MAS)</h3>
                <div class="d-flex gap-2">
                    <button class="btn btn-sm btn-danger"
onclick="clearOrdersHistory()">Очистити історію</button>
                    <button class="btn btn-sm btn-outline-primary"
onclick="window.location.reload()">Оновити статуси</button>
                </div>
            </div>

            <div class="card p-4 shadow-sm">
                <table class="table custom-table mb-0">
                    <thead>
                        <tr>
                            <th># ID Заявки</th>
                            <th>Товар</th>
                            <th style="text-align: center;">Кількість</th>
                            <th>Ціна за одиницю</th>
                            <th>Статус та Склад</th>
                            <th style="text-align: right;">Дата</th>
                            <th style="text-align: center;">Дія</th>
                        </tr>
                    </thead>
                    <tbody>
                        {% for order in orders %}
                        <tr>
                            <td class="fw-bold text-secondary">#{{ order.id if
order.id else loop.index }}</td>

                            <td style="font-weight: bold; text-transform:
capitalize;">
                                {{ order.name }}
                            </td>

                            <td style="text-align: center; color: #6c757d;">
                                {% if order.qty and order.qty != None %}
```

```
                                x{{ order.qty }}
                            {% else %}
                                x1
                            {% endif %}
                        </td>

                        <td class="fw-bold text-success">
                            {% if order.price %}{{ order.price }} грн.{% else
%}—{% endif %}
                        </td>

                        <td>
                            {% if "Pending" in order.status %}
                                <span class="badge status-badge status-alert
pending-magic"
                                      id="status-text-{{ loop.index }}"
                                      data-order-id="{{ loop.index }}">
                                    Очікування відповіді (storeb)
                                </span>
                            {% elif "confirmed" in order.status or "Completed"
in order.status %}
                                <span class="badge status-badge status-ok">{{
order.status }}</span>
                            {% elif "partial" in order.status %}
                                <span class="badge status-badge
status-alert">{{ order.status }}</span>
                            {% else %}
                                <span class="badge status-badge
status-error">{{ order.status }}</span>
                            {% endif %}
                        </td>

                        <td style="text-align: right; color: #6c757d;
font-size: 13px;">
                            {{ order.created_at }}
                        </td>

                        <td style="text-align: center;">
                            <button class="btn btn-sm btn-link text-danger p-0
fw-bold"
                                    style="text-decoration: none;"
```

```
                                        onclick="deleteSingleOrder('{{ order.id if
order.id else loop.index }}')">
                                    Видалити
                                </button>
                            </td>
                        </tr>
                        {% else %}
                        <tr>
                            <td colspan="7" style="text-align: center; color:
#6c757d; padding: 30px;">
                                Замовлень поки немає
                            </td>
                        </tr>
                        {% endfor %}
                    </tbody>
                </table>
            </div>
        </div>

    </div>
</div>

<script>
// 1. Асинхронне видалення одного рядка за його ID
async function deleteSingleOrder(orderId) {
    if (!confirm(`Вилучити цей запис про замовлення з журналу?`)) return;

    try {
        const response = await fetch(`/delete-order/${orderId}`, { method: "POST"
});
        const result = await response.json();

        if (result.status === "success") {
            window.location.reload();
        } else {
            alert("Помилка видалення: " + result.message);
        }
    } catch (err) {
        alert("Помилка з'єднання з API сервером");
    }
}
```

```
// 2. Повне очищення журналу для підготовки демонстрації
async function clearOrdersHistory() {
    let confirmClear = confirm("Ви впевнені, що хочете повністю очистити весь
журнал заявок?");
    if (!confirmClear) return;

    try {
        const response = await fetch("/clear-orders", { method: "POST" });
        const result = await response.json();

        if (result.status === "success") {
            alert("Журнал успішно очищено!");
            window.location.reload();
        } else {
            alert("Помилка очищення: " + result.message);
        }
    } catch (err) {
        alert("Критична помилка API сервера");
    }
}
document.addEventListener("DOMContentLoaded", () => {
    const statusBadges = document.querySelectorAll('.status-badge');

    statusBadges.forEach((badge) => {
        const statusText = badge.textContent.trim();

        if (statusText.toLowerCase().includes('storeb'))
            const realStatusFromDB = badge.innerHTML;
            badge.classList.remove('status-ok', 'status-error');
            badge.classList.add('status-alert');
            badge.innerHTML = `
                <span class="spinner-border spinner-border-sm me-1" role="status"
aria-hidden="true"></span>
                МАС: Опитування Магазину Б...
            `;

            setTimeout(() => {

                badge.classList.remove('status-alert');
                if (statusText.toLowerCase().includes('confirm') ||
statusText.toLowerCase().includes('ok')) {
                    badge.classList.add('status-ok');
```

```
                } else {
                    badge.classList.add('status-error');
                }

                badge.innerHTML = realStatusFromDB;
                badge.style.transition = "all 0.4s ease";
                badge.style.opacity = "0.3";
                setTimeout(() => badge.style.opacity = "1", 50);

            }, 2500); // Задержка 2.5 секунды
        }
    });
});
</script>

</body>
</html>
```

Папка tempates/ файл **login.html**

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Вхід</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
<body class="bg-light">

<div class="container mt-5">
    <div class="row justify-content-center">
        <div class="col-md-5">
            <div class="card shadow">
                <div class="card-header bg-primary text-white text-center">
                    <h4>Вхід</h4>
                </div>
                <div class="card-body">

                    {% if error %}
                        <div class="alert alert-danger">{{ error }}</div>
                    {% endif %}
```

```
                    <form method="post" action="/login">
                        <div class="mb-3">
                            <label class="form-label">Логін</label>
                            <input type="text" name="username"
class="form-control" required>
                        </div>
                        <div class="mb-3">
                            <label class="form-label">Пароль</label>
                            <input type="password" name="password"
class="form-control" required>
                        </div>
                        <div class="d-grid">
                            <button type="submit" class="btn
btn-success">Увійти</button>
                        </div>
                    </form>

                </div>
            </div>
        </div>
    </div>
</div>

</body>
</html>
```

Файл auth.py

```
from fastapi import APIRouter, Request, Form
from fastapi.responses import RedirectResponse
from fastapi.templating import Jinja2Templates
from db import get_user

router = APIRouter()
templates = Jinja2Templates(directory="templates")
def authenticate_user(username: str, password: str):
    user = get_user(username)

    if not user:
        return None
```

```
    # user = (id, username, password)
    if user[2] == password:
        return user[1]

    return None
@router.get("/login")
async def login_page(request: Request):
    return templates.TemplateResponse("login.html", {
        "request": request,
        "error": ""
    })


@router.post("/login")
async def login(request: Request, username: str = Form(...), password: str =
Form(...)):
    user = authenticate_user(username, password)

    if not user:
        return templates.TemplateResponse("login.html", {
            "request": request,
            "error": "Невірный логин або пароль"
        })

    request.session["user"] = user
    return RedirectResponse("/", status_code=303)
@router.get("/logout")
async def logout(request: Request):
    request.session.clear()
    return RedirectResponse("/login")
```

Файл db.py

```
import sqlite3

conn = sqlite3.connect("db.sqlite")
cursor = conn.cursor()

# users
cursor.execute("""
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT,
```

```
    password TEXT
)
""")

# products
cursor.execute("""
CREATE TABLE IF NOT EXISTS products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT
)
""")

def get_user(username):
    conn = sqlite3.connect("db.sqlite")
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM users WHERE username = ?", (username,))
    user = cursor.fetchone()

    conn.close()
    return user

conn.commit()
conn.close()
```

Файл main.py

```
from fastapi import FastAPI, Request
from fastapi.responses import RedirectResponse, HTMLResponse
from fastapi.templating import Jinja2Templates
from starlette.middleware.sessions import SessionMiddleware
import secrets
import sqlite3
from auth import router as auth_router
from pydantic import BaseModel
import datetime
from pydantic import BaseModel
import random
from fastapi import Body
import requests
import asyncio
```

```
class CartUpdate(BaseModel):
    product_id: str
    action: str
class OrderRequest(BaseModel):
    product_id: str

app = FastAPI()
# подключення сесії
app.add_middleware(SessionMiddleware, secret_key=secrets.token_hex(32))

# подключення auth
app.include_router(auth_router)

templates = Jinja2Templates(directory="templates")

class MASSearchService:

    def search(self, query):
        if not query.strip():
            return []

        response = requests.get(
            "http://localhost:8001/search",
            params={"query": query}
        )
        data = response.json()

        return [
            {
                "name": item.get("name"),
                "status": item.get("status"),
                "source": item.get("source"),
                "display_qty": item.get("display_qty", "0/0"),
                "price": item.get("price", 0)
            }
            for item in data.get("results", [])
        ]

    # МЕТОД для верифікації перед купівлею(POST)
    def verify_item(self, product_id, price):
        response = requests.post(
            "http://localhost:8001/verify",
```

```
            json={
                "product": product_id,
                "price": price  # <--- Передаем цену!
            }
        )

        return response.json()


    def checkout_item(self, product_id, price, quantity):
        response = requests.post(
            "http://localhost:8001/checkout",
            json={
                "product": product_id,
                "client_price": price,
                "client_qty": quantity
            }
        )

        return response.json()
@app.get("/", response_class=HTMLResponse)
async def index(request: Request):
    user = request.session.get("user")
    if not user:
        return RedirectResponse("/login")
    products = get_products()
    return templates.TemplateResponse("index.html", {
        "request": request,
        "user": user,
        "products": products
    })


def get_products():
    conn = sqlite3.connect("db.sqlite")
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM products")
    products = cursor.fetchall()
    conn.close()
    return products


search_service = MASSearchService()
```

```
@app.get("/search")
async def search(query: str):
    results = search_service.search(query)
    return {"results": results}

@app.post("/order")
async def create_order(request: Request, order: OrderRequest):
    user = request.session.get("user")

    if not user:
        return {"message": "Не авторизований"}

    conn = sqlite3.connect("db.sqlite")
    cursor = conn.cursor()

    cursor.execute(
        "INSERT INTO orders (user, product_id) VALUES (?, ?)",
        (user, order.product_id)
    )

    conn.commit()
    conn.close()

    return {"message": "Товар додано до замовлення"}
@app.get("/orders")
async def my_orders(request: Request):
    user = request.session.get("user")

    conn = sqlite3.connect("db.sqlite")
    conn.row_factory = sqlite3.Row
    cursor = conn.cursor()

    cursor.execute("""
        SELECT
            id,
             price,
            product_id AS name,
            status,
            quantity AS qty,
            created_at
        FROM orders
        WHERE user = ?
```

```
        ORDER BY id DESC
    """, (user,))

    orders = cursor.fetchall()
    conn.close()

    return templates.TemplateResponse("orders.html", {
        "request": request,
        "user": user,
        "orders": orders
    })
@app.post("/add-to-cart")
async def add_to_cart(request: Request, data: dict = Body(...)):
    cart = request.session.get("cart", {})
    pid = data.get("product_id")

    if pid in cart:
        cart[pid]["quantity"] += 1
    else:

        cart[pid] = {
            "quantity": 1,
            "price": data.get("price"),
            "source": data.get("source")
        }

    request.session["cart"] = cart
    return {"message": "Добавлено"}
@app.get("/cart-data")
async def get_cart(request: Request):
    cart = request.session.get("cart", {})
    items = []
    for pid, info in cart.items():
        items.append({
            "id": pid,
            "name": pid,
            "quantity": info["quantity"],
            "price": info["price"]
        })
    return {"cart": items}
@app.post("/remove-from-cart")
async def remove_from_cart(request: Request, order: OrderRequest):
```

```
    cart = request.session.get("cart", {})

    pid = str(order.product_id)

    if pid in cart:
     del cart[pid]

    request.session["cart"] = cart

    return {"message": "Видалено"}
@app.post("/checkout")
async def checkout(request: Request):
    user = request.session.get("user")
    cart = request.session.get("cart", {})

    if not cart:
        return {"message": "Корзина пуста"}

    mas_service = MASSearchService()

    # --- ЭТАП 1: ВЕРИФИКАЦІЯ ПЕРЕД КУПІВЛЕЮ ---
    for product_id, item_data in list(cart.items()):
        real_data = mas_service.verify_item(product_id, item_data.get("price"))
        if not real_data:
            continue

        best_system_price = real_data.get("best_price", 0)
        client_price = item_data.get("price", 0)



        if best_system_price > client_price:

            cart[product_id]["price"] = best_system_price
            request.session["cart"] = cart  # збереження сесії
            return {
                "status": "alert",
                "type": "PRICE_CHANGED",
                "old_price": client_price,
                "new_price": best_system_price
            }

   # --- ЕТАП 2  СПИСАННЯ ЧЕРЕЗ МАС та  ЗАПИС В БД ---
```

```
    await asyncio.sleep(2.0)
    conn = sqlite3.connect("db.sqlite")
    cursor = conn.cursor()

    try:
        is_anything_bought = False  # Флаг - невдалося купити
        all_transactions_report = []
        created_at = datetime.datetime.now().strftime("%d.%m %H:%M")

        for product_id, item_data in cart.items():

            fallback_price = item_data.get("price", 0)

            # Звертаємося до Агента А для реального каскадного списання
            sell_result = mas_service.checkout_item(
                product_id,
                fallback_price,
                item_data["quantity"]
            )

            # новый статус від Агента А
            status_answer = sell_result.get("status")

            if status_answer == "processed":
                confirmed_items = sell_result.get("items", [])

                # Запис кожної відповіді (и confirmed, и rejected) до базу данних!
                for item in confirmed_items:
                    # Формируем статус, например: "confirmed (w1@localhost)"
                    full_status = f"{item['status']} ({item['source']})"
                    qty_part = item["qty"]  # Сколько конкретно этот склад отдал

                    warehouse_price = item.get("price", fallback_price)

                    cursor.execute(
                        """
                        INSERT INTO orders (user, product_id, price, status,
quantity, created_at)
                        VALUES (?, ?, ?, ?, ?, ?)
                        """,
                        (user, product_id, warehouse_price, full_status, qty_part,
created_at)
```

```
                    )

                    if item['status'] == 'confirmed':
                        source_wallet = item.get("source")
                        is_anything_bought = True

                all_transactions_report.extend(confirmed_items)
            else:

                cursor.execute(
                    """
                    INSERT INTO orders (user, product_id, price, status, quantity,
created_at)
                    VALUES (?, ?, ?, ?, ?, ?)
                    """,
                    (user, product_id, fallback_price, f"rejected
(system_timeout)", item_data["quantity"], created_at)
                )

        # Фіксуємо транзакцию в БД
        conn.commit()
        conn.close()

        # Очищення корзини в сесії клиента
        request.session["cart"] = {}

        # Повертаємо повну відповідь для фронтенда
        return {
            "status": "success" if is_anything_bought else "failed",
            "message": "Обробка замовлення завершена",
            "table_data": all_transactions_report
        }

    except Exception as e:
        conn.rollback()
        conn.close()
        print(f" Критична помилка при оформлені в БД: {e}")
        return {"status": "error", "message": f"Технічна помилка: {str(e)}"}

@app.post("/update-cart")
async def update_cart(request: Request, data: CartUpdate):
    cart = request.session.get("cart", {})
```

```
    pid = str(data.product_id)

    if pid in cart:
        # ПРОВЕРКА: если это старые данные (просто число), превращаем их в новый
формат
        if isinstance(cart[pid], int):
            cart[pid] = {"quantity": cart[pid], "price": 0, "source": "unknown"}

        if data.action == "inc":
            cart[pid]["quantity"] += 1
        elif data.action == "dec":
            cart[pid]["quantity"] -= 1


            if cart[pid]["quantity"] <= 0:
                del cart[pid]

    request.session["cart"] = cart
    return {"message": "ok"}
@app.post("/delete-order/{order_id}")
async def delete_order(order_id: int, request: Request):
    user = request.session.get("user")
    if not user:
        return {"status": "error", "message": "Неавторизований користувач"}

    try:
        conn = sqlite3.connect("db.sqlite")
        cursor = conn.cursor()

        cursor.execute("DELETE FROM orders WHERE id = ? AND user = ?", (order_id,
user))

        conn.commit()
        conn.close()
        return {"status": "success", "message": f"Замовлення #{order_id}
видалено"}
    except Exception as e:
        if 'conn' in locals(): conn.close()
        return {"status": "error", "message": str(e)}

@app.post("/clear-orders")
async def clear_orders(request: Request):
```

```
    user = request.session.get("user")
    if not user:
        return {"status": "error", "message": "Неавторизований користувач"}

    try:
        conn = sqlite3.connect("db.sqlite")
        cursor = conn.cursor()

        # Очищення логів для користувача
        cursor.execute("DELETE FROM orders WHERE user = ?", (user,))

        conn.commit()
        conn.close()
        return {"status": "success", "message": "Історію успішно очищено"}
    except Exception as e:
        if 'conn' in locals(): conn.close()
        return {"status": "error", "message": str(e)}
```

Папка agents/ файл mas_api.py

```
from aiohttp import web
from spade.agent import Agent
from spade.message import Message
from spade.behaviour import OneShotBehaviour
import uuid
import asyncio
import json
import logging

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger("spade")


class SenderBehaviour(OneShotBehaviour):
    def __init__(self, msg):
        super().__init__()
        self.msg = msg

    async def run(self):
        await self.send(self.msg)
        print(f" Міст відправив повідомлення: {self.msg.body}", flush=True)
```

```
# ========================
# ClientAgent (мост к SPADE)
# ========================
class ClientAgent(Agent):
    async def setup(self):
        #  Словник зберігання  відповідей  по их thread_id
        self.results = {}
        print(f" Міст {self.jid} готовий до работ!", flush=True)

    async def ask(self, query, request_type="search"):
        thread_id = str(uuid.uuid4())
        msg = Message(to="storea@localhost")
        msg.body = str(query)
        msg.thread = thread_id

    # Упакування метаданих
        msg.set_metadata("performative", "inform")
        msg.set_metadata("request_type", request_type)

        self.add_behaviour(SenderBehaviour(msg))



        timeout = 10
        start_time = asyncio.get_event_loop().time()

        print(f" Ожидаю ответ для потока: {thread_id}...", flush=True)

        while (asyncio.get_event_loop().time() - start_time) < timeout:
            if thread_id in self.results:
                answer = self.results.pop(thread_id)
                data = json.loads(answer)
                return data

            await asyncio.sleep(0.2)

        return {"status": "error"}

from spade.behaviour import CyclicBehaviour

class ResponseListener(CyclicBehaviour):
    async def run(self):
        msg = await self.receive(timeout=1)
```

```
        if msg:
            print(f" Поведение-слушатель поймало письмо: {msg.body}", flush=True)

            self.agent.results[str(msg.thread)] = msg.body

# HTTP endpoint

async def handle_verify(request):
    data = await request.json()
    product = data.get("product")
    price = data.get("price")

    body_payload = json.dumps({
        "product": product,
        "client_price": price,
        "client_qty": 1
    })

    client = request.app["client"]
    result = await client.ask(body_payload, request_type="verify")

    return web.json_response(result)

async def handle_checkout(request):
    data = await request.json()
    product = data.get("product")
    price = data.get("client_price")
    qty = data.get("client_qty", 1)

    client = request.app["client"]
    body_payload = json.dumps({
        "product": product,
        "client_price": price,
        "client_qty": qty
    })

    result = await client.ask(body_payload, request_type="sell")

    return web.json_response(result)

async def handle_search(request):
    query = request.query.get("query", "")
```

```
    print(f" Запрос из браузера: {query}", flush=True)

    client = request.app["client"]
    result = await client.ask(query)
    if "product" not in result:
        return web.json_response({"results": []})

    return web.json_response({
    "results": [
        {
            "name": result.get("product"),
            "status": result.get("status"),
            "source": result.get("source", "warehouse"),
            "display_qty": result.get("display_qty", "0 / 0"),
            "price": result.get("best_price", 0),
            "total": result.get("total", 0)
        }
    ]
})

# =========================
# Startup / Cleanup
# =========================
async def start_background_tasks(app):
    client = ClientAgent("client@localhost", "password")
    await client.start(auto_register=True)

    client.add_behaviour(ResponseListener())

    app["client"] = client
    print(" ClientAgent (Міст) запущено зі слухачем")


async def cleanup_background_tasks(app):
    await app["client"].stop()
    print(" ClientAgent остановлен")


# Настройка додатку

app = web.Application()
app.add_routes([
```

```
    web.get("/search", handle_search),
    web.post("/verify", handle_verify),
    web.post("/checkout", handle_checkout) # Новый эндпоинт
])
app.on_startup.append(start_background_tasks)
app.on_cleanup.append(cleanup_background_tasks)

if __name__ == "__main__":
    web.run_app(app, port=8001)
```

Папка agents/ файл SoreAAgent.py

```
import json
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour
from spade.message import Message

class StoreAAgent(Agent):
    async def setup(self):
        self.add_behaviour(self.HandleRequest())
        print("Агент StoreA запущено та готовий до работи.")


    # МЕТОД ЗБІРКИ ТА СОРТУВАННЯ ПРОПОЗИЦІЙ

    async def get_sorted_offers(self, product, thread_id, behaviour):
        warehouses = ["w1@localhost", "w2@localhost"]
        results = []

        # 1. Опитування прямих поостачальників (складів)
        for w in warehouses:
            req = Message(to=w, body=product, thread=thread_id)
            req.set_metadata("performative", "query")
            await behaviour.send(req)

        # Збір відповідей від складів
        for _ in warehouses:
            reply = await behaviour.receive(timeout=2)
            if reply and reply.body:

                if reply.thread == thread_id:
```

```
                        try:
                            data = json.loads(reply.body)
                            if data.get("qty", 0) > 0:
                                data.setdefault("source", str(reply.sender))
                                results.append(data)
                        except:
                            pass

            # Сортування відповідей від складів
            if results:
                results.sort(key=lambda x: x["price"])

             # 2. КАСКАД: Опитування резерву (Магазин Б) коли на складів нема товару
            if not results:
                print(f"StoreA: На складах порожньо! Запит резервів у Магазину Б для {product}...")
                    store_b_req = Message(to="storeb@localhost", body=product, thread=thread_id)
                store_b_req.set_metadata("performative", "query")
                await behaviour.send(store_b_req)

                # Чекаємо відповідь від магазина Б
                b_reply = await behaviour.receive(timeout=4)
                if b_reply and b_reply.body and b_reply.thread == thread_id:
                    try:
                        b_data = json.loads(b_reply.body)
                        if b_data.get("qty", 0) > 0:
                            b_data.setdefault("source", "storeb@localhost")
                            results.append(b_data)
                    except:
                        pass

            return results


        # ВНУТРІШНЯ ПОВЕДІНКА АГЕНТА (ОБРАБОТКА ЗАПИТІВ)

        class HandleRequest(CyclicBehaviour):

            def _safe_parse_response(self, ans):
                """ Проверяет ответ от склада на пустоту и битый JSON """
```

```
                if not ans or not ans.body:
                    return {"status": "rejected", "reason": "No response"}
                try:
                    return json.loads(ans.body)
                except json.JSONDecodeError:
                        return {"status": "rejected", "reason": "Invalid JSON
format"}


            # ФУНКЦИЯ 1  ГЛОБАЛЬНЫЙ ПОШУК

            async def handle_search(self, reply, offers, product_id):
                print(f"[Agent A] handle_search  Каталог для: {product_id}")
                if offers:
                    best = offers[0]
                    total_qty = sum(item["qty"] for item in offers)
                    reply.body = json.dumps({
                        "product": product_id,
                        "status": "found",
                        "best_price": best["price"],
                        "source": best["source"],
                        "display_qty": f"{best['qty']} / {total_qty}",
                        "qty": best["qty"]
                    })
                else:
                      reply.body = json.dumps({"status": "not_found", "product":
product_id})
                await self.send(reply)


            # ФУНКЦИЯ 2  ВЕРИФИКАЦІЯ ПЕРЕД ОФОРМЛЕННЯМ (verify)

                async def handle_verify(self, reply, offers, client_price,
client_qty):
                print(f"[Agent A] handle_verify  Мягкая проверка остатков")
                if offers:
                    best = offers[0]

                    price_match = (best["price"] == client_price)
                    qty_ok = (best["qty"] >= client_qty)
                    status = "ok" if (price_match and qty_ok) else "alert"
```

```
                    reply.body = json.dumps({
                        "status": status,
                        "best_price": best["price"],
                        "qty": best["qty"],
                        "source": best["source"]
                    })
                else:
                            reply.body  =  json.dumps({"status":  "not_found",
"best_price": 0, "qty": 0})
                await self.send(reply)


            # ФУНКЦИЯ 3  КАСКАДНЕ СПИСАННЯ (sell)

             async def handle_sell(self, reply, offers, product_id, client_qty,
thread_id):
                print(f"[Agent A] handle_sell  Каскад для {product_id}. Нужно:
{client_qty} шт.")

                successful_deals = []
                remaining_qty = client_qty

                for shop in offers:
                    if remaining_qty <= 0: break

                    take_qty = min(remaining_qty, shop["qty"])
                    sell_thread = f"{thread_id}-sell-{shop['source']}"

                       sell_req = Message(to=shop["source"], thread=sell_thread,
metadata={"performative": "request"})
                      sell_req.body = json.dumps({"action": "reduce", "product":
product_id, "qty": take_qty})
                    await self.send(sell_req)

                    # цикл на очікування выдповіді
                    ans = None
                    for _ in range(5):
                        potential_ans = await self.receive(timeout=1)
                                if  potential_ans  and  potential_ans.thread  ==
sell_thread:
                            ans = potential_ans
                            break
```

```
                        elif potential_ans:
                                                                    await
self.agent.HandleRequest().enqueue(potential_ans)

                    ans_data = self._safe_parse_response(ans)
                      is_success = (ans and ans.get_metadata("performative") in
["inform", "accept-proposal"] and ans_data.get("status") == "success")

                    successful_deals.append({
                            "source": shop["source"], "qty": take_qty, "price":
shop["price"],
                        "status": "confirmed" if is_success else "rejected",
                        "reason": "OK" if is_success else ans_data.get("reason",
"Timeout/Policy")
                    })

                    if is_success:
                        remaining_qty -= take_qty

                # формируем відповіді
                reply.body = json.dumps({"status": "processed", "final_success":
(remaining_qty == 0), "items": successful_deals})
                await self.send(reply)


            # ГОЛОВНИЙ ДИСПЕТЧЕР

            async def run(self):
                msg = await self.receive(timeout=10)
                if not msg:
                    return

                request_type = msg.get_metadata("request_type")

                try:
                    request_data = json.loads(msg.body)
                    product_id = request_data.get("product")
                    client_price = request_data.get("client_price")
                    client_qty = request_data.get("client_qty", 1)
                except:
                    product_id = msg.body
                    client_price = None
```

```
                    client_qty = 1

                      offers = await self.agent.get_sorted_offers(product_id,
msg.thread, behaviour=self)
                reply = msg.make_reply()

                match request_type:
                    case "search":
                        await self.handle_search(reply, offers, product_id)

                    case "verify":
                           await self.handle_verify(reply, offers, client_price,
client_qty)

                    case "sell":
                              await self.handle_sell(reply, offers, product_id,
client_qty, msg.thread)

                    case _:
                        print(f" Невідомий тип запиту: {request_type}")
```

Папка agents/ файл StoreBAgent.py

```
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour
from spade.message import Message
import json
import random

# Локальне сховище
STORE_B_DATA = {
    "milk": {"available": True, "quantity": 5, "price": 50},
    "oil": {"available": True, "quantity": 2, "price": 20},
}

class StoreBAgent(Agent):
    class HandleIncoming(CyclicBehaviour):
       async def run(self):
            msg = await self.receive(timeout=10)
            if not msg:
                return

            # ---  Парсинг вхідних даних ---
            performative = msg.get_metadata("performative")

            try:
                req_data = json.loads(msg.body)
                product = req_data.get("product")
                qty_requested = req_data.get("qty", 1)
            except Exception:
                product = msg.body
                qty_requested = 1

            reply = msg.make_reply()
            stock = STORE_B_DATA.get(product)

            if performative in ["propose", "request"]:

                rejection_chance = 0.20
                is_willing_to_sell = random.random() > rejection_chance
```

```
                if stock and stock["available"] and stock["quantity"] >=
qty_requested and is_willing_to_sell:
                    # Списання товару в магазина Б
                    stock["quantity"] -= qty_requested
                    if stock["quantity"] == 0:
                        stock["available"] = False

                    print(f"[Store B]  Угода підтверджена: {product}
(-{qty_requested} шт.) Осталось: {stock['quantity']} шт.")

                    reply.set_metadata("performative", "inform")
                    reply.body = json.dumps({"status": "success",
"confirmed_qty": qty_requested})

                    await self.send(reply)
                else:
                    if not is_willing_to_sell:
                        print(f"[Store B]  ВІДМОВА  - Внутрішня політика ")
                    else:
                        print(f"[Store B]  ВІДМОВА -  Нема необхіної
кількості товару {product}")

                    reply.set_metadata("performative", "inform")
                    reply.body = json.dumps({"status": "rejected", "reason":
"Out of stock"})
                    await self.send(reply)

            # ПОШУК ТОВАРУ
            else:
                if stock and stock["available"]:
                    reply.body = json.dumps({
                        "status": "confirmed",
                        "product": product,
                        "qty": stock["quantity"],
                        "quantity": stock["quantity"],
                        "price": stock["price"],
                        "source": str(self.agent.jid)
                    })
                else:
```

```
                    reply.body = json.dumps({"status": "rejected",
"product": product})

                reply.set_metadata("performative", "inform")
                await self.send(reply)

    async def setup(self):
        self.add_behaviour(self.HandleIncoming())
        print("Агент StoreB (Магазин Б) успішно запущено та готовий
відповідати.")
```

Папка agents/ файл WarehouseAgent.py

```
import asyncio
from spade.agent import Agent
from spade.behaviour import CyclicBehaviour
from spade.message import Message
import json

# Головний каталог
PRODUCT_CATALOG = {
    "milk": {"description": "Молоко 1.5%", "default_price": 50},
    "bread": {"description": "Хліб житній", "default_price": 20}
}

# Залишки та індивідуальні ціни постачальників
DATA_STORAGE = {
    "w1@localhost": {
        "milk": {"qty": 90, "price": 45},
        "bread": {"qty": 10, "price": 22}
    },
    "w2@localhost": {
        "milk": {"qty": 160, "price": 65},

    }
}

class WarehouseAgent(Agent):
    async def setup(self):
        self.my_data = DATA_STORAGE.get(str(self.jid), {})
```

```
        self.add_behaviour(self.HandleIncoming())

    class HandleIncoming(CyclicBehaviour):
        async def run(self):
            msg = await self.receive(timeout=10)
            if not msg:
                return

            # Зчитання метаданих
            performative = msg.get_metadata("performative")

            # СПИСАННЯ ТОВАРУ (request) ---
            if performative == "request":
                data = json.loads(msg.body)
                product_name = data.get("product")
                qty_to_reduce = data.get("qty", 1)

                info = self.agent.my_data.get(product_name)

                # перевірка наявності та кількості
                if info and info["qty"] >= qty_to_reduce:
                    info["qty"] -= qty_to_reduce
                    status = "success"
                    print(f"[{self.agent.jid}] СПИСАНО: {product_name}
({qty_to_reduce} шт.)")
                else:
                    status = "error_low_stock"

                reply = msg.make_reply()
                reply.set_metadata("performative", "inform")
                reply.body = json.dumps({"status": status, "product":
product_name})
                await self.send(reply)

            #ПОШУК (query)
            else:
                product_name = msg.body
                info = self.agent.my_data.get(product_name, {"qty": 0, "price":
0})
                catalog_info = PRODUCT_CATALOG.get(product_name, {"description":
"Невідомий товар"})
```

```
                if info["qty"] > 0:
                    response_body = {
                        "status": "available",
                        "product": product_name,
                        "qty": info["qty"],
                        "price": info["price"],
                        "description": catalog_info["description"],
                        "warehouse_id": str(self.agent.jid)
                    }
                else:
                    response_body = {"status": "not_found", "product":
product_name, "qty": 0, "price": 0}

                reply = msg.make_reply()

                reply.body = json.dumps(response_body)
                await self.send(reply)
```