МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 16.00.00.000 ПЗ

Група ІПм-24-3

Солнцев Олександр

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Солнцев Олександр Юрійович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та алгоритми інтеграції онтологій у веб-застосунки

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Солнцев О.Ю
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Яцишин Микола Миколайович, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри
доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль
доц. Вовк Р.Б.
(посада) (підпис) (дата) (ініціали та прізвище)

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

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

кафедра інженерії програмного забезпечення

Освітньо-кваліфікаційний рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Солнцеву Олександру Юрійовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи **“**Моделі, методи та алгоритми інтеграції онтологій у веб-застосунки**”**

керівник проекту (роботи) Яцишин Микола Миколайович, к.т.н., доцент

затверджені наказом закладу вищої освіти від **“** 09 ” листопада 2025 р. № 561/7

2. Строк подання студентом проекту (роботи) 15 грудня 2025 р.

3. Вихідні дані до проекту (роботи) Архітектура, формальний опис та алгоритми функціонування онтологій у веб-застосунках

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Теоретичні відомості про онтології та їх роль в інтеграції знань у веб-застосунках

2. Сучасні технології та методи інтеграції онтологій у веб-застосунки

3. Розробка алгоритму та програмної реалізації інтеграції онтології у веб-застосунок

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Приклад RDF графа, що ілюструє трійки ‘суб’єкт - предикат - об’єкт’ та їх інтерпретацію як орієнтованого графа знань. (рис. 1.3, ст. 20)

2. Відповідність елементів ER діаграми (сутності, атрибути, зв’язки) елементам OWL онтології (класи, дата та об’єктні властивості) на прикладі простої предметної області. (рис. 1.5, ст. 23)

3. Схема онлайнової синхронізації через Django signals. (рис. 2.6, ст. 45)

4. ER діаграма бази даних маркетплейсу зі сутностями (рис. 3.3, ст. 51)

5. Візуалізація онтології у вікні OntoGraf (рис. 3.16, ст. 59)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц. к.т.н. Вовк Р. Б.	

7. Дата видачі завдання 04 вересня 2025 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____ _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури	20.09.2025	виконано
2	Аналіз сучасних технологій та методи інтеграції онтологій у веб-застосунки	01.10.2025	виконано
3	Способи інтеграції онтологій у веб-застосунки	12.10.2025	виконано
4	Дослідження алгоритму інтеграції онтології у веб-застосунок	25.10.2025	виконано
5	Формулювання вимог та алгоритмів функціонування системи	05.11.2025	виконано
6	Програмна реалізація рішення	22.11.2025	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2025	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 95 с., 41 рис., 40 джерел.

Тема: Моделі, методи та алгоритми інтеграції онтологій у веб-застосунки.

Об'єкт дослідження: Моделі, методи та алгоритми інтеграції і використання онтологій для вдосконалення веб-застосунків

Мета роботи: Розробка та дослідження моделей і механізмів, що забезпечують ефективну інтеграцію онтологій у веб-застосунок, а також автоматичний обмін і синхронізацію даних між реляційною та графовою базами даних з використанням технологій Django, GraphDB, RDF/OWL, SPARQL.

Предмет дослідження:

Методи та програмні засоби інтеграції онтологій у веб-застосунок маркетплейсу для закладів харчування, включно з побудовою онтології, обміном даними між реляційною та графовою базами та використанням онтології для роботи рекомендаційних сервісів.

Результати дослідження:

Реалізовано онтологічну модель предметної області, механізми перенесення та синхронізації даних між реляційною та графовою БД і рекомендаційну підсистему, що показали ефективність інтеграції онтологій для покращення структуризації даних та інтелектуальних функцій веб-застосунку.

Висновок:

У результаті дослідження отримано програмне рішення, яке забезпечує ефективну інтеграцію і синхронізацію онтології з реляційною моделлю даних веб-застосунку.

ОНТОЛОГІЯ, СЕМАНТИЧНА МОДЕЛЬ ЗНАНЬ, ВЕБ-ЗАСТОСУНОК, ІНТЕГРАЦІЯ ДАНИХ, RDF/OWL, GRAPHDB, DJANGO, SPARQL, РЕКОМЕНДАЦІЙНА СИСТЕМА, МАРКЕТПЛЕЙС ЗАКЛАДІВ ХАРЧУВАННЯ

ANNOTATION

Master's work: 95 p., 41 fig., 40 sources.

Topic: Models, Methods and Algorithms for Integrating Ontologies into Web Applications.

Object of research: Models, methods and algorithms for integrating and using ontologies to improve web applications.

Purpose of the work: Development and study of models and mechanisms that provide effective integration of ontologies into a web application, as well as automatic data exchange and synchronization between relational and graph databases using Django, GraphDB, RDF/OWL and SPARQL technologies.

Subject of research: Methods and software tools for integrating ontologies into a marketplace web application for catering establishments, including ontology construction, data exchange between relational and graph databases, and the use of the ontology to support recommendation services.

Research results:

An ontological model of the subject area, mechanisms for data transfer and synchronization between the relational and graph databases, and a recommendation subsystem were implemented, demonstrating the effectiveness of ontology integration for improving data structuring and the intelligent functions of the web application.

Conclusion:

As a result of the research, a software solution was obtained that ensures effective integration and synchronization of an ontology with the relational data model of the web application.

ONTOLOGY, SEMANTIC KNOWLEDGE MODEL, WEB APPLICATION, DATA INTEGRATION, RDF/OWL, GRAPHDB, DJANGO, SPARQL, RECOMMENDER SYSTEM, FOOD-SERVICE MARKETPLACE

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ…….... 9
ВСТУП……....10

РОЗДІЛ 1
ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО ОНТОЛОГІЇ ТА ЇХ РОЛЬ В ІНТЕГРАЦІЇ ЗНАНЬ У ВЕБ-ЗАСТОСУНКАХ……... 15
1.1 Поняття онтологій та формальні основи семантичного представлення знань 15
1.2 Класифікація онтологій та моделі подання знань (RDF, RDFS, OWL) ……17
1.3 Онтологічний підхід до моделювання предметних областей у веб-системах…….22
1.4 Використання онтологій у веб-застосунках для підтримки інтелектуальних функцій……..28
1.5 Висновки до розділу……. 31

РОЗДІЛ 2
ТЕХНОЛОГІЇ ТА МЕТОДИ ІНТЕГРАЦІЇ ОНТОЛОГІЙ У ВЕБ ЗАСТОСУНКИ…….. 32
2.1 Технологічний стек Semantic Web та онтологій (Protégé, RDF-репозиторії, SPARQL, Triple Store)…… 32
2.2 Інтеграція онтологій з веб-фреймворками(на прикладі Django)…….35
2.3 Моделі та алгоритми перенесення даних з реляційних БД в онтології…… 39
2.4 Методи синхронізації даних між реляційною БД та онтологічним репозиторієм…… 42
2.5 Висновки до розділу…….. 46

РОЗДІЛ 3
РОЗРОБКА АЛГОРИТМУ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ІНТЕГРАЦІЇ ОНТОЛОГІЙ У ВЕБ-ЗАСТОСУНКИ ..…… 48

3.1 Аналіз предметної області(веб-застосунок маркетплейс-логістичного рішення для закладів харчування з інтеграцією онтології для реалізації рекомендаційної системи) ……..…. 48
3.2 Створення онтології в редакторі Protege …… 50
3.3 Реалізація функціоналу завантаження вже існуючих даних веб-застосунку в онтологію……..... 60
3.4 Реалізація синхронізації даних між веб-застосунком та онтологією……61
3.5 Налаштування онтології для виведення рекомендаці…….... 62
3.6 Тестування рекомендаційної системи…… 77
3.7 Висновки до розділу…….. 88

ВИСНОВКИ……. 90
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ……..... 91
ДОДАТКИ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

OWL - Web Ontology Language

RDF - Resource Description Framework

RDFS - RDF Schema

SPARQL - SPARQL Protocol and RDF Query Language

GraphDB - графова база даних для RDF/OWL

Django - веб-фреймворк Python

Protégé - редактор онтологій

API - прикладний програмний інтерфейс

Triple Store - сховище RDF-трійок

MVC - Model-View-Controller

MVT - Model-View-Template

БД - база даних

ПЗ - програмне забезпечення

URI - уніфікований ідентифікатор ресурсу

Semantic Web - семантична павутина

Object Property - об'єктна властивість

Data Property - властивість даних

Individuals - індивіди (екземпляри)

Matching - зіставлення онтологій

Alignment - узгодження онтологій

ВСТУП

Актуальність роботи

Стрімкий розвиток веб-технологій та зростання обсягів даних у сучасних інформаційних системах призводять до ускладнення структури даних, збільшення кількості взаємопов’язаних сутностей та необхідності інтеграції різнорідних джерел інформації. У класичних веб-застосунках, побудованих на реляційних базах даних, виникають труднощі з гнучким розширенням моделі даних, повторним використанням знань, а також з реалізацією інтелектуальних функцій, що потребують розуміння семантики предметної області. Особливо гостро ці проблеми проявляються у складних доменах, таких як маркетплейси з логістичними процесами, де одночасно взаємодіють клієнти, заклади харчування, кур’єри, кухарі та керівники закладів, а дані розподілені між різними підсистемами.

Онтологічні моделі та технології Semantic Web дозволяють формалізувати знання про предметну область, забезпечити єдину семантичну модель, підтримати інтероперабельність між компонентами системи та створити основу для побудови рекомендаційних сервісів і семантичного аналізу. Проте інтеграція онтологій у реальні веб-застосунки, побудовані на популярних фреймворках, потребує розробки спеціальних моделей, методів та алгоритмів синхронізації даних між реляційними та графовими сховищами, а також організації взаємодії між прикладною логікою і онтологічним репозиторієм. Це зумовлює актуальність дослідження моделей, методів та алгоритмів інтеграції онтологій у веб-застосунки на прикладі маркетплейсу – логістичного рішення для закладів харчування.

Порівняння роботи з відомими розв’язаннями проблеми

У сучасних веб-системах для закладів харчування та служб доставки їжі широко використовуються платформи, які реалізують функціонал замовлення, оплати, відстеження статусів та логістики. Більшість таких рішень базуються на класичних реляційних моделях даних і спрямовані передусім на операційні процеси,

тоді як рівень формалізації знань про предметну область обмежується структурою таблиць і бізнес-логікою застосунку.

У наукових працях активно досліджуються питання побудови онтологій, використання RDF/OWL, семантичного пошуку та онтологічних моделей для підтримки рекомендуючих систем. Водночас практичних рішень, де онтологія інтегрована безпосередньо в архітектуру реального веб-застосунку з реляційною БД, механізмами міграції та синхронізації даних, а також використовується як основа для рекомендацій у конкретній предметній області (маркетплейс закладів харчування), значно менше.

Багато існуючих підходів або зосереджуються на проєктуванні онтології без глибокої інтеграції з прикладним кодом, або використовують онтологію як окремий модуль без повноцінної двосторонньої синхронізації з реляційною БД. Це створює розрив між теоретичними моделями та реальними веб-додатками. Тому виникає потреба у розробці підходу, який поєднує класичний веб-фреймворк (Django), графовий репозиторій (GraphDB), онтологію предметної області та механізми синхронізації даних і використання онтології в рекомендаційній підсистемі.

Мета і задачі дослідження

Метою магістерської роботи є розробка та дослідження моделей і механізмів, що забезпечують ефективну інтеграцію онтологій у веб-застосунок, а також автоматичний обмін і синхронізацію даних між реляційною та графовою базами даних з використанням технологій Django, GraphDB, RDF/OWL, SPARQL. Досліджувана модель повинна забезпечувати:

– відображення структури реляційної БД веб-застосунку у вигляді онтології предметної області;

– перенесення наявних даних у графове сховище з використанням формальних онтологічних конструкцій;

– синхронізацію змін даних у режимі, наближеному до реального часу;

– можливість використання семантичних зв’язків в онтології для формування рекомендацій користувачам маркетплейсу.

Досягнення мети включало розв’язання таких **задач:**

1) виконати аналіз предметної області «Маркетплейс – логістичне рішення для закладів харчування» з виділенням основних сутностей, ролей та процесів;

2) дослідити теоретичні основи онтологій, моделі подання знань (RDF, RDFS, OWL) та технології Semantic Web, релевантні до інтеграції у веб-застосунки;

3) розробити онтологічну модель предметної області на основі існуючої реляційної моделі БД веб-додатку;

4) спроєктувати та реалізувати алгоритм початкового перенесення даних із реляційної бази даних Django у репозиторій GraphDB з використанням RDF/OWL та SPARQL;

5) розробити механізм синхронізації даних між реляційною БД та онтологічним репозиторієм у режимі реального часу з використанням інструментів фреймворку Django (signals);

6) побудувати та інтегрувати рекомендаційну підсистему, що використовує дані онтології та поведінкову інформацію користувачів для формування персоналізованих рекомендацій;

7) оцінити вплив інтеграції онтології на можливості структуризації даних та реалізацію інтелектуальних функцій веб-застосунку.

Об’єкт дослідження

Об’єктом дослідження є моделі, методи та алгоритми інтеграції і використання онтологій для вдосконалення веб-застосунків.

Предмет дослідження

Предметом дослідження є методи та програмні засоби інтеграції онтологій у веб-застосунок маркетплейсу для закладів харчування, включно з побудовою онтології, обміном даними між реляційною та графовою базами та використанням онтології для роботи рекомендаційних сервісів.

Методи дослідження

Для досягнення поставленої мети використовувалися методи аналізу та узагальнення наукових джерел у галузі онтологічного моделювання та Semantic Web, методи порівняння існуючих архітектур веб-систем та підходів до інтеграції онтологій, системний підхід до проєктування архітектури веб-застосунку з комбінованою (реляційною та графовою) моделлю даних, методи формального моделювання предметної області у вигляді онтології, методи експериментального дослідження роботи реалізованих алгоритмів перенесення та синхронізації даних, а також методи емпіричної перевірки коректності функціонування рекомендаційної підсистеми.

Наукова новизна одержаних результатів

Наукова новизна роботи полягає у розробці та апробації комплексного підходу до інтеграції онтології предметної області у веб-застосунок маркетплейсу на основі фреймворку Django та репозиторію GraphDB, який включає:
– модель відображення реляційної структури даних у онтологічну модель з урахуванням специфіки предметної області закладів харчування;
– алгоритм початкового перенесення даних із реляційної БД у онтологічне сховище з контролем цілісності та унікальності тріплетів;

– механізм синхронізації даних у режимі реального часу на основі подієвих механізмів веб-фреймворку та SPARQL-запитів;

– використання інтегрованої онтології як основи для побудови рекомендаційної підсистеми, що поєднує семантичну інформацію та поведінкові дані користувачів.

Практичне значення одержаних результатів

Практичне значення роботи полягає у створенні діючого програмного рішення, яке реалізує інтеграцію онтології з веб-застосунком маркетплейсу для закладів харчування, побудованим на Django, та репозиторієм GraphDB. Розроблені моделі, методи й алгоритми можуть бути використані для модернізації існуючих веб-

систем шляхом додавання семантичного рівня представлення даних, впровадження рекомендаційних сервісів і поліпшення можливостей аналізу даних. Отримані результати можуть бути адаптовані до інших предметних областей, де необхідна інтеграція реляційних БД з онтологічними репозиторіями та підтримка інтелектуальних функцій у веб-додатках.

Структура магістерської роботи.

Магістерська робота викладена на 95 сторінках друкованого тексту, який складається з вступу, трьох розділів, висновків, списку використаних джерел (40 найменувань). Робота містить 41 рисунок та 1 додаток.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО ОНТОЛОГІЇ ТА ЇХ РОЛЬ В ІНТЕГРАЦІЇ ЗНАНЬ У ВЕБ-ЗАСТОСУНКАХ

1.1 Поняття онтологій та формальні основи семантичного представлення знань

Онтологія походить з філософії, де вона стосується вивчення буття та категорій реальності. В інформатиці ця ідея була адаптована для позначення формального способу опису знань про певну предметну область, які можуть обробляти машини. Одне поширене визначення - ‘явна формальна специфікація концептуалізації’. Це означає, що онтологія дає формальний опис узгодженого набору понять та зв'язків між ними. Вона допомагає спільноті розробників або систем інтерпретувати дані однаково. Таким чином, онтологія є важливою для представлення знань, інтеграції даних та створення семантичних веб-систем.

Онтологія має кілька основних елементів: класи (концепції), властивості (зв'язки) та особи (конкретні об'єкти). Класи визначають узагальнені типи сутностей у предметній області. Наприклад, у сервісах електронної комерції класи можуть включати Користувача, Продукт та Замовлення. Властивості класифікуються на властивості об'єктів (зв'язки між особами, такі як ‘користувач зробив замовлення’) та властивості даних (зв'язки особи з літеральними значеннями, такими як рядки, числа та дати). Особи - це конкретні екземпляри класів, що представляють реальні об'єкти, такі як певний користувач або конкретне замовлення. Колекція класів, властивостей та індивідів, описаних за допомогою формальних методів (таких як RDF та OWL), утворює граф знань, де можуть виконуватися логічні операції. Для складніших систем онтологія також може включати ієрархії класів, типізацію властивостей, обмеження значень та структури, що розширює виразні можливості моделі. Більшість сучасних мов онтологій, особливо OWL, базуються на логіках опису (DL). Вони дозволяють нам описувати класи та властивості за допомогою логічних конструкцій та аксіом, гарантуючи, що основні проблеми логічного виводу все ще можуть бути вирішені

В описових логіках база знань зазвичай поділяється на дві частини: TBox (термінологічний компонент) та ABox (стверджувальний компонент). TBox визначає специфікації класів, ієрархії та властивості, тоді як ABox містить твердження про конкретних індивідів. Аксіоми, такі як subClassOf, обмеження домену та діапазону для властивостей, несумісність класів та кардинальні обмеження, встановлюють значення онтології та формують основу для автоматичного логічного виводу. Завдяки цьому система може не тільки зберігати явно викладені факти, але й виводити нові. Наприклад, він може автоматично класифікувати об'єкт як частину підкласу, виявляти конфлікти між операторами або заповнювати відсутні зв'язки, що випливають з аксіом.

Застосування онтологій у веб-додатках виникає з необхідності переходу від суто структурного представлення даних, такого як таблиці, поля та ключі, до семантичного представлення, де значення понять та їхні зв'язки чітко визначені. Хоча реляційна база даних підходить для обробки транзакцій, онтологія пропонує єдину концептуальну модель знань, яку різні системні компоненти та зовнішні служби можуть використовувати як спільну мову для інтеграції та аналізу даних. Завдяки формальній семантиці та підтримці логічного висновку, онтологія допомагає виявляти приховані зв'язки, підтримувати узгодженість даних та служити основою для інтелектуальних функцій, таких як семантичний пошук, узагальнення запитів та генерація рекомендацій. У складних розподілених системах цей підхід спрощує інтеграцію різних джерел інформації, дозволяючи створювати єдиний граф знань та додавати інтелектуальні модулі без суттєвої зміни вихідної структури даних.

Крім того, онтології підтримують повторне використання та розширення моделей знань. Окремі онтології предметної області можуть успадковувати або імпортувати частини ширших онтологій верхнього рівня, зберігаючи термінологію узгодженою та уникаючи надлишкових визначень. Це дозволяє створювати багаторівневі системи знань, де локальні моделі (наприклад, для певної галузі чи послуги) вписуються в ширший семантичний контекст. Це особливо корисно для інтеграції різних веб-сервісів та платформ в єдине інформаційне середовище. Рис. 1.1 ілюструє роль онтологію у багаторівневій архітектурі семантичної мережі. Словник

онтологій та рівень RDF/RDFS розташовані над фундаментальними технологіями для ідентифікації та структурування даних (таких як URI та XML), але нижче рівнів логіки, правил та довіри, утворюючи міцну основу для розробки інтелектуальних, семантично усвідомлених застосунків.

На зображенні 1.1 можна побачити, що онтологія є ключовим елементом семантичного рівня між ‘сирими’ даними та інтелектуальними функціями системи.

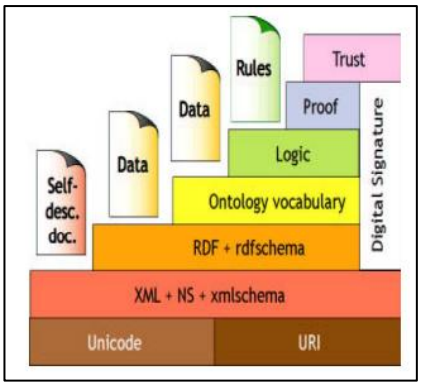

Рис. 1.1. Узагальнена шарова модель технологій Semantic Web, де онтології (Ontology vocabulary) та RDF/RDFS виступають семантичним рівнем над структурою даних (XML/URI) і підґрунтям для логіки, правил і довіри

1.2 Класифікація онтологій та моделі подання знань (RDF, RDFS, OWL)

1.2.1 Типи онтологій за призначенням

У сучасній інженерії знань ми визначаємо кілька основних типів онтологій на основі їхнього рівня абстракції, охоплення та призначення. Ця класифікація допомагає у виборі відповідного підходу до моделювання предметної області. Вона допомагає уникнути надмірної деталізації, коли достатньо загальних концепцій. З іншого боку, вона також гарантує, що ми забезпечуємо достатню глибину для спеціалізованих областей.

Найбільш абстрактний рівень складається із загальних онтологій. Ці фундаментальні онтології описують основні категорії реальності, що зустрічаються в багатьох предметних областях. Вони включають об'єкти, події, процеси, часові та просторові відносини, ролі, властивості, зв'язки частина-ціле та причинно-наслідкові

зв'язки. Ці онтології не прив'язані до певної галузі. Натомість вони надають універсальний словник та набір структур для розробки більш спеціалізованих моделей. Використання спільної онтології як верхнього рівня забезпечує сумісність між різними онтологіями предметних областей, полегшуючи інтеграцію даних з різних систем.

Онтології предметних областей зосереджені на знаннях у певній предметній області, враховуючи її специфіку та термінологію. В охороні здоров'я ключовими поняттями можуть бути ‘пацієнт’, ‘діагноз’, ‘рецепт’ та ‘ліки’. В освіті вони включатимуть ‘курс’, ‘студент’, ‘навчальна програма’ та ‘оцінювання’. Для електронної комерції ключовими термінами можуть бути ‘продукт’, ‘категорія’, ‘кошик’ та ‘замовлення’. Онтологія предметної області фіксує те, як фахівці галузі розуміють об'єкти та процеси. Вона служить основою для створення інформаційних систем, що працюють з однаковою термінологією. Цей тип онтології актуальний для громадського харчування, логістики замовлень та взаємодії користувачів на веб-маркетплейсу.

Ще більш цілеспрямованим типом є онтології додатків або завдань. Вони призначені для вирішення певного класу проблем в рамках онтології предметної області. У цьому випадку фокус включає не лише структуру предметної області, але й інформацію, необхідну для вирішення конкретної проблеми. Це може включати моделювання профілів користувачів, параметрів якості обслуговування та типових сценаріїв взаємодії. Для маркетплейсу харчування онтологія додатків може розширити онтологію предметної області, додавши такі поняття, як ‘інтерес користувача до страви’, ‘схожість страв’ та ‘історія взаємодії’. Ці елементи мають вирішальне значення для створення персоналізованих рекомендацій.

На практиці ми часто поєднуємо ці типи онтологій. Загальна онтологія визначає основні категорії. Онтологія предметної області уточнює специфіку галузі. Онтологія прикладної області вводить додаткові конструкції, що підтримують окремі функції інформаційної системи. Така багаторівнева організація дозволяє повторно використовувати існуючі моделі, підтримує узгоджену термінологію та допомагає інтегрувати системи в різних пов'язаних областях.

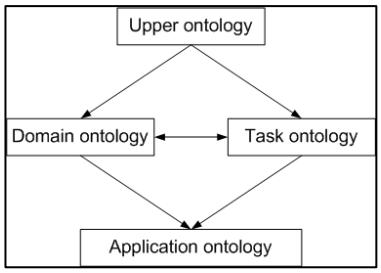

Рис. 1.2. Приклад поділу онтологій на загальні (upper-level), доменні, задачні та прикладні рівні

1.2.2 RDF та RDFS як базова модель представлення знань

Структура опису ресурсів, або RDF, - це стандартна модель представлення даних у семантичній мережі. Вона описує ресурси таким чином, щоб їх могли зчитувати машини. На відміну від реляційних таблиць, де знання організовані у фіксовані схеми рядків і стовпців, RDF використовує прості оператори, які називаються трійками суб'єкт-предикат-об'єкт. Суб'єкт і предикат зазвичай є ресурсами, ідентифікованими унікальними URI, тоді як об'єкт може бути іншим ресурсом або літерним значенням, таким як число, рядок або дата.

Графова структура RDF робить його особливо корисним для інтеграції даних з різних джерел. Якщо два набори трійок використовують один і той самий URI для тих самих сутностей, їхні графіки можна легко об'єднати без зміни схеми. Наприклад, оператор ‘користувач розмістив замовлення’ може бути виражений як трійка. Тут суб'єкт - це URI певного користувача, предикат - ‘розмістив замовлення’, а об'єкт - URI певного замовлення. Якщо додаються трійки приблизно однакового порядку, такі як ‘порядок включає страву X’ та ‘порядок має кількість N’, вони автоматично створюють один зв'язний граф.

Однак чистий RDF не пояснює, які ресурси є класами або як вони пов'язані один з одним. Йому також бракує механізмів для визначення ієрархій або обмежень на властивості. Для цієї мети використовується схема RDF, або RDFS. Вона розширює RDF, вводячи основні терміни для опису онтологічної структури: rdfs:Class для оголошення класів, rdfs:subClassOf для відображення зв'язків підкласів, а також rdfs:domain та rdfs:range для визначення домену та діапазону властивостей. Ця схема

дозволяє нам вказати, що властивість застосовується лише до певних типів ресурсів та має значення певного класу.

Наприклад, якщо ми вказуємо в RDFS, що властивість hasOrder має домен User та діапазон Order, то будь-який ресурс, який є суб'єктом трійки з цим предикатом, розуміється як користувач, а будь-який ресурс в об'єкті інтерпретується як порядок. Це дозволяє системі робити логічні висновки про типи ресурсів, навіть якщо їхні типи чітко не визначені. Поєднання RDF та RDFS забезпечує базовий семантичний рівень: дані відображаються у вигляді графа, тоді як прості онтологічні зв'язки, такі як класи, ієрархії, домени та діапазони, додають додаткового значення.

У практичних застосуваннях дані RDF зберігаються у спеціалізованих сховищах RDF, відомих як потрійні сховища, які призначені для обробки великої кількості потрійних сховищ та виконання запитів мовою SPARQL. Ця можливість дозволяє використовувати RDF та RDFS як універсальний формат для інтеграції, обміну та аналізу даних між різними системами, незалежно від їхніх внутрішніх баз даних.

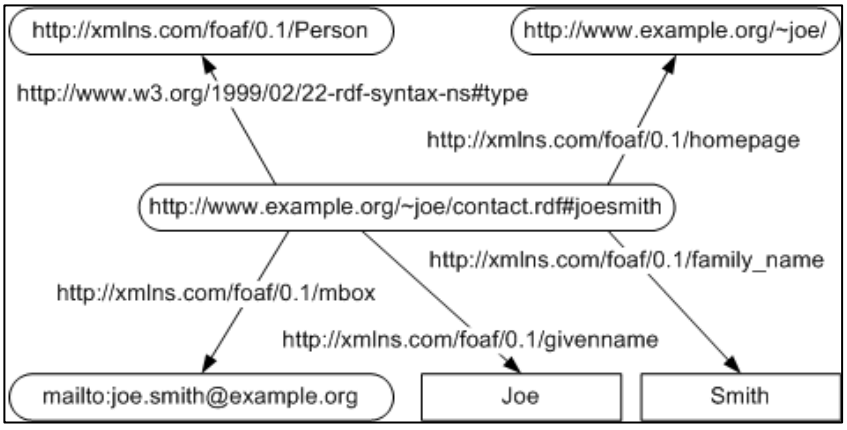

Рис. 1.3. Приклад RDF-графа, що ілюструє трійки ‘суб’єкт - предикат - об’єкт’ та їх інтерпретацію як орієнтованого графа знань

1.2.3 OWL як мова онтологій поверх RDF(S)

Хоча RDF та RDFS забезпечують корисний семантичний рівень, їм бракує виразності, необхідної для складних завдань представлення знань. Зокрема, RDFS не дозволяє формально заявити, що два класи не можуть перетинатися, встановити обмеження на кількість елементів, пов'язаних певною властивістю, або визначити

складні класи як комбінації інших класів. Для вирішення цих проблем була розроблена OWL (Web Ontology Language). OWL розширює RDF/S та використовує математичну основу описової логіки.

OWL пропонує розробникам різноманітні інструменти для точного визначення значення класів та властивостей. Поряд з простим успадкуванням (subClassOf), він дозволяє вказати, що два класи є еквівалентними, несумісними, або визначити клас як перетин, об'єднання або доповнення інших класів. Для властивостей він може визначити їх як функціональні (кожен індивід має максимум одне значення для цієї властивості), інверсні, транзитивні або симетричні. Він також підтримує обмеження кардинальності, які встановлюють мінімальну, максимальну або точну кількість зв'язків певного типу, які повинен мати індивід.

В результаті, онтологія на рівні OWL може включати не лише список класів та властивостей, але й багатий набір аксіом, що окреслюють дозволені та заборонені комбінації, приховані зв'язки та інваріанти домену. Це дозволяє використовувати методи міркування, які є спеціальними інструментами логічного висновку, що автоматично перевіряють узгодженість в онтології, виводять ієрархії класів, визначають класи, до яких належать індивіди, та виявляють неочевидні факти, що виникають з аксіом.

Враховуючи різні потреби в продуктивності та складності, існують профілі OWL (наприклад, OWL 2 EL, OWL 2 QL, OWL 2 RL). Кожен профіль обмежує доступні конструкції, але спрощує реалізацію висновку та масштабування до великих наборів даних. Це створює баланс між виразністю та ефективністю: для компактних онтологій з багатою логікою можна використовувати повну OWL DL, тоді як для великих прикладних систем, орієнтованих на розгалужені RDF-графи, краще вибрати один з профілів, розроблених для покращення продуктивності.

Таким чином, OWL слугує потужнішим шаром поверх RDF/RDFS. Він дозволяє створювати онтології, які описують не лише структуру даних, але й визначають глибокий зміст предметної області та підтримують автоматизований логічний висновок. Це робить OWL вирішальним інструментом у розробці інтелектуальних веб-систем, що спираються на знання, а не лише на необроблені дані.

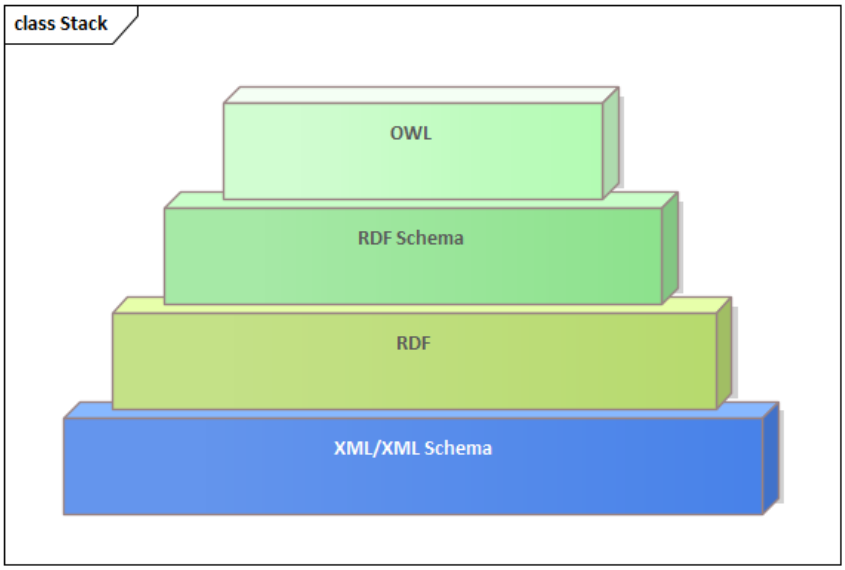

Рис. 1.4. Стек технологій RDF, RDFS та OWL, де OWL виступає мовою онтологій поверх моделі трійок та схемних засобів RDFS

1.3 Онтологічний підхід до моделювання предметних областей у веб-системах

1.3.1 Принципи моделювання онтологій

Опис домену у веб-системі за допомогою онтології передбачає створення формальної моделі знань. Ця модель показує основні сутності, їхні типи зв'язків, обмеження та ролі користувачів за допомогою класів, властивостей та аксіом. На відміну від неформальних текстових описів, цей метод робить знання зчитуваними машинами. Веб-додатки, зовнішні сервіси, інструменти аналізу даних та модулі міркувань можуть інтерпретувати їх однаково. Ця узгодженість особливо важлива для розподілених веб-систем, де незалежні команди розробляють різні компоненти. Спільне розуміння домену має бути зафіксовано в загальній моделі.

У цьому контексті онтологія фіксує компактне, але змістовне ядро знань про домен:

- Ключові сутності (наприклад, користувач, заклад, замовлення, страва)

- Зв'язки між ними (хто що замовляє, хто що володіє, хто що готує або доставляє)

- Основні обмеження (допустимі статуси замовлення, кардинальність зв'язків, сумісність ролей)

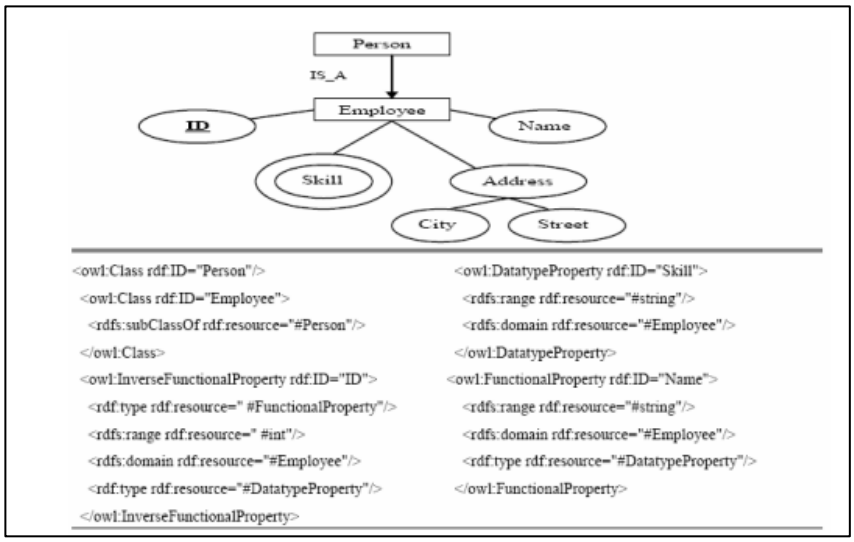

Рис. 1.5. Відповідність елементів ER-діаграми (сутності, атрибути, зв’язки) елементам OWL-онтології (класи, дата- та об’єктні властивості) на прикладі простої предметної області

Деякі з цих обмежень у традиційних системах вбудовані в код бізнес-логіки або тригери бази даних. На відміну від цього, онтологія дозволяє підняти їх до рівня моделі, роблячи їх чіткими та перевіреними логічними висновками.

На відміну від ER-діаграм, які зосереджені на структурі таблиць, ключів та типів полів, онтологічне моделювання зосереджується на семантиці, логічній узгодженості та потенціалі повторного використання моделі в інших системах. ER-модель часто тісно пов'язана з певною СУБД та її реалізацією. З іншого боку, онтологія сприяє обміну знаннями. Її можна опублікувати як окремий ресурс, розширити іншими онтологіями та підключити до відкритих даних. Крім того, онтології в OWL підтримують опис аксіом (наприклад, еквівалентність класів, непересічна непересічна нерівність, частковий порядок), що дозволяє автоматично генерувати нові факти. Однак ER-модель обмежена статичною структурою.1.3.2. Побудова онтологічної моделі веб-додатку

Типовий процес побудови онтологічної моделі веб-додатку починається з аналізу вимог і виділення ключових понять предметної області, які групуються в класи та підкласи. Далі формується ієрархія класів, задаються об’єктні й дата-властивості, визначаються основні обмеження (кардинальності, домени, коди), після чого модель наповнюється індивідами, що представляють конкретні екземпляри об’єктів із реального світу.

Для підтримки цього процесу широко застосовуються інструменти на кшталт Protégé, які надають візуальне редагування OWL-онтологій, інтегровані reasoner’и для перевірки узгодженості та експорт у стандартні формати. Репозиторії RDF-даних, зокрема GraphDB, виступають середовищем зберігання й виконання SPARQL-запитів до онтології, що дозволяє інтегрувати семантичну модель безпосередньо в роботу веб-додатку як джерело знань і правил.

1.3.2 Побудова онтологічної моделі веб-застосунку

Типовий процес побудови онтологічної моделі веб-застосунку починається з аналізу вимог та вибору ключових концепцій з предметної області. Ці концепції групуються в класи та підкласи. На цьому етапі важливо розрізняти ‘словник користувача’ (як його називають клієнти та експерти) від технічних термінів, що використовуються розробниками. Це допомагає узгодити назви та значення основних концепцій. Далі створюється ієрархія класів. Встановлюються властивості об'єктів та даних, а також визначаються основні обмеження, такі як потужності, домени та коди. Потім модель заповнюється індивідами, які представляють конкретні екземпляри об'єктів з реального світу або існуючих баз даних. На заключних етапах онтологія перевіряється на відповідність тестовим сценаріям. Перевіряється, чи правильно вона охоплює типові бізнес-кейси, і гарантується, що аксіоми не суперечать одна одній.

Для підтримки цього процесу зазвичай використовуються такі інструменти, як Protégé. Вони забезпечують візуальне редагування онтологій OWL, вбудовані міркування для перевірки узгодженості та підтримують експорт у стандартні формати RDF/OWL. Protégé також дозволяє поєднувати графічне моделювання з формальними описами. Користувачі можуть одразу побачити вплив доданих аксіом, таких як

суперкласи або виявлені суперечності. Репозиторії даних RDF, особливо GraphDB, служать середовищами зберігання та виконання для запитів SPARQL до онтології. Вони підтримують роботу з мільйонами трійок, індексацію та міркування на рівні зберігання. Така конфігурація дозволяє інтегрувати семантичну модель безпосередньо в роботу веб-додатку, виступаючи джерелом знань та правил, а не просто окремим документом дизайну.

У реальному проекті процес побудови онтології часто включає ітерації. Початкова приблизна модель створюється на основі першої версії вимог. Потім, під час розробки веб-додатку, вона доповнюється деталями та новими класами. Наприклад, спочатку можна моделювати базові класи "Користувач", "Заклад" та "Замовлення". Пізніше додаються класи для програм лояльності, акцій та складних типів доставки. Такий підхід допомагає уникнути надмірного проектування на початку та поступово узгоджує онтологію з фактичною поведінкою системи.

1.3.3 Онтологічна модель предметної галузі "Маркетплейс - логістичне рішення для закладів харчування"

На базовому рівні онтологічна модель маркетплейсу для закладів харчування включає кілька сутностей: користувачів (клієнтів, власників закладів, кур'єрів, адміністраторів), заклади, меню, страви, замовлення, різні типи взаємодій (перегляди, додавання до кошика, оформлення замовлення) та рейтинги. Кожну групу можна додатково розділити на підкласи. Наприклад, 'Заклад' поділяється на 'Ресторан', 'Кафе', 'Служба доставки готових наборів', тоді як 'Користувач' поділяється на 'Нового клієнта', 'Постійного клієнта' та 'VIP-клієнта'. Зв'язки між цими сутностями показують ключові процеси: клієнт розміщує замовлення в певному закладі, замовлення містить страви з меню цього закладу, кур'єр доставляє замовлення, а відгуки та рейтинги відображають результат. На онтологічному рівні ці зв'язки відображаються властивостями об'єктів, які допомагають створювати складні запити, такі як 'всі заклади, які регулярно отримують високі оцінки від постійних клієнтів у певному районі'.

Ця модель слугує практичною основою для веб-застосунку, оскільки вона легко підтримує різні ролі та сценарії: дії клієнтів, управління закладами та меню, планування та контроль логістики, а також відстеження історії взаємодії. Завдяки формальній структурі онтології система може легко відповідати новим вимогам. Нову роль, таку як ‘оператор підтримки’ або ‘партнер по франшизі’, можна додати, не порушуючи основну структуру знань. Зв'язки та обмеження між сутностями вже окреслені в семантичному графі знань, який легко аналізувати та розширювати. Крім того, онтологічна модель дозволяє вводити додаткові ‘приховані’ концепції, корисні для аналітики та рекомендацій, такі як ‘Шаблони замовлень’, ‘Сегмент клієнтів’ та ‘Профіль закладу’. Ці концепції не відображаються в окремих таблицях реляційної бази даних, а описують колективні знання, що походять з історії взаємодій та відгуків користувачів. За допомогою цих абстракцій маркетплейс може відображати поточний стан системи, а також робити ширші висновки про поведінку аудиторії та якість послуг закладів.

1.3.4 Відповідність між онтологічною моделлю та архітектурою веб-системи

Онтологічна модель природно ‘вписується’ в трирівневу архітектуру веб-системи торговельного майданчика. На рівні даних об'єднані реляційна база даних (для транзакційних операцій із замовленнями, платежами, сеансами) та GraphDB як сховище даних RDF, яке зберігає онтологію, осіб та їхні зв'язки у вигляді графа знань. Транзакційні таблиці забезпечують швидку обробку змінних даних (нові замовлення, оновлення статусу, журнали сеансів), тоді як граф знань фіксує більш стабільну структуру предметної області та похідні зв'язки, що використовуються для аналітики та рекомендацій. Це дозволяє розділити операційне та ‘семантичне’ робоче навантаження без шкоди для продуктивності.

На логічному рівні фреймворк Django реалізує бізнес-процеси, координує доступ до обох репозиторіїв та генерує як SQL, так і SPARQL-запити залежно від характеру завдання (операційна обробка чи семантичні запити). Наприклад, підтвердження замовлення обробляється переважно через реляційну базу даних, а

побудова списку ‘схожих страв’ або ‘місць, які можуть зацікавити користувача’ спирається на запити SPARQL до онтології. Окремий сервіс або модуль у застосунку може відповідати за синхронізацію: відображення змін у реляційних таблицях на рівні RDF, щоб модель знань залишалася актуальною.

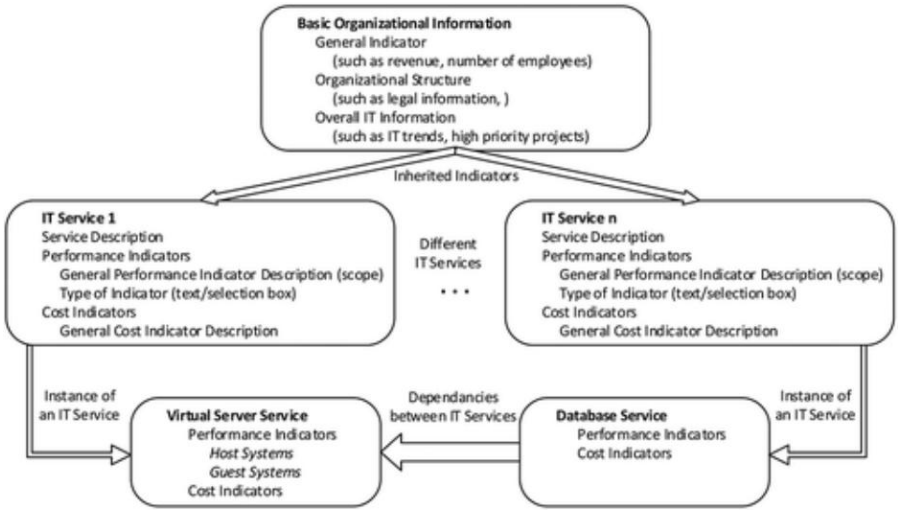

Рис. 1.6. Узагальнена схема організації доменних сервісів у багаторівневій архітектурі: верхній рівень - загальна інформація та показники, середній - окремі сервіси, нижній - їхні інстанси та залежності

На рівні представлення інтерфейси для різних ролей користувачів (клієнти, власники закладів, кур'єри, адміністратори) використовують дані, отримані з обох рівнів зберігання: оперативну інформацію з реляційної бази даних та узагальнені, семантично збагачені результати запитів до онтології. Для клієнта це можуть бути персоналізовані добірки закладів, динамічні фільтри за типом кухні, режимом роботи, рейтингами; для власника закладу - аналітичні панелі, що показують сегменти аудиторії, популярні страви та часові закономірності замовлень. У такій архітектурі онтологія відіграє роль не лише документованої концептуальної моделі, але й активного компонента, який впливає на логіку системи, забезпечує узгодженість даних між сервісами, підвищує сумісність та забезпечує основу для впровадження інтелектуальних сервісів на основі знань.

1.4 Використання онтологій у веб-застосунках для підтримки інтелектуальних функцій

Онтології у веб-застосунках дозволяють представляти дані у вигляді пов’язаного графа знань, де явно задані класи об’єктів, їхні властивості та відношення. Це створює основу для реалізації різних ‘інтелектуальних’ можливостей: семантичного пошуку, персоналізації, рекомендацій, аналізу знань, оскільки системі доступна не лише структура даних, а й описаний зміст. У загальному випадку онтологія може використовуватися як спільний семантичний шар для кількох компонентів веб-платформи, які звертаються до неї через запити до сховища RDF-даних та сервіси бізнес-логіки.

1.4.1. Онтології у семантичному пошуку та навігації

У семантичному пошуку онтологія відіграє роль формалізованого словника предметної області: поняття організовані в ієрархії, між ними визначені відношення типу ‘є підтипом’, ‘частина’, ‘пов’язане з’, можуть бути задані синоніми та варіанти термінів. Завдяки цьому запит користувача може бути перетворено з текстового рядка на структурований запит до графа знань, що дозволяє враховувати більш широкий контекст, ніж простий пошук за ключовими словами. Наприклад, в системі для закладів харчування пошук за назвою кухні може автоматично охоплювати всі страви та меню, пов’язані з цією кухнею в онтології, навіть якщо у їхніх текстових описах ця назва не повторюється явно.

У типових веб-маркетплейсах онтологія може описувати категорії страв, інгредієнти, дієтичні обмеження, типи закладів, режими роботи тощо, а також зв’язки між цими поняттями. Це дає можливість реалізувати пошук та навігацію, де користувач зміщує фокус не лише між окремими сторінками, а й між семантично пов’язаними категоріями: від загальної кухні до підкатегорій страв, від певного інгредієнта до всіх позицій, що його містять, від типу закладу до переліку відповідних пропозицій.

Такий підхід покращує знаходження релевантного контенту та зменшує залежність від точного формулювання запиту.

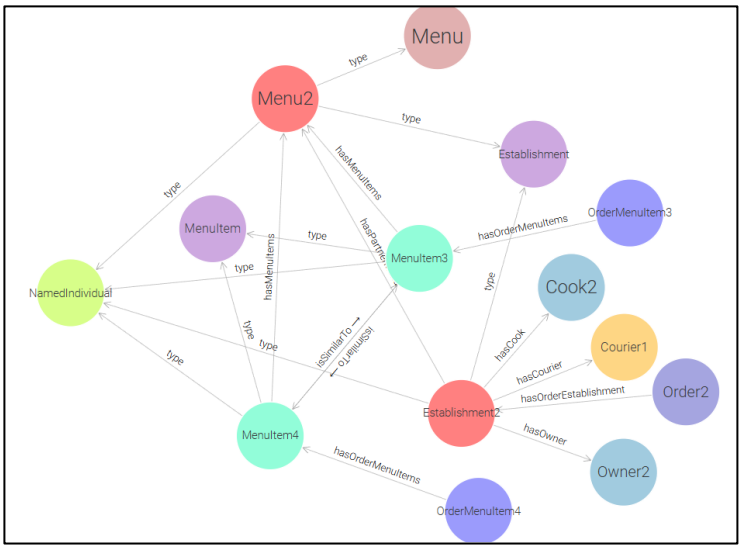

Рис. 1.7. Фрагмент онтологічної моделі предметної області (зв'язки між кухнею, стравами та закладами), який може бути використаний як семантична основа для пошуку та навігації в веб-застосунку

1.4.2. Онтології у персоналізації та моделюванні користувачів

У персоналізованих веб-застосунках онтологія може використовуватися для моделювання профілю користувача як частини графа знань. Профіль у цьому випадку включає не лише базову інформацію (ідентифікатор, контактні дані), а й пов'язані сутності: ролі, типові дії, вподобання, обмеження, історію взаємодій із різними об'єктами системи. Таке подання спрощує додавання нових характеристик користувача: щоб урахувати новий аспект поведінки або контексту, достатньо ввести новий клас чи властивість в онтології та пов'язати його з наявними елементами.

Персоналізація, що спирається на онтологію, може враховувати не лише прямі події (перегляд, замовлення, оцінка), а й семантичні зв'язки між об'єктами, з якими взаємодіє користувач. Наприклад, якщо користувач часто обирає страви певного типу, системи такого класу можуть виводити з онтології інші об'єкти з подібними властивостями (спільна кухня, діапазон цін, набір інгредієнтів) і використовувати це для налаштування пропозицій, фільтрів або пріоритетів у відображенні результатів.

У загальному випадку одна й та сама онтологічна модель профілю може підтримувати різні інтерфейси для різних ролей користувачів, якщо у ній описані відповідні ролі та їхній зв’язок із доступними функціями.

1.4.3. Онтологічні моделі в рекомендаційних системах веб-додатків

У рекомендаційних системах веб-додатків онтологічні моделі застосовуються для поєднання поведінкових даних з доменними знаннями. Поведінковий рівень зазвичай представлений логами переглядів, замовлень, кліків і оцінок, тоді як онтологія описує структуру предметної області: класи об’єктів (наприклад, страви, заклади, категорії меню), їхні властивості (тип кухні, склад, ціновий сегмент) та відношення між ними (належність до меню, зв’язок із певним закладом тощо). Поєднання цих двох джерел дає змогу будувати гібридні рекомендаційні алгоритми, що враховують як схожість поведінки користувачів, так і семантичну близькість об’єктів.

У системах для онлайн-замовлення їжі, наприклад, онтологія може допомагати визначати, які страви вважати ‘подібними’ для цілей рекомендацій: не лише за історією співзамовлень, а й за загальними ознаками домену (одна кухня, близький набір інгредієнтів, однакові або суміжні категорії меню). Це особливо корисно в ситуаціях, коли про конкретну страву ще мало поведінкових даних (‘холодний старт’), але її місце в онтологічній структурі вже відомо. Загалом використання онтології в рекомендаційних модулях дозволяє робити висновки не тільки на основі статистики, а й на основі явно заданих знань про предметну область, що покращує пояснюваність та стійкість рекомендацій.

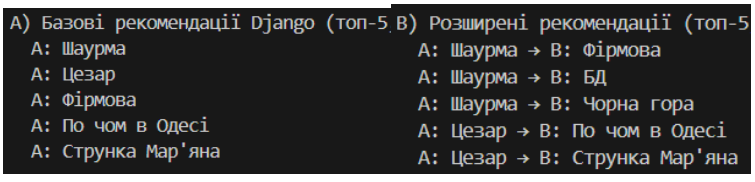

Рис. 1.8. Приклад формування базових рекомендацій Django та їх онтологічного розширення на основі зв’язків isSimilarTo між стравами в GraphD

1.5. Висновок до розділу

Розділ 1 формує теоретичну основу роботи: вводиться поняття онтології, описуються її основні елементи (класи, властивості, індивіди) та роль логік опису, TBox/ABox та аксіом для формального представлення знань та логічного виводу. Показано, як RDF, RDFS та OWL формують технологічний стек Semantic Web, який дозволяє перейти від табличного представлення даних до графу знань, та як онтологічний підхід дозволяє точніше моделювати предметну область веб-систем та підтримувати семантичний пошук, інтеграцію та інтелектуальні функції.

Окремо розглядається класифікація онтологій (загальна, предметна, задачна, прикладна) та місце онтологій у багаторівневій архітектурі Semantic Web, а також принципи побудови онтологічної моделі веб-застосунку маркетплейсу (сутності, зв'язки, обмеження). Порівняння з ER-моделюванням показує переваги семантичного підходу для повторного використання знань, логічної узгодженості та інтеграції з іншими системами, що обґрунтовує вибір онтологічної технології як основи для подальшої практичної реалізації.

РОЗДІЛ 2
ТЕХНОЛОГІЇ ТА МЕТОДИ ІНТЕГРАЦІЇ ОНТОЛОГІЙ У ВЕБ ЗАСТОСУНКИ

2.1 Стек технологій та онтологія семантичного вебу (репозиторії RDF, SPARQL, Triple Store)

Онтологічні моделі у веб-застосунках спираються на стандартні технології семантичного вебу, які забезпечують єдиний спосіб представлення, зберігання та запиту даних у вигляді графа знань. Цей стек включає модель RDF, інструменти збагачення на основі RDFS та OWL, спеціалізовані репозиторії RDF (Triple Store) та мову запитів SPARQL, які разом утворюють інфраструктуру для інтеграції онтології в програмні системи.

2.1.1 RDF як основа для представлення даних

Структура опису ресурсів (RDF) описує інформацію як набір трійок об'єктів, де суб'єкт та об'єкт є ресурсами або літеральними значеннями, а предикат визначає тип зв'язку між ними. Ідентифікація ресурсів здійснюється за допомогою URI/IRI, що дозволяє однозначно посилатися на об'єкти незалежно від конкретної реалізації або розташування даних. Колекція трійок утворює орієнтований граф: вузли призначені для суб'єктів та об'єктів, а ребра - для предикатів, що з'єднують ці вузли.

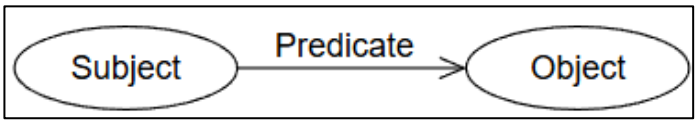

Рис. 2.1. Трійка subject - predicate - object

Графова природа RDF забезпечує гнучкість, яка тісно пов'язує типізовані табличні моделі: нові вузли та ребра можна додавати до існуючого графа без схем міграції, оскільки дані з різних джерел можна об'єднувати за спільними URI. У семантичному вебі RDF використовується для опису ресурсів та їх зв'язків на

формальному рівні, причому кожна трійка є твердженням про предметну область, а граф - це набір таких тверджень, які придатні для логічного виведення та повторного використання в інших програмах. Саме на RDF будуються схеми RDFS, онтології OWL та запити SPARQL, які працюють не з таблицями, а з узгодженим графом знань.

2.1.2 RDFS та OWL як засоби збагачення моделі RDF

Схема RDF (RDFS) розширює базову модель RDF, надаючи можливість оголошувати класи, властивості та ієрархії між ними. Використовуючи конструкції rdfs:Class, rdfs:subClassOf, rdfs:subPropertyOf, розробник формує дерево понять (наприклад, "Заклад" → "Ресторан" → "Піцерія"), а використовуючи rdfs:domain та rdfs:range, він вказує, для яких типів ресурсів використовуються певні властивості. Такі оголошення не тільки документують модель, але й використовуються системами міркування для автоматичного виведення типів: якщо ресурс відображається як об'єкт у властивості з певним розділом, система може зробити висновок, що він належить до відповідного класу.

OWL (Web Ontology Language) - це наступний рівень поверх RDF(S) і забезпечує набагато потужніший апарат описової логіки для побудови онтологій. На відміну від RDFS, де вираження обмежене простими ієрархіями та доменами/діапазонами, OWL дозволяє встановлювати аксіоми еквівалентності та несумісності класів, встановлювати обмеження кардинальності, визначати транзитивні, симетричні та функціональні властивості, а також конструювати складні класи за допомогою операцій перетину, об'єднання або додавання. У результатах онтології OWL може явно кодувати правила домену та підтримувати автоматичне виявлення звітів, класифікацію об'єктів та виведення нових фактів - це можливості, що використовуються в системах, де онтологія відіграє роль активного компонента, а не лише документації.

2.1.3 RDF-репозиторії та Triple Store (на прикладі GraphDB)

Triple Store або RDF-репозиторій - це спеціалізована СУБД, оптимізована для зберігання та обробки RDF-трійок та запитів до графа знань. На відміну від

реляційних баз даних, де структура жорстко визначена схемою таблиці та зовнішніми ключами, Triple Store працює в універсальному потрійному форматі та підтримує динамічне додавання нових типів ресурсів та зв'язків без зміни схем представлення. Такі репозиторії раніше підтримували іменовані графи, різні індекси над трійками, механізми кешування результатів виводу та стандартні інтерфейси доступу до даних у вигляді кінцевих точок SPARQL.

GraphDB - це приклад промислового RDF-репозиторію, орієнтованого на роботу з великими обсягами даних RDF/OWL та використання RDFS/OWL-впливу. Він надає вбудований засіб міркування, який може автоматично виводити нові трійки з урахуванням аксіом RDFS та OWL, підтримує інтерфейси REST/HTTP для виконання запитів та оновлень SPARQL, а також містить інструменти візуалізації графів та моніторингу стану репозиторію. У таких проектах, як веб-маркетплейси, GraphDB використовується як семантичний шар: онтологія предметної області та пов'язані з нею особи (заклади, страви, взаємодії) зберігаються в окремому репозиторії, до якого веб-додаток звертається зовні через кінцеву точку SPARQL паралельно з роботою з реляційною базою даних.2.1.4 Мова запитів SPARQL для доступу та оновлення даних

2.1.4 Мова запитів SPARQL для доступу та оновлення даних

SPARQL - це стандартна мова запитів для RDF-даних, яка виконує в екосистемі Semantic Web роль, подібну до SQL у світі реляційних баз даних.

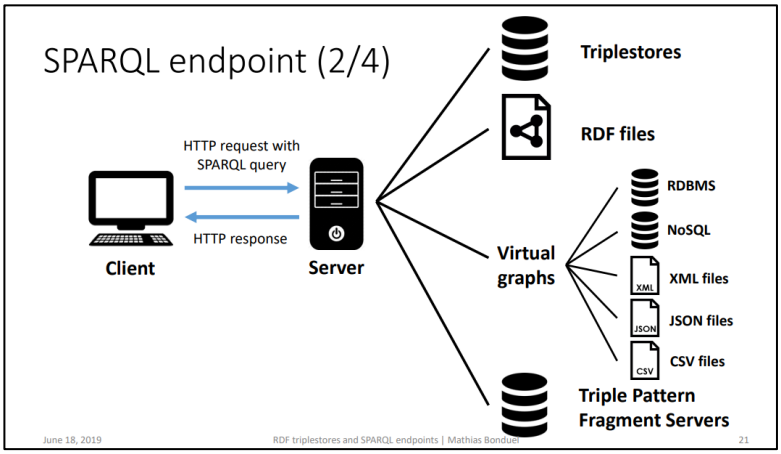

Рис. 2.2. Приклад архітектури SPARQL-endpoint

Основою запиту SPARQL є потрійний шаблон, де змінні можуть займати місце суб'єктів, предикатів та об'єктів; система знаходить у графі всі підстановки, для яких виконується цей шаблон, і повертає їх у вигляді таблиці результатів або нового підграфа. Окрім вибору (оператори SELECT, CONSTRUCT, DESCRIBE, ASK), SPARQL підтримує операції оновлення даних (INSERT/DELETE), що дозволяє не тільки читати, але й змінювати RDF-граф у потрійному сховищі через єдиний інтерфейс запитів.

У поєднанні зі сховищами RDF, такими як GraphDB, SPARQL реалізується через кінцеву точку HTTP, до якої можуть отримати доступ зовнішні веб-додатки, сервіси або інтеграційні скрипти. Це дозволяє чітко розподілити обов'язки: транзакційні операції та базові дані обробляються реляційною базою даних, тоді як семантичні запити, пов'язані з онтологією та графом знань (пошук зв'язків, класифікація об'єктів, генерація рекомендацій), виконуються через SPARQL поверх Triple Store. Такий підхід типовий для архітектур, де онтологія інтегрована у веб-застосунок як окремий логічний та технологічний рівень.

2.2 Інтеграція онтологій з веб-фреймворками (на прикладі Django)

2.2.1 Загальна архітектурна схема ‘Реляційна база даних ↔ Онтологія ↔ Веб-додаток’

Веб-додатки, що поєднують реляційні дані та онтології, часто використовують трирівневу схему, де класична база даних виступає операційним ядром, репозиторій онтологій - семантичним шаром, а веб-фреймворк (Django) координує взаємодію між ними. На рівнях реляційних баз даних (SQLite, PostgreSQL або інша СУБД) структуровані таблиці користувачів, закладок, меню, страв, замовлень та історії взаємодії зберігаються за допомогою ORM, яка забезпечує механізми транзакцій, цілісності та міграції, знайомі розробникам. Онтологія в окремому репозиторії RDF представляє ті ж сутності, що й граф знань, доповнені класами, властивостями, аксіомами та додатковими зв'язками (наприклад, ‘схожі страви’, ‘тип кухні’, ‘приналежність до категорії’), які явно не присутні в реляційній схемі.

Веб-додаток Django розташований над обома репозиторіями та виступає єдиною точкою входу для клієнтської системи. Типовий потік даних виглядає так: користувач виконує дію в інтерфейсі змін (створює замовлення, змінює меню, додає оцінку страви), Django обробляє HTTP-запит та записує його до реляційної бази даних через ORM. Після цього окремий компонент синхронізації (сигнал post_save, фонове завдання або спеціальна служба) генерує відповідні RDF-сутності та відношення і надсилає запити на оновлення до репозиторію графів, щоб підтримувати граф онтології узгодженим зі станом транзакційних даних. У зворотному напрямку веб-додаток може ініціювати семантичні запити до онтології - наприклад, час від часу або за певними сценаріями (створення рекомендацій, семантичний пошук) - та включати отримані результати у відповідь користувачам.

Таке розділення дозволяє вирішити додаткові вимоги: реляційна база даних відповідає швидким та надійним операціям, а онтологія - більш ‘важливим’ семантичним обчисленням та аналізу структури предметної області. Важливо, щоб вебзастосунок не дублював логіку зберігання всередині себе, а працював з обома рівнями через чітко визначені інтерфейси - ORM для зберігання SQL та HTTP/SPARQL для графічного.

Рис. 2.3. Узагальнена архітектура інтеграції веб-застосунку Django з реляційною базою даних та RDF-репозиторієм GraphDB, де онтологія діє як семантичний шар над операційними даними

2.2.2 Зіставлення сутностей реляційної бази даних з класами та властивостями онтологій

Для того, щоб онтологія та транзакційна база даних описували одну й ту саму реальність, потрібне явне зіставлення між сутностями реляційної бази даних та елементами моделі RDF/OWL. Основна рекомендація - відповідність ‘таблиця → клас онтології’: таблиці User, Establishment, Menu, MenuItem, Order, Rating у схемі Django перетворюються на класи User, Establishment, Menu, MenuItem, Order, Rating в онтології. Поля таблиці, що містять значення (імена, описи, ціну, позначки часу, статуси), перетворюються на властивості класів даних з відповідними літеральними типами (рядок, число, дата/час). Зовнішні ключі між таблицями перетворюються на властивості об'єктів: наприклад, відношення ‘Order - Establishment’ стає властивістю hasOrderEstablishment, а ‘MenuItem - Menu’ стає властивістю hasMenuItem.

Під час проектування такого зіставлення важливу роль відіграє узгодженість імен та типів. Бажано уникати ситуацій, коли одна й та сама сутність називається по-різному в базі даних та онтології, оскільки це ускладнить синхронізацію та читання моделі. Для зв'язків ‘один до багатьох’ достатньо забезпечити правильну спрямованість властивостей об'єкта, а для зв'язків ‘багато до багатьох’ достатньо вирішити, чи відображати таблицю зв'язків як окремий клас (подія взаємодії, участь у замовленні), чи як багаторазове використання властивості об'єкта. У складніших випадках, коли реляційна схема містить денормалізовані поля або технічні таблиці, в онтології можна розробити більш ‘чисту’ концептуальну структуру, а відображення можна реалізувати на рівні коду перетворення.

Окремим питанням є стабільні ідентифікатори. Якщо рядки в таблицях мають числові первинні ключі, RDF-індивідам зазвичай надаються IRI, які включають тип сутності та значення ключа (наприклад, …/User/123, …/Order/456). Це полегшує відстеження між реляційним та семантичним рівнями та дозволяє однозначно асоціювати дані під час оновлень.

2.2.3 Технічна інтеграція з Django

З точки зору Django, інтеграція онтологій реалізується як набір сервісних функцій або модулів, що відповідають базовій бізнес-логіці. Стандартні моделі Django використовуються для виконання операцій CRUD над реляційними таблицями, а модуль семантичної інтеграції обробляє RDF-представлення та обмін зі сховищем графів. Найпростіша версія коду для роботи з онтологією включає:

- функції, які будують набір RDF-трійок (рядки Turtle або SPARQL INSERT) з екземплярів моделі Django;
- клієнт для надсилання цих запитів до кінцевої точки SPARQL через HTTP та обробки відповідей;
- функції читання, які надають запити SPARQL SELECT/CONSTRUCT та повертають результати як зручні об'єкти Python.

Для збереження чистоти архітектури доцільно перемістити такі функції в окремий модуль або сервісний клас та виключити з цих частин програми ті частини, де робота з онтологією дійсно потрібна: під час імпорту даних, під час оновлення ключових сутностей, під час побудови рекомендацій. Для фінальних або масових операцій доречно використовувати керуючі команди або фоновий виконавець (чергу завдань), де будуть реалізовані сценарії типу ‘експорт усіх поточних страв та закладок до GraphDB’ або ‘перерахунок семантичних зв'язків після змін у меню’.

Ще однією технічною деталлю є обробка помилок та відмовостійкість. Якщо репозиторій графів є зовнішнім компонентом, що відповідає локальній БД, варто врахувати, що станеться з системою, якщо кінцева точка SPARQL буде тимчасово недоступна: чи буде застосовано оновлення онтології в цей час, чи певні функції (наприклад, розширені рекомендації) будуть тимчасово вимкнені, а базові скрипти продовжать працювати лише над реляційними даними.

2.2.4 Місце онтологічного шару в життєвому циклі веб-додатку

Онтологічний шар відіграє різні ролі на різних етапах життєвого циклу системи. На початковому етапі розгортання він використовується для створення семантичного ‘каркасу’ предметної області: після застосування міграцій у реляційній

базі даних можна запускати скрипти, які зчитують посилання (типи закладів, категорії страв, базові меню) та завантажують їх у RDF-репозиторій як класи та початкові особини. На цьому етапі формується впорядкований граф, який потім буде доповнено актуальними даними з сервісу.

В операційному режимі онтологія зазвичай оновлюється на основі подій: створення нового закладу або страви, зміна важливої властивості, поява нового типу взаємодії може спровокувати додавання або модифікацію трійок у GraphDB. Ці оновлення можуть виконуватися синхронно (в межах одного запиту) або асинхронно, якщо допускається невелика затримка між транзакційними та семантичними даними. Паралельно, онтологія використовується як джерело знань у сценаріях, де потрібні ‘інтелектуальні’ функції: генерація списків схожих страв, пошук закладів за комбінацією властивостей, побудова рекомендацій на основі поведінкових та семантичних зв'язків. На цьому етапі веб-додаток виконує SPARQL-запит до GraphDB, отримує семантично збагачені результати та поєднує їх з транзакційними даними з реляційної бази даних.

На етапах підтримки та розробки онтологія дозволяє системі розвиватися без радикальних змін реляційної схеми: можна додавати нові класи, властивості та аксіоми, що відображають уточнену модель предметної області або нові бізнес-вимоги. Часто достатньо розширити онтологію та налаштувати відповідні SPARQL-запити для отримання нових аналітичних представлень або сценаріїв рекомендацій, не торкаючись основних таблиць бази даних. Таким чином, шар онтології стає не лише пасивним дзеркалом операційних даних, але й інструментом для еволюції знань у системі.

2.3 Моделі та алгоритми передачі даних з реляційних баз даних до онтології

2.3.1 Концепція відображення між реляційною моделлю та онтологією

Передача даних з реляційної бази даних до онтології базується на концепції відображення, коли елементи реляційної схеми узгоджуються з елементами моделі

RDF/OWL. Реляційна база даних оперує таблицями, полями та ключами, потім, як онтологія - класами, властивостями даних та об'єктів, індивідами та аксіомами, першим кроком є визначення відповідності між цими двома рівнями.

У найпростішому варіанті застосування використовується набір основних правил:

- кожна суттєва схема таблиці (Користувач, Сховище, Меню, Елемент меню, Порядок тощо) відображається на окремий клас онтології;
- поле простого типу (рядок, число, дата, логічне значення) стає властивістю дати відповідного класу;
- зовнішній ключ, що вказує на іншу таблицю, перетворюється на властивість об'єкта, яка пов'язує індивіди двох класів;
- Кожен запис таблиці створює об'єкт класу з IRI, сформованим на основі типу сутності та первинного ключа (наприклад, …/Order/123).

Ці правила дозволяють відтворити в RDF-графі структуру зв'язків, яка вже міститься в реляційній базі даних, а потім доповнити її багатшими семантичними зв'язками та аксіомами.

2.3.2 Алгоритм початкової міграції даних з реляційної бази даних до онтології

Початкова міграція даних у таких проектах, як веб-маркетплейс, зазвичай реалізується у вигляді наступної кількості кроків.

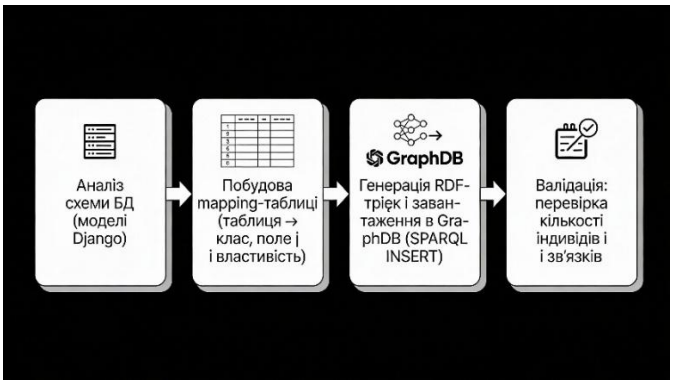

Рис. 2.4. Алгоритм початкової міграції даних з реляційної бази даних до репозиторію онтологій: аналіз схеми, побудова таблиці відображення, генерація RDF-трійки та завантаження в GraphDB з подальшою валідацією результатів

На першому кроці аналізується схема веб-застосунку бази даних: вивчаються моделі Django, їх поля, типи та зв'язки ‘один до багатьох’ та ‘багато до багатьох’. Це дозволяє програмі визначити, які таблиці є основними сутностями предметної області (а які забезпечують технічну роль (таблиці зв'язків, каталоги послуг).

На другому кроці будується таблиця відображення між моделями Django та елементами онтології: для кожної моделі вказується цільовий клас OWL, для кожного поля - властивість дати або властивість об'єкта, а також спосіб формування IRI осіб. На третьому кроці запускається обхід записів у базі даних: для кожного об'єкта Django створює набір RDF-трійок, які або записуються у файл (Turtle/N-Triples), або одразу надсилаються до GraphDB через запити SPARQL-INSERT. У цьому випадку зовнішні ключі перетворюються на трійки з відповідними властивостями об'єкта, що відновлюють зв'язки між особами.

Четвертий крок - валідація міграції. Після завантаження даних у Triple Store перевіряється, чи відповідає кількість створених індивідів основних класів кількості записів у вихідних таблицях, а також чи зберігаються ключові зв'язки між ними (наприклад, ‘кожне замовлення має клієнта та заклад’) та чи можуть бути відтворені за допомогою запитів SPARQL.

2.3.3 Особливості передачі даних до проектів маркетплейсу

У веб-додатку маркетплейсу для закладів громадського харчування основні моделі реляційних баз даних - User, Role, Partner (Establishment), Menu, MenuItems, Order, OrderMenuItems, Rating, UserInteraction, UserMenuInterest - мають природні аналоги в онтології предметної області. Наприклад, таблиця User відповідає класу User, таблиця Role відповідає класу Role, а таблиця Partner відповідає класу Establishment; Таблиці Menu та MenuItems формують класи Menu та MenuItem (або Dish), які пов'язані з властивістю hasMenuItem. Зв'язки між моделями, заданими в Django через ForeignKey, утворюють властивості об'єкта: Order.client відповідає hasOrderClient, Order.partner відповідає hasOrderEstablishment, OrderMenuItems.menu_item відповідає orderHasOrderMenuItem тощо.

Приклади типових зіставлень для цього об'єкта можуть виглядати так:

- поле MenuItems.price у реляційній базі даних відповідає властивості даних :price класу MenuItem з літеральним типом "decimal";
- зовнішній ключ Order.partner_id - властивості об'єкта :hasOrderEstablishment, яка пов'язує об'єкт класу Order з об'єктом класу Establishment;
- Таблиця ‘Рейтинг’ створює клас ‘Рейтинг’, де поля ‘оцінка’ та ‘коментар’ залишаються властивостями даних, а зв’язки з користувачем та стравою - властивостями об’єкта hasRatingAuthor та hasRatingItem.
- Додаткові таблиці, такі як UserInteraction та UserMenuInterest, дозволяють вводити в онтологію класи, що представляють взаємодію з користувачем та події інтересів, які можна використовувати для побудови рекомендацій та аналізу поведінки.

У цій предметній області дані умовно поділяються на два типи. Деякі сутності є ‘операційними’ - це замовлення та записи взаємодій користувачів, які часто змінюються. Інша частина - це відносно стабільні довідники, такі як ролі користувачів, типи закладів або категорії страв. Під час міграції зручно спочатку завантажити дані довідника в онтологію як базовий шар, а оперативні записи переносити поступово: великими порціями або кожного разу після певних подій у реляційній базі даних. Такий підхід допомагає підтримувати граф знань в актуальному стані та водночас не перевантажувати GraphDB постійними оновленнями.

2.4 Методи синхронізації даних між реляційною базою даних та репозиторієм онтологій

2.4.1 Архітектурні підходи до синхронізації: пакетний імпорт та онлайн-синхронізація

Синхронізація реляційної бази даних з репозиторієм онтологій може виконуватися у двох основних режимах: пакетному та майже в реальному часі. У пакетному варіанті граф RDF оновлюється за розкладом: запускається окрема команда управління або cron-процес, який зчитує поточний стан таблиць, формує трійки та

повністю або частково перезавантажує граф онтології (наприклад, лише каталоги або дані за останні 24 години). Цей режим простіший у реалізації, добре підходить для повільно змінюваних даних (типи закладів, ролі, категорії харчування) та дозволяє планувати навантаження на Triple Store - запускати інтенсивні операції вночі або з обмеженою частотою. Для ілюстрації корисно використовувати діаграму, де ліва частина показує ‘Django + реляційна БД’, середня частина показує ‘Пакетна обробка / cron’, а права частина показує ‘GraphDB (репозиторій RDF)’

Онлайн-синхронізація працює інакше: онтологія оновлюється в режимі, близькому до реального часу, у відповідь на події в логіці програми. На веб-маркетплейсі це означає, що після створення нового замовлення, зміни статусу, додавання відгуку або нової страви до меню, відповідні особи та зв'язки одразу з'являються в GraphDB, а граф знань постійно відображає поточний стан системи. Перевагою такого підходу є найактуальніші дані для семантичних запитів та рекомендацій; недоліками є додаткові запити до RDF-репозиторію для кожної події, складність обробки помилок та необхідність механізмів повторення або черг завдань, щоб уникнути блокування транзакційних операцій у реляційній БД. На рисунку можна побачити сценарій ‘Django (події в моделях) → GraphDB (SPARQL INSERT/DELETE в реальному часі)’, який підкреслює різницю між режимами.

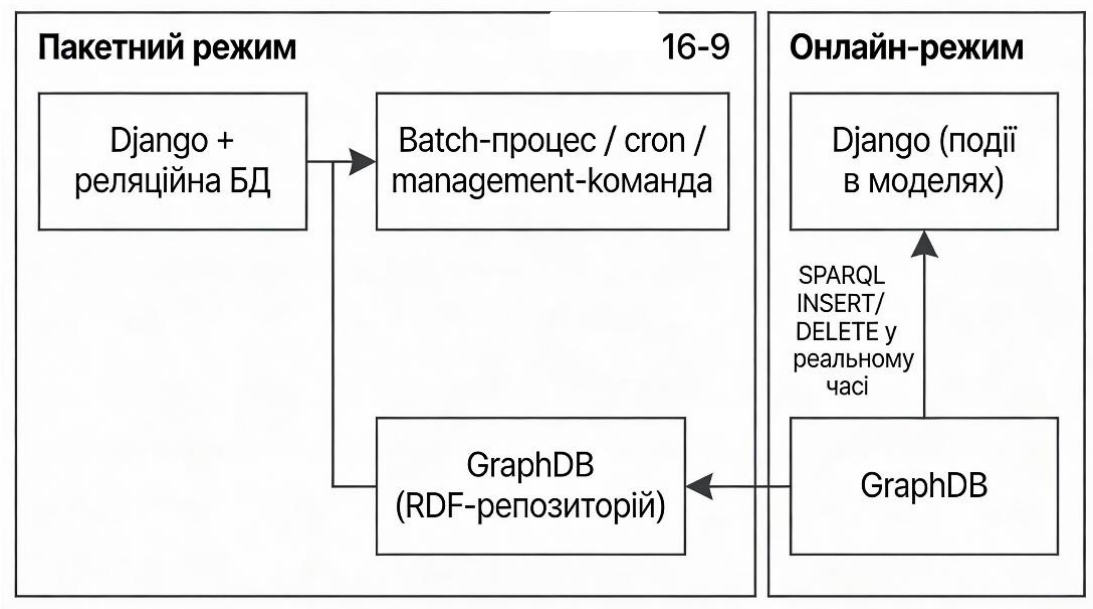

Рис. 2.5. Пакетний vs онлайн-режим

Підсумовуючи, можна коротко порівняти режими:

- Пакетний імпорт: простіший код, але менш актуальні дані.
- Онлайн-синхронізація: складніша реалізація, але граф онтології завжди близький до поточного стану системи.

2.4.2 Реалізація синхронізації в реальному часі (події/тригери/сигнали)

У таких фреймворках, як Django, онлайн-синхронізація з репозиторієм онтологій зазвичай реалізується на рівні подій логіки програми, а не через тригери в самій СУБД. Django надає механізм сигналів (зокрема, post_save, post_delete або pre_delete), який дозволяє централізовано реагувати на створення, оновлення та видалення екземплярів моделі. Для важливих сутностей - User, Establishment (Partner), MenuItem, Order, Rating, UserInteraction - можна визначити обробники сигналів, які:

- Перетворюють поточний стан об'єкта в набір RDF-трійок, що відповідають онтологічній моделі (класи, дата та властивості об'єкта);
- Генерують SPARQL INSERT/DELETE;
- Ініціюють оновлення графа знань у GraphDB

Типовий ланцюжок виглядає так:

1. Користувач створює або змінює об'єкт у веб-інтерфейсі;
2. Django зберігає його через ORM у реляційній базі даних а
3. Активується відповідний сигнал;
4. В обробнику поля моделі зіставляються з властивостями онтології, генерується запит SPARQL;
5. запит надсилається до кінцевої точки SPARQL GraphDB через HTTP;
6. У разі успіху граф знань відображає новий стан системи; у разі помилки запит може бути повторно поставлений у чергу або записаний у журнали.

Доцільно відобразити цю послідовність окремою блок-схемою ‘Подія у веб-інтерфейсі → Модель Django + ORM → Сигнал → Модуль генерації SPARQL → Кінцева точка SPARQL GraphDB → Оновлений граф знань’ (рис. 2.6).

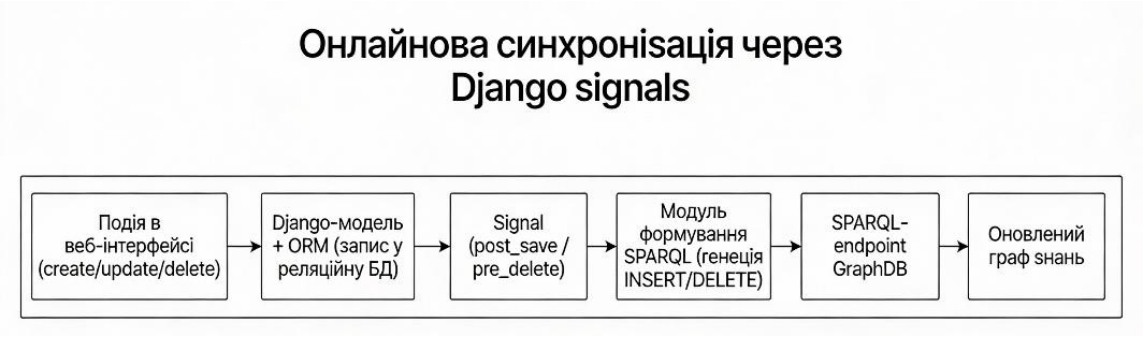

Рис. 2.6. Схема онлайнової синхронізації через Django-signals

Така діаграма чітко показує, що реляційна база даних залишається основним транзакційним сховищем, а онтологія є синхронізованою проекцією для семантичних завдань.

2.4.3 Однонаправлена та двонаправлена синхронізація

З точки зору напрямку обміну даними, існує два підходи до синхронізації між реляційною базою даних та репозиторієм онтологій.

Однонаправлена синхронізація:

- реляційна база даних є єдиним джерелом істини (головним);
- усі операції CRUD виконуються через моделі Django та репозиторій SQL;
- репозиторій RDF виступає як похідний семантичний шар (семантична проекція), який оновлюється на основі змін у базі даних та використовується для міркувань, семантичних запитів та рекомендацій.

Для маркетплейсу цей варіант є базовим: транзакційна модель залишається простою та надійною, а GraphDB забезпечує додатковий «інтелект», не впливаючи на критичні бізнес-процеси.

Двонаправлена синхронізація:

- зміни можна вносити як до реляційної бази даних, так і безпосередньо до репозиторію онтологій;
- обидва репозиторії повинні бути узгодженими, тобто зміни з одного боку повинні бути коректно відображені в іншому;

- Існує потреба в механізмах вирішення конфліктів, визначення порядку оновлень, фіксації джерела істини для кожного типу даних, а також для ведення журналу та контролю версій.

Цей підхід зазвичай використовується рідше, в системах, де експерти активно редагують знання безпосередньо в онтології (наприклад, додають нові семантичні відношення), і ці зміни необхідно повертати до реляційної схеми.

Зручно візуально показати різницю між підходами за допомогою рисунка, де ліворуч розташована односпрямована стрілка "Реляційна БД (основна) → GraphDB (семантична проекція)", а праворуч - дві протилежні стрілки "оновлення / зворотний зв'язок" між "Реляційною БД" та "GraphDB" з позначкою "конфлікти / узгодження версій" (рис. 2.7).

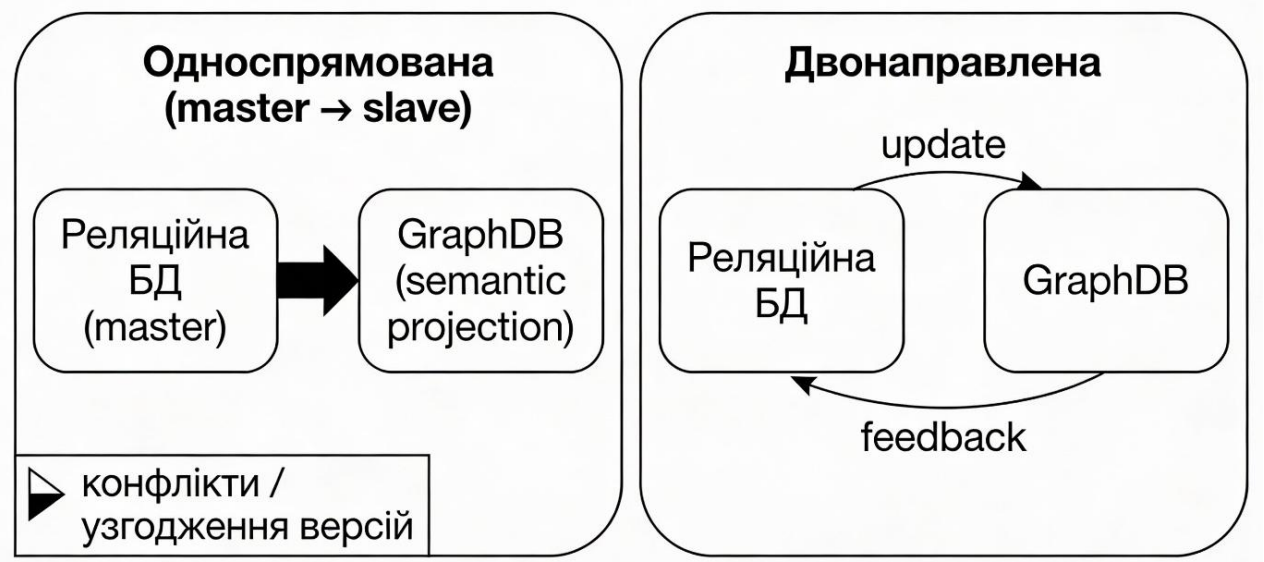

Рис. 2.7. Порівняння односпрямованої та двонаправленої синхронізації

Це чітко фіксує, що поточний дизайн використовує односпрямовану схему, тоді як двонаправлена синхронізація вважається теоретично можливою, але більш складною альтернативою.

2.5. Висновки до розділу

У розділі 2 аналізується технологічний стек Semantic Web (RDF, RDFS, OWL,

SPARQL, RDF-репозиторії) та показано, як ці інструменти інтегруються з класичними веб-фреймворками на прикладі Django. Описано трирівневу архітектуру "реляційна БД - онтологія в GraphDB - веб-додаток", принципи відображення таблиць та зв'язків БД на класи та властивості онтологій, а також технічні аспекти взаємодії через кінцеву точку SPARQL.

Особлива увага приділяється моделям передачі даних з реляційної БД до онтології та методам синхронізації між двома репозиторіями: пакетний імпорт, онлайн-синхронізація через сигнали Django, односпрямовані та двонаправлені схеми обміну. Це дозволяє сформувати загальну методологію інтеграції онтологічного рівня у веб-додаток без порушення транзакційної логіки та забезпечення релевантності графа знань для семантичних запитів та рекомендацій.

РОЗДІЛ 3
РОЗРОБКА АЛГОРИТМУ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ІНТЕГРАЦІЇ ОНТОЛОГІЇ З ЦІЛЛЮ ФОРМУВАННЯ РЕКОМЕНДАЦІЙ НА ЇЇ ОСНОВІ

3.1. Аналіз предметної області(веб-застосунок маркетплейс-логістичного рішення для закладів харчування з інтеграцією онтології для реалізації рекомендаційної системи)

Аналіз предметної області веб-застосунку маркетплейсу для закладів харчування є ключовим етапом при побудові інтелектуальної системи, яка поєднує класичні механізми обробки замовлень із семантичним моделюванням знань та рекомендаційною підсистемою. Такий застосунок повинен підтримувати взаємодію між кількома групами користувачів, керувати логістичними процесами та забезпечувати персоналізовані пропозиції на основі історії дій і семантичних зв’язків між об’єктами предметної області.

- Користувачі та ролі

У центрі системи знаходяться користувачі, які можуть виконувати різні ролі: клієнт (замовник страв), власник закладу, кухар, кур’єр, адміністратор. Для кожного користувача зберігаються персональні атрибути (ім’я, контактні дані, логін, електронна пошта, роль), а також історія його взаємодій із сервісом (замовлення, перегляди меню, оцінки страв). В онтології це відображається через класи User, Role та відповідні зв’язки (наприклад, hasRole, hasOrderClient), що дозволяє формалізувати відмінності в поведінці та правах доступу, а надалі використовувати роль і історію дій як основу для персоналізації рекомендацій.

- Заклади харчування, меню та страви

Маркетплейс об’єднує множину закладів харчування (ресторани, кафе, піцерії тощо), кожен з яких має власного власника, штат кухарів та кур’єрів, а також одне або кілька меню. Меню складається з набору страв (позицій меню), що описуються атрибутами: назва, ціна, склад, тип або категорія, зображення. У онтологічній моделі

ці сутності формалізуються як класи Establishment, Menu, MenuItem (або Dish), пов’язані властивостями hasMenu, hasMenuItems, hasOwner, hasCook, hasCourier тощо. Структура дозволяє явно виразити, які страви належать до якого меню й закладу, і підтримує подальший семантичний аналіз (наприклад, пошук подібних страв за інгредієнтами чи категоріями).

- Замовлення та логістичні процеси

Важливою складовою предметної області є процес оформлення та виконання замовлень. Кожне замовлення пов’язане з клієнтом, вибраним закладом, набором позицій меню, кур’єром, який здійснює доставку, адресою та статусом (створене, в обробці, готується, доставляється, виконане тощо). Ці зв’язки відображаються в онтології через клас Order та об’єктні властивості на кшталт hasOrderClient, hasOrderEstablishment, hasOrderCourier, orderHasOrderMenuItems. Логістичний аспект (хто готує, хто доставляє, з якого закладу, у який час) може бути додатково уточнений через аксіоми й обмеження, що дозволяють семантично перевіряти коректність зв’язків та аналізувати ефективність роботи закладів і кур’єрів.

- Онтологічна модель та графове сховище знань

На відміну від традиційної реляційної бази даних, що використовується у веб-фреймворку (наприклад, Django) для зберігання транзакційних даних, онтологічна модель описує предметну область на семантичному рівні: класи, ієрархії, ролі, обмеження та зв’язки між сутностями. Збереження онтології в RDF-сховищі (Triple Store) типу GraphDB у вигляді трійок (суб’єкт - предикат - об’єкт) дозволяє виконувати запити мовою SPARQL, використовувати механізми логічного виведення та отримувати з наявних фактів нові знання. Це створює основу для інтелектуальної поведінки системи - наприклад, визначення подібних страв через спільні інгредієнти чи категорії, виявлення найпопулярніших позицій меню для певного типу закладів або сегмента користувачів.

- Рекомендаційний механізм на основі онтології та поведінкових даних Рекомендаційна система у цій предметній області базується на поєднанні двох типів інформації:

- **поведінкових даних** (логи дій користувачів: перегляди страв, додавання до кошика, оформлення замовлень, виставлені рейтинги), які зберігаються в реляційній БД і відображаються в онтології через класи UserInteraction, Rating, UserMenuInterest;
- **семантичних зв'язків** між об'єктами (подібні страви, належність до певних меню й закладів, категорії кухні, цінові діапазони), зафіксованих в онтології через відповідні об'єктні властивості.

Узгоджена онтологічна модель предметної області разом із графовим сховищем знань та механізмами синхронізації з реляційною БД створюють цілісну інфраструктуру, у межах якої маркетплейс-логістичне рішення для закладів харчування може не лише обробляти замовлення, а й надавати інтелектуальну підтримку користувачам за рахунок семантично обґрунтованих рекомендацій.

3.2. Створення онтології в редакторі Protege

3.2.1. Створення файлу онтології та налаштування IRI

Для створення нової онтолігії спершу необхідно створити файл для даної онтології в програмі 'Protégé'. Після іменування даного файлу при спробі його зберегти у файловій системі ПК, відбувається збереження. Також в розділі 'Active Ontology' рядка 'Ontology IRI' відображається адреса - URI (Uniform Resource Identifier), що використовується для ідентифікації онтології в семантичній мережі:

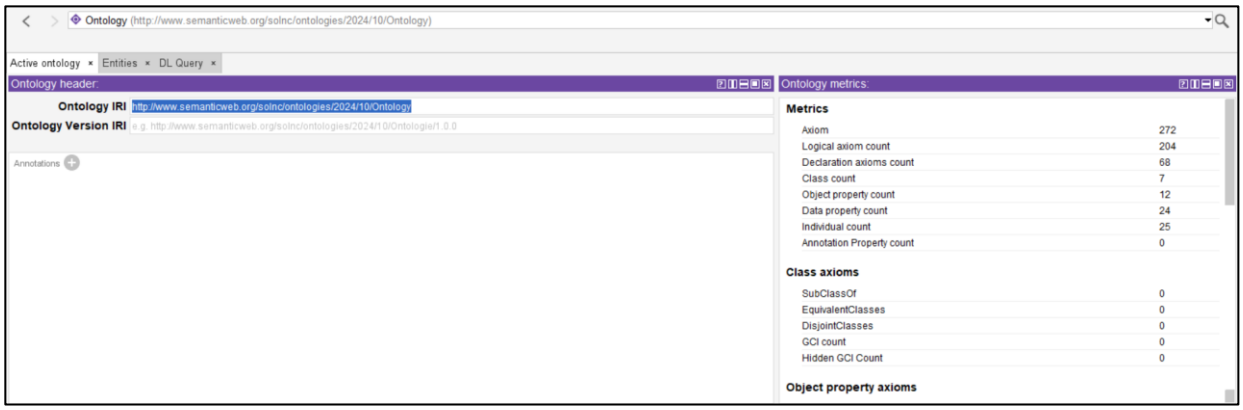

Рис. 3.1. Вікно Protégé з вказаним IRI створеної онтології

3.2.2. Перехід до розділу Entities та вибір методу моделювання

Для безпосереднього створення онтології необхідно перейти в розділ ‘Entities’:

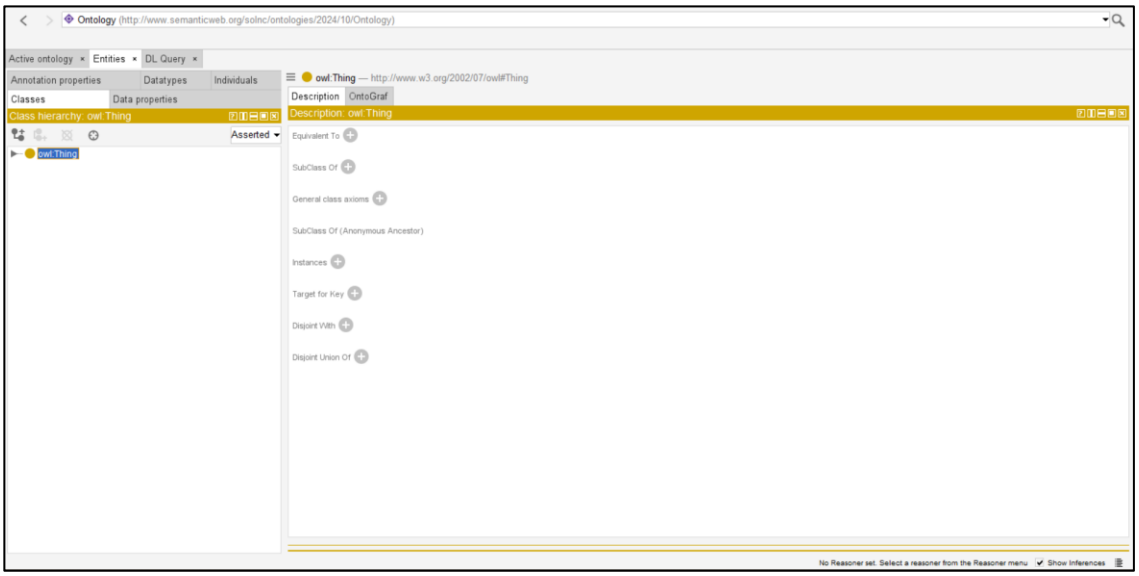

Рис. 3.2. Інтерфейс вкладки класів (Classes) з кореневим елементом owl:Thing

Дана онтологія створюватиметься на основі вже існуючої моделі реляційної бази даних, яка візуалізує структуру сайту для маркетплейсу закладів харчування:

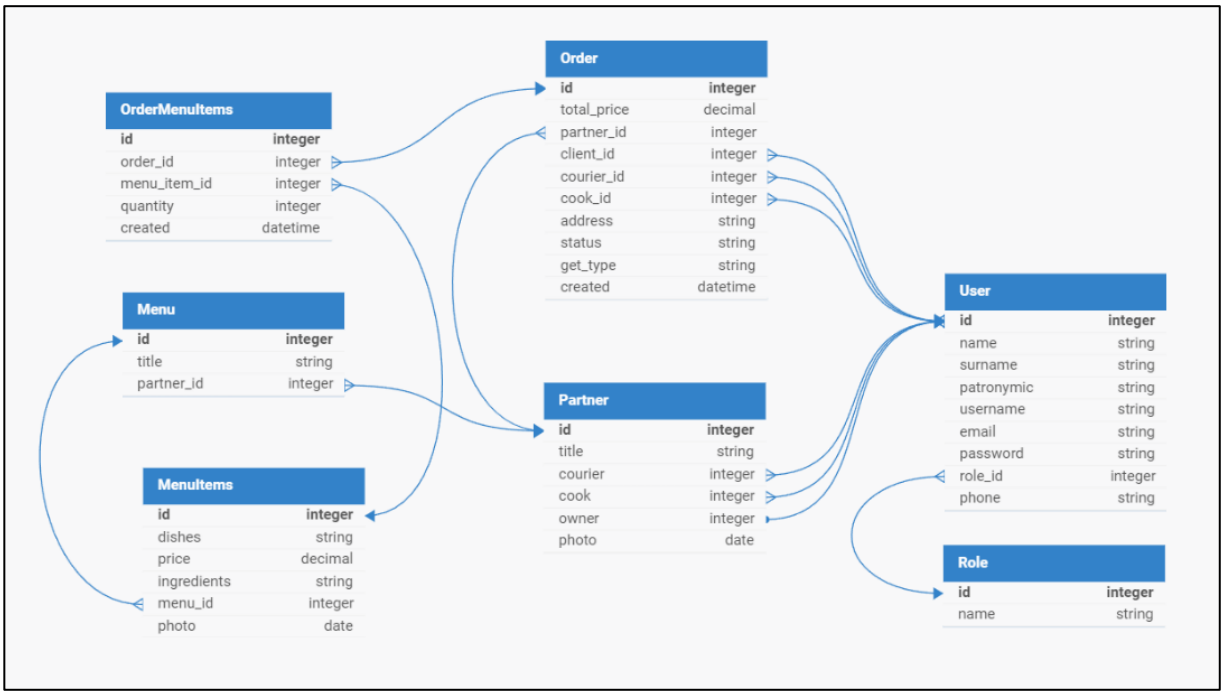

Рис. 3.3. ER-діаграма бази даних маркетплейсу із сутностями

Тож, стратегією виконання даної онтології вибрано створення:

1. Класів онтології як таблиць реляційних БД;
2. Зв’язків(Object Properties) між класами як ключів в реляційних БД;
3. Властивості даних(Data Properties) як поля реляційних БД.

Задля демонстрації роботи зв’язків класів один між одним будуть створені екземпляри класів(Individuals), які і володітимуть властивостями даних, на які накладатимуться зв’язки, що відображатимуть структуру онтології в цілому.

Для створення класу в відкритому розділі ‘Entities’ натискаємо на вкладку ‘Classes’. В даній вкладці з’явиться простір зліва, в якому знаходитиметься кореневий елемент ієрархії класів ‘owl:Thing’:

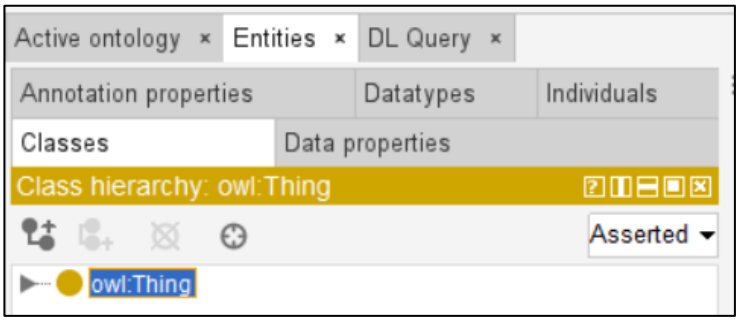

Рис. 3.4. Кореневий клас owl:Thing у Protégé

Натискаємо на даний каталог, та використовуємо першу піктограми над каталогом, що виконує функцію ‘Add subclass’, додаючи підклас в каталог:

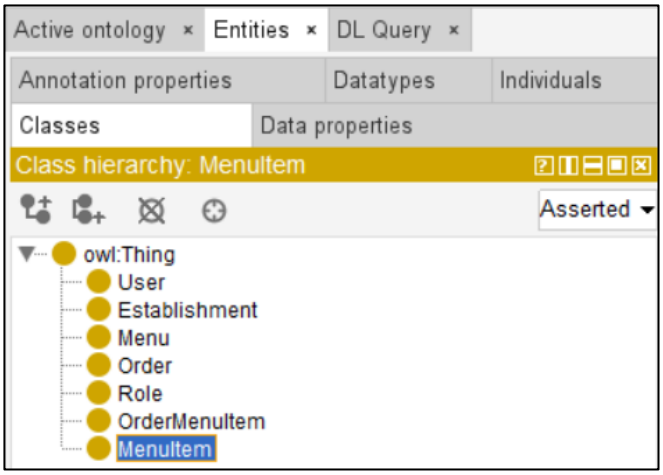

Рис. 3.5. Дерево класів онтології маркетплейсу у вкладці Classes

Для полегшення процесу заповнення каталогу я використовував другу піктограму над кореневим елементом ‘Thing’ - ‘Add sibling class’, що додає клас на тому ж рівні каталогу, на якому зараз знаходиться клас, який є виділеним.

3.2.3. Створення класів онтології на основі реляційних таблиць

Наступним кроком буде наповнення даної онтології зв’язками(Object properties). Для створення зв’язку(об’єктної властивості) в відкритому розділі ‘Entities’ натискаємо на вкладку ‘Object properties’. Принцип створення зв’зків такий самий як і створення класів:

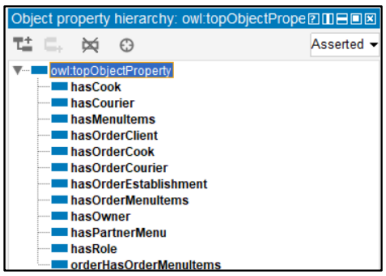

Рис. 3.6. Інтерфейс вкладки Object properties для створення об’єктних зв’язків

Налаштування зв’язків я продемонструю на прикладі зв’язку ‘hasOrderEstablishment’. В правій стороні екрану в вкладці ‘Description’ обираємо пункт `Domains` - визначає, до яких класів належить об'єкт, якому можна призначати певну властивість, та обираємо необхідний клас:

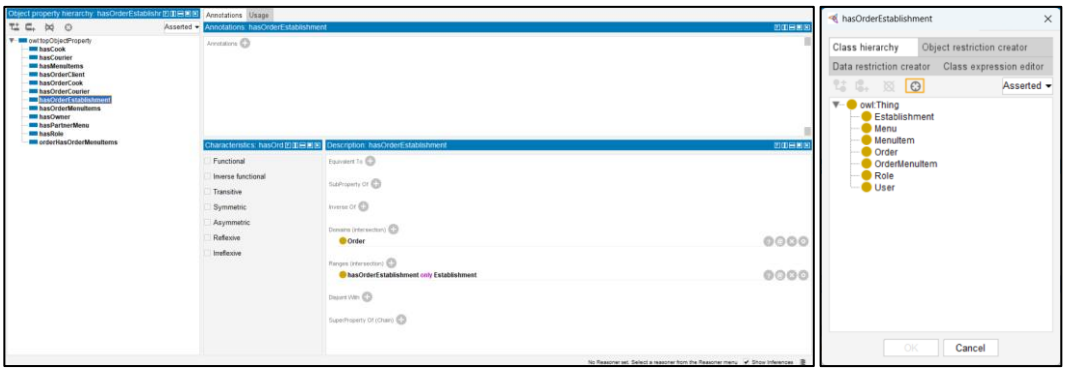

Рис. 3.7. Налаштування Domain для зв’язку hasOrderEstablishment у вкладці Description

В цій ж вкладці обираємо пункт ‘Ranges’ - діапазон, що визначає тип значень, які може мати властивість, тобто до яких класів або типів даних належать значення властивості, та встановлюємо значення використовуючи обмеження які дозволяють визначати кардинальність - кількість екземплярів того чи іншого класу, або характер зв’язку, по аналогії до реляційних БД: один до одного або один до багатьох:

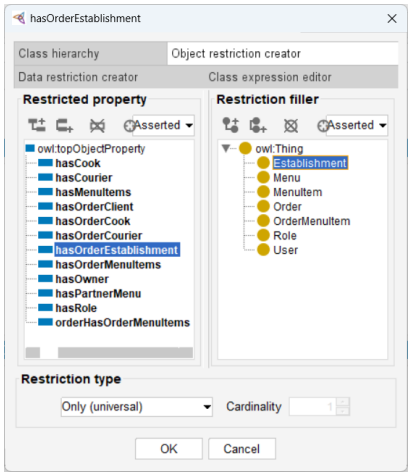

Рис. 3.8. Налаштування Range та кардинальності зв’язку hasOrderEstablishment

Після виконання даних маніпуляцій даний зв’язок належить до класу ‘Order’ діапазоном значень якого є клас ‘Establisment’, який є обмеженим в межах тільки одного екземпляру класу ‘Establisment’. Простими словами тепер в замовленні(Order) заклад, з якого було зроблено замовлення може бути тільки один(це є особливістю системи, що не дозволяє випадково, чи через проблеми з системою взяти позицію меню з іншого закладу, меню якого відкрите в іншій вкладці).

Таким чином налаштовуємо інші зв’язки, що зв’язують класи один між одним.

3.2.4. Налаштування об’єктних та дата-властивостей класів

Тепер слід налаштувати властивості даних(Data Properties), які будуть полями-значеннями для екземплярів класів. Для створення властивості даних(поля класу) в відкритому розділі ‘Entities’ натискаємо на вкладку ‘Object properties’. Принцип створення зв’язків такий самий як і створення класів, з єдиною відмінністю, що перед

самими властивостями було створено підклас, який називався так само як клас, якому призначалися дані властивості:

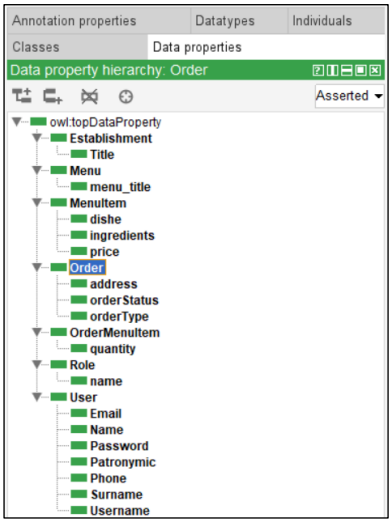

Рис. 3.9. Ієрархія властивостей даних (Data properties)

Налаштування властивостей я продемонструю на прикладі властивості ‘price’ підкласу ‘MenuItem’. В правій стороні екрану в вкладці ‘Description’ обираємо пункт `Ranges` - визначає, які типи даних можна присвоїти екземпляру класу(Individuals), та обираємо необхідний тип даних:

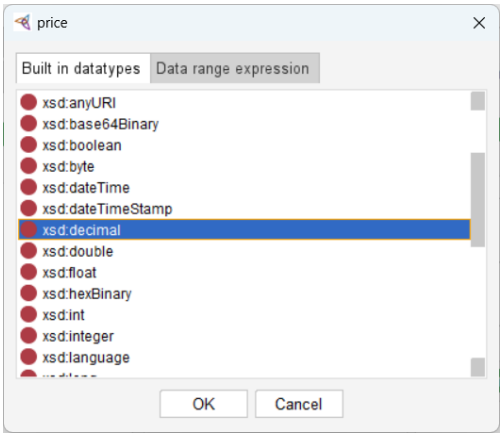

Рис. 3.10. Налаштування типу даних (Range) властивості price.

Таким чином налаштовуємо всі властивості даних, що відображаються як значення екземплярів класу.

3.2.5. Створення екземплярів класів та задання значень властивостей

Тепер слід створити необхідні екземпляри класів(Individuals), які відображатимуть всі зв'язки між класами, та можливість володіння значеннями, що передбачені властивостями даних самих класів. Для створення екземпляри класів (Individuals) в відкритому розділі 'Entities' натискаємо вкладку 'Individuals'.

Для створення екземплярів класу використовуємо кнопку 'Add individuals' у верхній панелі вкладки 'Individuals':

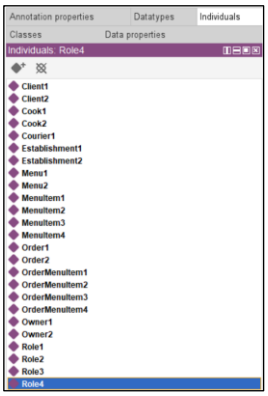

Рис. 3.11. Список екземплярів (Individuals) класів у відповідній вкладці

Налаштування 'Individuals' я продемонструю на прикладі екземпляру 'price' підкласу 'MenuItem'. В правій стороні екрану в вкладці 'Description' обираємо пункт 'Types' - дозволяє призначати екземпляр класу самому класу, та обираємо клас, якому належатиме даний 'Individual':

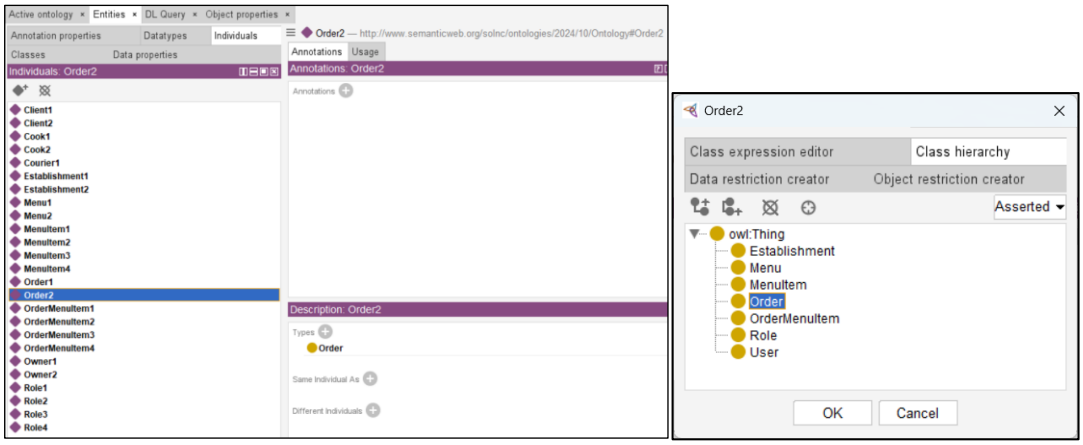

Рис. 3.12. Призначення типу (Types) екземпляру класу у вкладці Description

Або замість такого способу додавання екземпляру можна в розділі ‘Classes’ в правій стороні екрану в вкладці ‘Description’ обираємо пункт `Instances`, в меню якого додаємо необхідні інстанси класу:

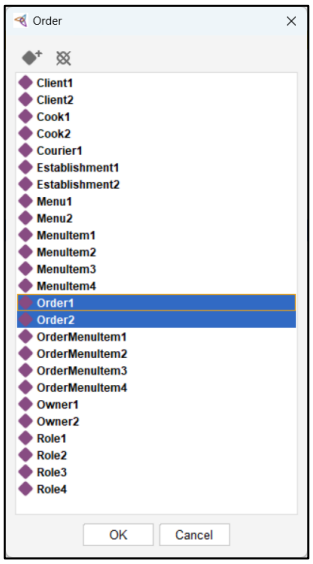

Рис. 3.13. Додавання екземплярів класу через розділ Instances

Задля ініціалізації інстансів значеннями потрібно в правій стороні екрану в вкладці ‘Property assertations’ обрати пункт `Data property assertations`, та в меню задати значення у відповідності до типу властивості даних:

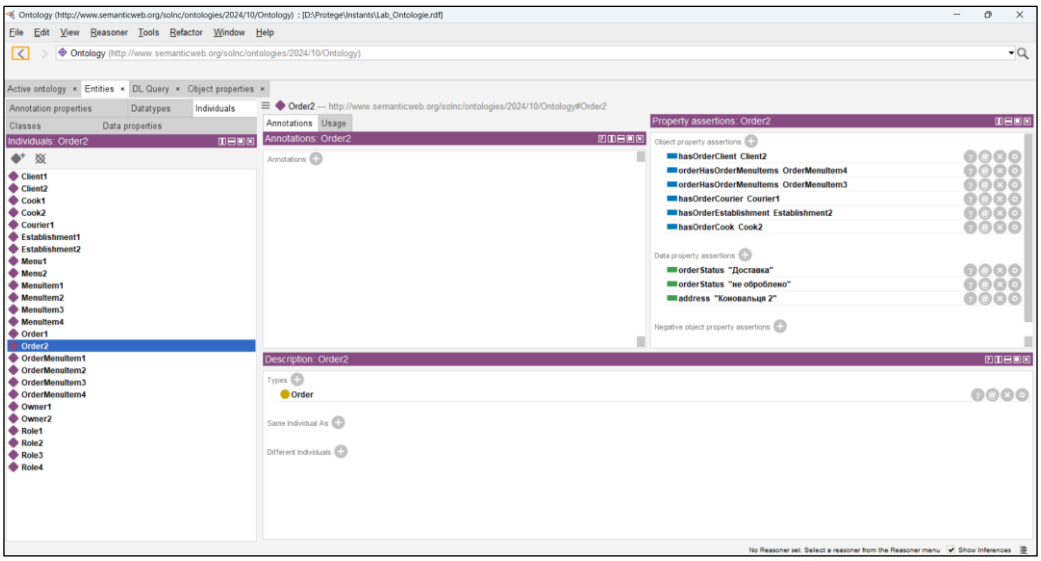

Рис. 3.14. Задання значень Data property assertions для екземпляра(початок)

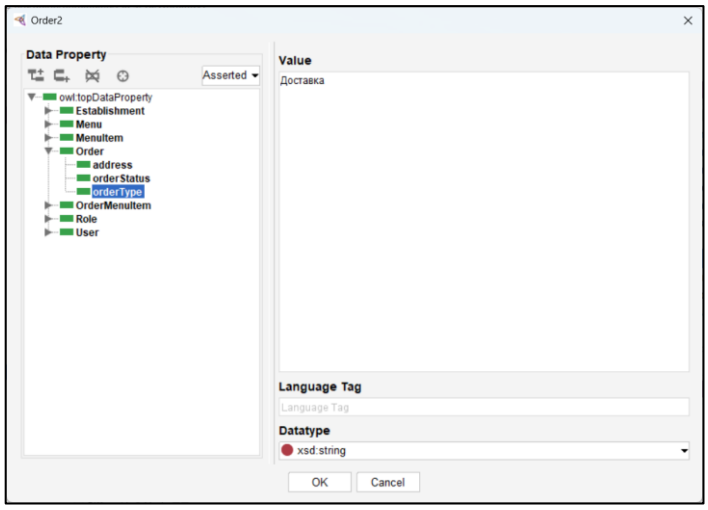

Рис. 3.14. Задання значень Data property assertions для екземпляра(закінчення)

Таким чином ініціалізуємо всі поля для всіх вже існуючих інстансів.

Для встановлення зв’язків між інстансами класів потрібно в правій стороні екрану в вкладці ‘Property assertations’ обрати пункт `Object property assertations`, та в меню задати пару-значення ‘зв’язок’ та ‘об’єкт, що потрібно зв’язати’:

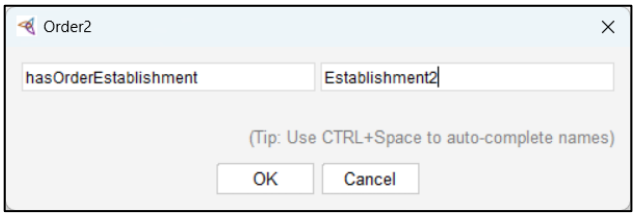

Рис. 3.15. Встановлення Object property assertions між екземплярами Order та Establishment

Тепер інстанс ‘Order2’ пов’язаний з інстансом ‘Establishment2’ зв’язком ‘hasOrderEstablishment’. Це означає, що замовлення було зроблено в закладі ‘Radius’(значення, яким ініціалізували поле ‘Title’ класу ‘Establishment’).

Таким чином задаємо всі необхідні зв’язки для існуючих інстансів.

3.2.6. Візуалізація онтології у модулі OntoGraf

Для візуалізації даної онтології у верхньому меню програми ‘Protégé’ перейти по шляху ‘Window/Views/Class views/OntoGraf’.

OntoGraf — це вбудований інструмент візуалізації онтологій у програмі Protégé. Він використовується для наочного представлення структури онтології, включно з класами, їх зв'язками та властивостями, що полегшує сприйняття та навігацію по складним онтологічним структурам.

Після корегування вигляду самої онтології вона виглядає ось таким чином:

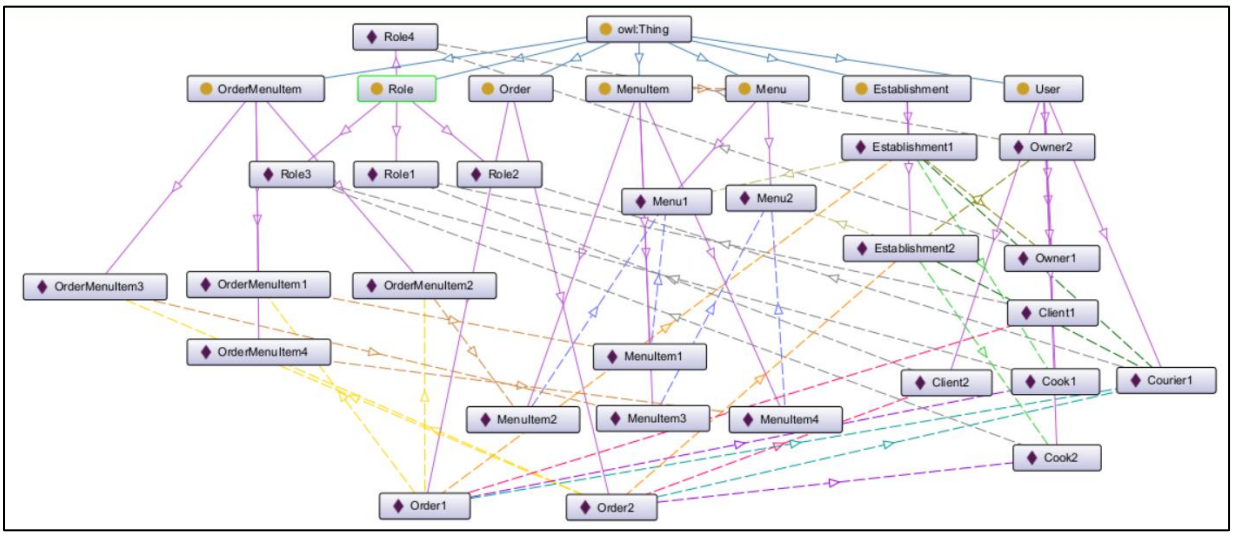

Рис. 3.16. Візуалізація онтології у вікні OntoGraf

- Прямокутники з жовтими колами всередині це класи;
- Прямокутники з фіолетовими ромбами всередині це екземпляри класу;
- Цілі стрілки це зв’язки приналежності інстансу до класу;
- Пунктирні стрілки це зв’язки між самими інстансами.

При наведенні (в даному випадку) на індивідуал, можна побачити поле зі всіма значеннями цього індивідуала та його зв’язками:

Рис. 3.17. Відображення екземпляра MenuItem1 в OntoGraf із об’єктними та дата-властивостями (назва страви, ціна та інгредієнти)

3.3 Реалізація функціоналу завантаження вже існуючих даних веб-застосунку в онтологію

Для подальшої роботи з онтологією потрібно зробити дві дії:

- Якщо існують попередні дані до існування онтології, реалізувати принцип портування даних в онтологію з БД сайту.
- Реалізацію принципу оновлення даних в онтології в режимі реального часу.

Для першої дії було обрано принцип створення файлу, що відтворює команду по запуску імпортування даних безпосередньо в онтологію, шляхом задання інструкції для імпортування.

Спершу для створення файлу в проекті Django в додатку, в будь-якому з додатків(apps) потрібно створити директорії management\commands\ файл_для_імпорту (в папках management та commands повинні бути файли _init_.py, для того, щоб проект вважав, що це службова папка).У файлі потрібно вказати шлях до репозиторію, який вміщує в собі онтологію, для динамічної роботи з даними(для репозиторію використовується GraphDB - це високоефективна, масштабована і надійна база даних графіків з підтримкою RDF і SPARQL). Після цього потрібно вказати інструкцію, завдяки двом мовам: Python(Django) та SPARQL(GraphDB), за якою відбуватиметься імпорт: `Назва класу в БД` - `Назва класу в онтологію`, в яку відбуватиметься імпорт, та зв’язки, на основі яких імпортуватимуться необхідні дані:

Код файлу для імпорту існуючих даних веб-застосунку в онтологію зображений в лістингу А.1.

Для виконання імпорту необхідно запустити команду в директорії знаходження додатку(не файлу, оскільки даний шлях вважається системним), в моєму випадку це ‘D:\Резерв+\GraduateWork(основа2)\mysite>’ та запустити команду ‘python manage.py load_to_ontology’. Після успішного виконання імпорту з’являється повідомлення ‘All data has been successfully loaded into the ontology.’(як передбачений механізм для сигналізації завершення дії).

Рис. 3.18. Вікно термінала з виконанням команди python manage.py load_to_ontology

3.4 Реалізація синхронізації даних між веб-застосунком та онтологією

Для реалізації принципу оновлення даних в онтології в режимі реального часу, використовуватиметься один з інструментів Django - Signals. Сигнали засікають дії, що відбуваються в БД(такі наприклад як post_save(включає в собі і створення об’єкту, та його оновлення) та pre_delete(викликається при видаленні об’єкта)). Даний інструмент використовуватиметься для задіювання SPARQL запитів в режимі реального часу, який активовуватиметься завдяки сигналам від БД, що оновлюватиме саму онтологію.

Даний код поміщатиметься в файл signals.py, який попередньо потрібно створити в кореневій директорії додатку(app):

Код файл signals.py зображений в лістингу А.2.

Відтепер онтологія буде оновлюватиметься в режимі реального часу завдяки використанню сигналів.

Використовуючи GraphDB можна отримати візуалізацію діаграми класів у круговому вигляді:

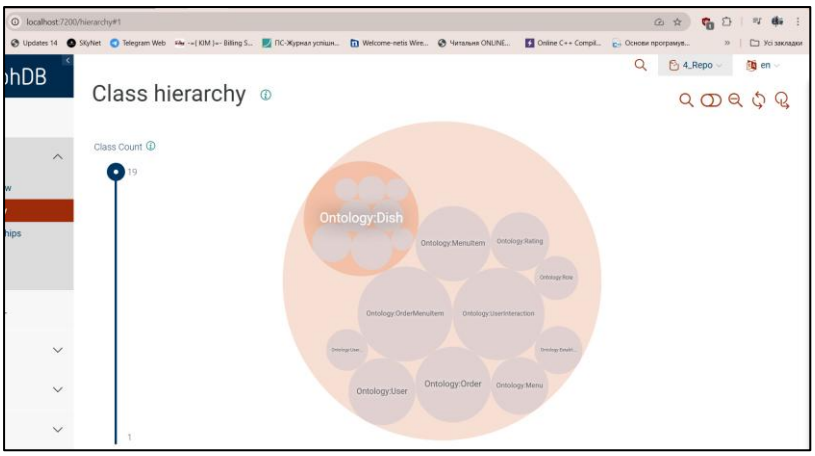

Рис. 3.19. Візуалізація ієрархії класів онтології в GraphDB у вигляді діаграми

3.5. Налаштування онтології для виведення рекомендацій

3.5.1. Формування поведінкових рекомендацій користувачу

Для отримання рекомендацій спершу я використовуватиму механізми Django, для того, щоб отримувати рекомендації, на основі дій користувача.

Є три класи моделі БД(для БД використовується SQLite3): Rating, UserInteraction, UserMenuInterest.

Реалізація класу Rating:

Лістинг 3.1.

```
class Rating(models.Model):
 user = models.ForeignKey(User, on_delete=models.CASCADE)
    menu_item = models.ForeignKey(MenuItems, on_delete=models.CASCADE)
    rating_value = models.IntegerField(default=0)
    created = models.DateTimeField(auto_now_add=True)
```

Клас(таблиця БД) має за ціль записати користувача, пункт меню, що оцінюється, та саму оцінку, виставлену користувачем.

Реалізація класу UserInteraction:

Лістинг 3.2.

```
class UserInteraction(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    menu_item = models.ForeignKey(MenuItems, on_delete=models.CASCADE)
    action_type = models.CharField(max_length=50, choices=[
        ('view', 'Перегляд'),
        ('purchase', 'Покупка'),
        ('Rate', 'Оцінювання'), ])
    timestamp = models.DateTimeField(auto_now_add=True)
```

Клас(таблиця БД) має за ціль записати користувача, пункт меню та дію, яку виконав користувач.

Реалізація класу UserMenuInterest:

Лістинг 3.3.

```
class UserMenuInterest(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    menu = models.ForeignKey(Menu, on_delete=models.CASCADE)
    interest_level = models.IntegerField(default=1)
    updated_at = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f"Interest of {self.user} in {self.menu}:
{self.interest_level}"
```

Клас(таблиця БД) має за ціль записати користувача, меню, яким користувався користувач та рівень інтересу користувача до даного меню, щоб опотім використовувати це меню, та його пункти в рекомендаціях.

Також для даного класу додатково було реалізовано функцію, яка вираховує рівень інтересу на основі виконаних дій користувачем, записаних в об’єктах класу UserInteraction:

Лістинг 3.4.

```
def update_interest_level(self):
        """

        Оновлює рівень інтересу до меню на основі взаємодій користувача.

        """
        interactions = UserInteraction.objects.filter(user=self.user,
menu_item__id_menu=self.menu)
        # Ваги для різних дій

        weights = {'view': 1.0,
            'purchase': 5.0,
            'Rate': 3.0,}
        # Розрахунок рівня інтересу

        total_interest = (sum(weights.get(interaction.action_type, 0) for
interaction in interactions))/3
```

```
        self.interest_level = total_interest # Обмеження рівня
        self.save()
```

Після правильної роботи з даними таблицями БД відбувається і саме формування рекомендацій. За це відповідає функція get_recommendations_for_user, яка відбирає меню авторизованого користувача, з яким він взаємодіяв, та на основі їхнього рівню інтересу, який є вирахуваним в класі UserMenuInterest, та результатом повертає рекомендовані страви з певного меню:

Лістинг 3.5.

```
def get_recommendations_for_user(user):

 # Get the menus the user is interested in

interested_menus=
UserMenuInterest.objects.filter(user=user).values_list('menu', flat=True)
    # Fetch menu items from those menus and annotate with the average rating

 recommended_items =
MenuItems.objects.filter(id_menu__in=interested_menus).annotate(
        avg_rating=Avg('rating__rating_value')  # Correct field for
avg_rating

    ).order_by('-avg_rating')[:7]  # Fetch top 2 items
    return recommended_items
```

3.5.2. Реєстрація взаємодій користувача для побудови історії взаємодій

Для роботи з функцією для рекомендацій потрібно реалізувати створення об’єктів UserInteraction на основі тригерних дій. Такими тригерними діями будуть натискання на блок , що відповідає за пункт меню, оцінювання пункту меню(натискаючи на одну з п’яти зірок в самому , що відповідає за пункт меню). Ці дії відбуваються в шаблоні ‘Меню’:

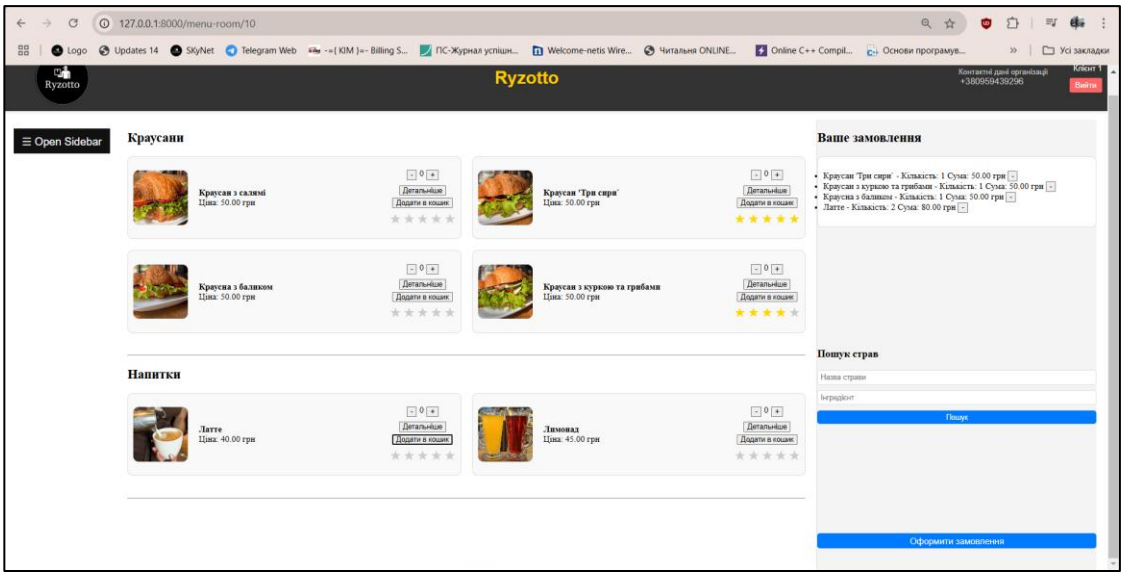

Рис. 3.20. Веб-сторінка шаблону ‘Меню’ з переліком страв

Також залишається дія купівлі. Безпосередньої покупки на сайті не реалізовано(не на часі), тому замість простого факту купівлі використовуватиметься механізм зміни статусу замовлення, початковим статусом якого є ‘Не оформлене замовлення’ та після обрання одного з типів замовлення(‘доставка’ або ‘на місці’) статус змінюється на ‘На розгляді’, що вважатиметься за покупку. Ця дія відбуваються в шаблоні ‘Кошик’:

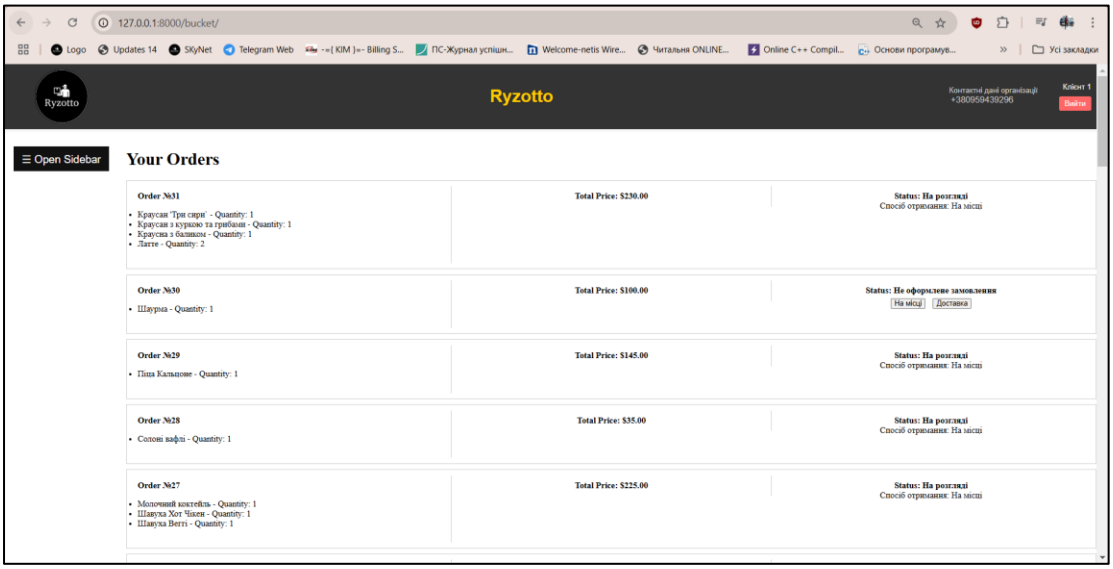

Рис. 3.21. Веб-сторінка шаблону ‘Кошик’ із переліком попередніх замовлень користувача

Для доказу, що необхідні об’єкти на основі дій користувача були створені, я виконав скрін з адміністративної панелі, на якому продемонстровані дані про виконані дії користувачем:

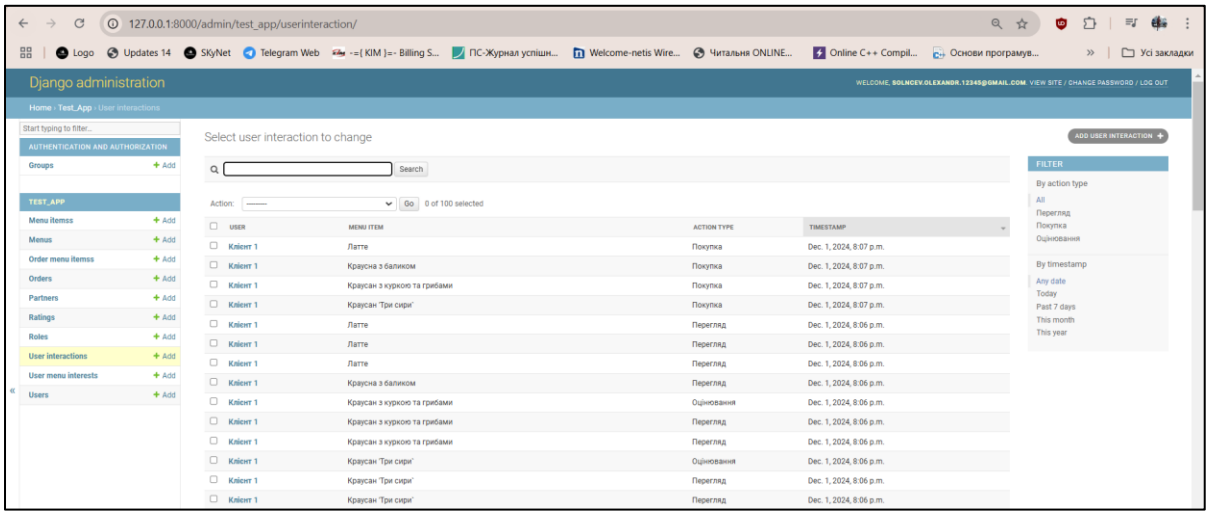

Рис. 3.22. Адміністративна панель Django з переліком об’єктів UserInteraction, що фіксують дії для окремих пунктів меню

3.5.3. Відображення поведінкових рекомендацій на головній сторінці веб-застосунку

Тепер для відображення рекомендацій на основі get_recommendations_for_user потрібно помістити дані цієї функції у в’ю, що відповідає за відображення сторінки, що відображатиме дані рекомендації. Для цього я обрав домашню сторінку, що відповідає за обрання користувачем закладу, з якого користувач хоче зробити замовлення.

Отож, спершу виконуємо порт даної функції з файлу models.py в файл views.py, після чого у в’ю що відповідає за відображення сторінки вставляємо механізм робити з функцією та дані в контекстний словник, для використання даних зі словника в шаблоні:

Лістинг 3.6.

```
from test_app.models import get_recommendations_for_user
```

```
@role_required(allowed_roles=['Користувач'])
def home(request):
    partners = Partner.objects.all()
    user_role = request.user.id_role.name if request.user.is_authenticated
and request.user.id_role else None
    roles = ["Партнер", "Кур'єр", "Кухар"]
    # Fetch recommendations
    recommended_items = get_recommendations_for_user(request.user) if
request.user.is_authenticated else []

    context = {
        'partners': partners,
        'user_role': user_role,
        'roles': roles,
        'recommended_items': recommended_items,
    }

    return render(request, 'test_app/Client/home.html', context)
```

Після цього в шаблон сторінки додаємо блок з наборів , який відображатиме рекомендації на основі дій користувача:

Лістинг 3.7.

```
<div class="recommended-wrapper">

  <h2>Рекомендовані страви на основі вашої активності</h2>
  <div class="recommended-container scroll-container">
    {% if recommended_items %}
      {% for item in recommended_items %}
      <a href="{% url 'menu-room' item.id_menu.id %}" class="recommended-
item">
        <img src="{{ item.photo.url }}" alt="{{ item.dishes }}" />
        <div class="item-info">
          <h3>{{ item.dishes }}</h3>
          <p>Ціна: {{ item.price }} грн</p>
        </div>
      </a>
```

```
      {% endfor %}
    {% else %}
      <p>Рекомендованих страв не знайдено.</p>
    {% endif %}
  </div>
</div>
```

Цей блок відображає рекомендації, які є результатом роботи функції get_recommendations_for_user:

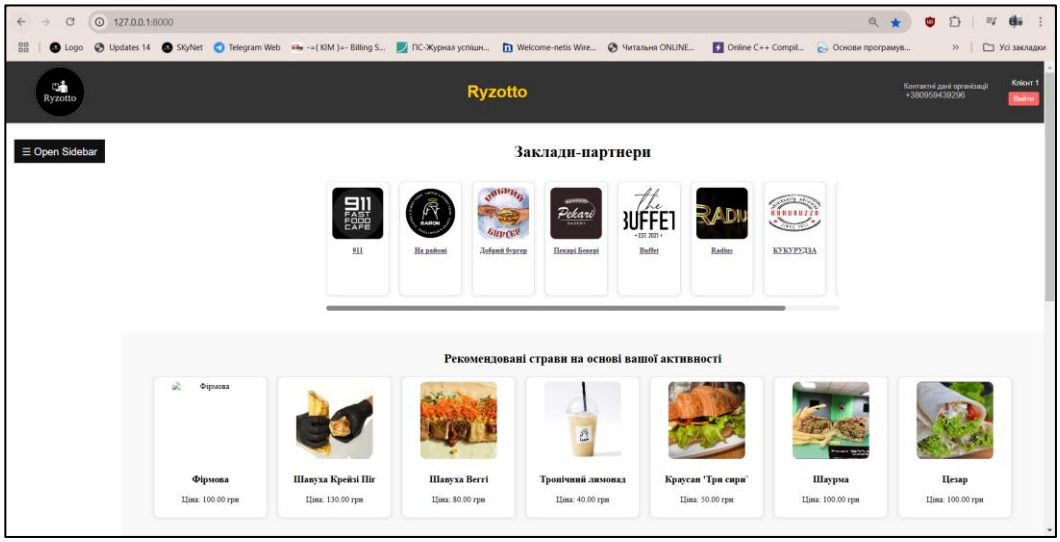

Рис. 3.23. Головна сторінка клієнтського інтерфейсу з переліком закладів-партнерів та блоком «Рекомендовані страви на основі вашої активності», що відображає результати роботи функції

3.5.4. Налаштування онтологічних зв'язків подібності між стравами (isSimilarTo)

Тепер розпочинаємо роботу з самою онтологією. Для реалізації рекомендацій на основі онтології було вирішено розробити рекомендації, що за основу беруть рекомендації на основі дій користувача, що діє за принципом подібності, або за принципом 'Якщо вам подобається страва 'А'(рекомендація на основі дій користувача), тоді вам може сподобатися страва 'Б'(страва з онтології)'. Для цього в онтологію було інтегровано новий зв'язок `isSimilarTo` з властивістю симетричності, який бере за основу(Domain) об'єкт класу MenuItem(пункт меню), та за область

значень(Range) бере теж об’єкт класу MenuItem(пункт меню, який може сподобатися по відношенню до того пункту меню, що був рекомендований на основі дій користувача).

Для цього було використано термінал GraphDB, в якому було запущено наступний код мовою SPARQL:

Лістинг 3.8.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX : <http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>

# Додавання властивості isSimilarTo як симетричної
INSERT DATA {
  :isSimilarTo rdf:type owl:ObjectProperty ;
               rdf:type owl:SymmetricProperty ;
               rdfs:domain :MenuItem ;
               rdfs:range :MenuItem .
}
```

В подальшому потрібно цим зв’язком зв’язати пункти меню між собою. Наприклад:

Співвідношення isSimilarTo для страв

1) **Шаурма і подібні страви:**

- ont:MenuItem_1 ("Шаурма") ↔ ont:MenuItem_24 ("Шавуха Крейзі Піг")
- ont:MenuItem_24 ↔ ont:MenuItem_26 ("Шавуха Хот Чікен")
- ont:MenuItem_26 ↔ ont:MenuItem_25 ("Шавуха Веггі")

2) **Піца:**

- ont:MenuItem_35 ("піца Бабусина з Фетою") ↔ ont:MenuItem_37 ("піца Капрічоза")
- ont:MenuItem_37 ↔ ont:MenuItem_50 ("Піца Маргарита")
- ont:MenuItem_50 ↔ ont:MenuItem_51 ("Піца Мексикана")
- ont:MenuItem_35 ↔ ont:MenuItem_36 ("Піца Кальцоне")

- ont:MenuItem_49 ("Піца Кватро Формаджі Россо") ↔ ont:MenuItem_51

3) Солодощі:

- ont:MenuItem_43 ("ЧІЗКЕЙК") ↔ ont:MenuItem_44 ("МОРОЗИВО СІМІФРЕДО")
- ont:MenuItem_22 ("Солодкі вафлі") ↔ ont:MenuItem_23 ("Солоні вафлі")
- ont:MenuItem_38 ("ГЛЯСЕ") ↔ ont:MenuItem_27 ("Молочний коктейль")

4) Напої:

- ont:MenuItem_34 ("Лимонад") ↔ ont:MenuItem_28 ("Тропічний лимонад")
- ont:MenuItem_33 ("Латте") ↔ ont:MenuItem_21 ("Айс латте")
- ont:MenuItem_39 ("Морс ягідний") ↔ ont:MenuItem_38

5) М’ясні страви:

- ont:MenuItem_46 ("Скумбрія з овочами") ↔ ont:MenuItem_47 ("Форель у вогняному панцері")
- ont:MenuItem_48 ("Салат теплий з телятиною та моцарелою") ↔ ont:MenuItem_17 ("Струнка Мар'яна")

Для цього був використаний наступний код:

Лістинг 3.9.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX : <http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>

INSERT DATA {
  # Шаурма

  :MenuItem_1 :isSimilarTo :MenuItem_24 .

  :MenuItem_24 :isSimilarTo :MenuItem_26 .

  :MenuItem_26 :isSimilarTo :MenuItem_25 .
```

```
  # Піца
  :MenuItem_35 :isSimilarTo :MenuItem_37 .
  :MenuItem_37 :isSimilarTo :MenuItem_50 .
  :MenuItem_50 :isSimilarTo :MenuItem_51 .
  :MenuItem_35 :isSimilarTo :MenuItem_36 .
  :MenuItem_49 :isSimilarTo :MenuItem_51 .

  # Солодощі
  :MenuItem_43 :isSimilarTo :MenuItem_44 .
  :MenuItem_22 :isSimilarTo :MenuItem_23 .
  :MenuItem_38 :isSimilarTo :MenuItem_27 .
  # Напої
  :MenuItem_34 :isSimilarTo :MenuItem_28 .
  :MenuItem_33 :isSimilarTo :MenuItem_21 .
  :MenuItem_39 :isSimilarTo :MenuItem_38 .
  # М’ясні страви
  :MenuItem_46 :isSimilarTo :MenuItem_47 .
  :MenuItem_48 :isSimilarTo :MenuItem_17 .
}
```

На рисунку 3.24 зображено утворення зв’язку між об’єктами, які були пов’язані вище:

	item1	
1	Ontology/MenuItem_1	Ontology/MenuItem_24
2	Ontology/MenuItem_17	Ontology/MenuItem_48
3	Ontology/MenuItem_21	Ontology/MenuItem_33
4	Ontology/MenuItem_22	Ontology/MenuItem_23
5	Ontology/MenuItem_23	Ontology/MenuItem_22
6	Ontology/MenuItem_24	Ontology/MenuItem_1
7	Ontology/MenuItem_24	Ontology/MenuItem_26
8	Ontology/MenuItem_25	Ontology/MenuItem_26
9	Ontology/MenuItem_26	Ontology/MenuItem_24
10	Ontology/MenuItem_26	Ontology/MenuItem_25
11	Ontology/MenuItem_27	Ontology/MenuItem_38
12	Ontology/MenuItem_28	Ontology/MenuItem_34
13	Ontology/MenuItem_33	Ontology/MenuItem_21
14	Ontology/MenuItem_34	Ontology/MenuItem_28
15	Ontology/MenuItem_35	Ontology/MenuItem_36
16	Ontology/MenuItem_35	Ontology/MenuItem_37
17	Ontology/MenuItem_36	Ontology/MenuItem_35
18	Ontology/MenuItem_37	Ontology/MenuItem_35
19	Ontology/MenuItem_37	Ontology/MenuItem_50
20	Ontology/MenuItem_38	Ontology/MenuItem_27
21	Ontology/MenuItem_38	Ontology/MenuItem_39
22	Ontology/MenuItem_39	Ontology/MenuItem_38
23	Ontology/MenuItem_43	Ontology/MenuItem_44
24	Ontology/MenuItem_44	Ontology/MenuItem_43
25	Ontology/MenuItem_46	Ontology/MenuItem_47
26	Ontology/MenuItem_47	Ontology/MenuItem_46
27	Ontology/MenuItem_48	Ontology/MenuItem_17
28	Ontology/MenuItem_49	Ontology/MenuItem_51
29	Ontology/MenuItem_50	Ontology/MenuItem_37
30	Ontology/MenuItem_50	Ontology/MenuItem_51
31	Ontology/MenuItem_51	Ontology/MenuItem_49
32	Ontology/MenuItem_51	Ontology/MenuItem_50

Рис. 3.24. Вигляд списку для зв’язку isSimilarTo

Тепер, якщо серед рекомендованих страв на основі дій користувача потраплятимуться ті, що в даному списку, користувач отримуватиме рекомендацію, побудовану на зв’язку з онтології.

3.5.5. Отримання онтологічних рекомендацій та їх інтеграція в інтерфейс користувача

Тепер потрібно розробити механізм для витягування даних з онтології та відображення рекомендацій користувачеві. Для цього спершу я створюю функцію для витягування даних з онтології ‘fetch_similar_item_ids’, яка реалізована інструментами Django та SPARQL, в середині якої відбувається запит до онтології, який відбувається на основі списку рекомендацій на основі дій користувача з функції ‘get_recommendations_for_user’. Після отримання необхідного ID з функції ‘get_recommendations_for_user’ надсилає запит до онтології, та звідти повертає ID об’єкту, який і буде використаний для рекомендації на основі онтології:

Лістинг 3.10.

```
def fetch_similar_item_ids(recommended_items):
    """
    Витягує ID подібних страв зі зв'язків isSimilarTo в онтології.
    """
    endpoint = "http://localhost:7200/repositories/4_Repo"
    sparql = SPARQLWrapper(endpoint)

    similar_item_ids = {}

    for item in recommended_items:
      item_uri =
f"http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#MenuItem_{item
.id}"

        query = f"""
      PREFIX :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
        SELECT ?similarItem
        WHERE {{
          <{item_uri}> :isSimilarTo ?similarItem .
        }}
        """
        sparql.setQuery(query)
        sparql.setReturnFormat(JSON)
        results = sparql.query().convert()
        similar_item_ids[item.id] = [
            result["similarItem"]["value"].split("#")[-
1].replace("MenuItem_", "")
            for result in results["results"]["bindings"]
        ]

    return similar_item_ids
```

Після цього результат дії даної функції приймає інша функція, що конвертує отримані дані з функції в правильне форматування для виведення рекомендації:

Лістинг 3.11.

```
def get_enhanced_recommendations(user):
    """
    Отримує рекомендації для користувача разом із подібними елементами на основі isSimilarTo.
    Дані подібних елементів отримуються з БД сайту.
    """
    if not user.is_authenticated:
        return []

    # Початкові рекомендації
    recommended_items = get_recommendations_for_user(user)

    # Отримання подібних ID зі зв'язків онтології
    similar_item_ids = fetch_similar_item_ids(recommended_items)

    # Отримання даних з БД сайту
    recommendations = []
    for item in recommended_items:
        similar_ids = similar_item_ids.get(item.id, [])
        similar_items_from_db = MenuItems.objects.filter(id__in=similar_ids)  # Запит до БД

        for similar_item in similar_items_from_db:  # Формуємо пари рекомендацій
            recommendations.append({
                "main_item": {
                    "id": item.id,
                    "name": item.dishes,
                    "price": item.price,
                    "photo": item.photo.url,
                },
                "similar_item": {
                    "id": similar_item.id,
                    "name": similar_item.dishes,
                    "price": similar_item.price,
                    "photo": similar_item.photo.url,
                }
```

```
            })

    return recommendations
```

Функція використовує ID отримане з функції ‘fetch_similar_item_ids’ для правильного отримання рекомендацій після сортування на основі даного ID.

Далі потрібно помістити дані функції ‘get_enhanced_recommendations’ у в’ю, що відповідає за відображення сторінки, що відображатиме рекомендації, що отримані завдяки результатам роботи з онтологією. Для цього я обрав домашню сторінку, що відповідає за обрання користувачем закладу, з якого користувач хоче зробити замовлення.

У в’ю, що відповідає за відображення сторінки вставляємо механізм робити з функцією та дані в контекстний словник, для використання даних зі словника в шаблоні:

Лістинг 3.12.

```
@role_required(allowed_roles=['Користувач'])
def home(request):
    partners = Partner.objects.all()
    user_role = request.user.id_role.name if request.user.is_authenticated
and request.user.id_role else None
    roles = ["Партнер", "Кур'єр", "Кухар"]
    # Fetch recommendations
    recommended_items = get_recommendations_for_user(request.user) if
request.user.is_authenticated else []
    recommendations = get_enhanced_recommendations(request.user) if
request.user.is_authenticated else []
    context = {
        'partners': partners,
        'user_role': user_role,
        'roles': roles,
        'recommended_items': recommended_items,
        'recommendations': recommendations,
    }
    return render(request, 'test_app/Client/home.html', context)
```

Після цього в шаблон сторінки додаємо блок з наборів , який відображатиме рекомендації на основі дій з онтологією:

Лістинг 3.13.

```
<div class="recommended-wrapper">
  <h2>Рекомендації на основі подібності</h2>
  <div class="recommendations-container">
    {% if recommendations %}
      {% for recommendation in recommendations %}
      <div class="recommendation-pair">
        <div class="recommendation-row">
          <div class="recommendation-item">
            <img src="{{ recommendation.main_item.photo }}" alt="{{
recommendation.main_item.name }}">
            <h3>{{ recommendation.main_item.name }}</h3>
            <p>Ціна: {{ recommendation.main_item.price }} грн</p>
          </div>
          <span class="arrow">&rarr;</span>
          <div class="recommendation-item">
            <
           img src="{{ recommendation.similar_item.photo }}" alt="{{
     recommendation.similar_item.label }}">

            <h3>{{ recommendation.similar_item.label }}</h3>
            <h3>{{ recommendation.similar_item.name }}</h3>
            <p>Ціна: {{ recommendation.similar_item.price }} грн</p>
          </div>
        </div>
      </div>
      {% endfor %}
    {% else %}
      <p>Рекомендацій не знайдено.</p>
    {% endif %}
  </div>
</div>
```

Цей блок відображає рекомендації, які є результатом роботи функції get_enhanced_recommendations:

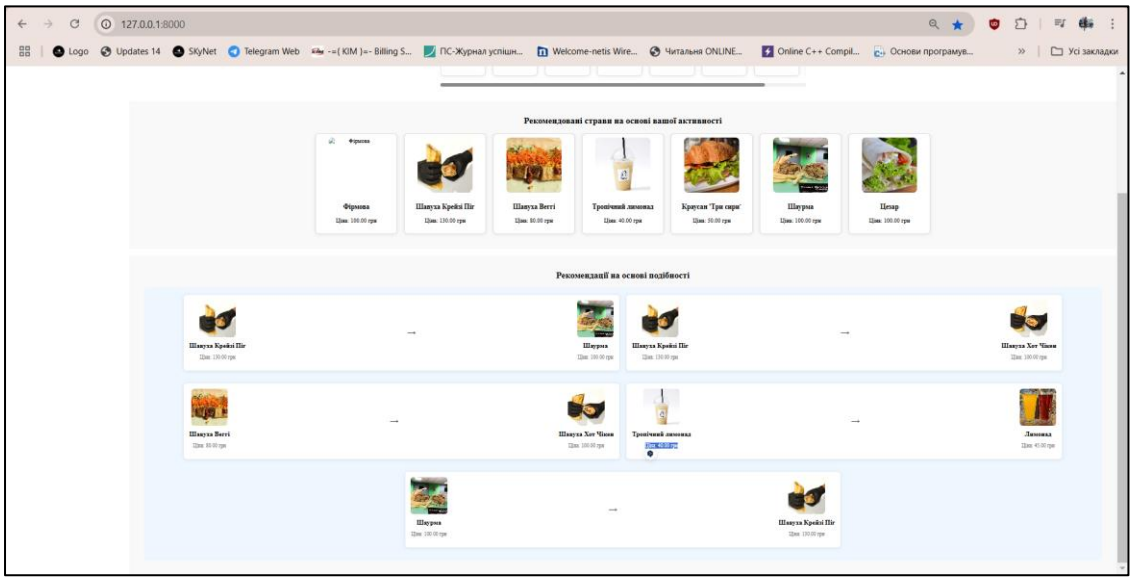

Рис. 3.25 Головна сторінка з двома блоками рекомендацій, де відображаються пари позицій меню, згенеровані функцією get_enhanced_recommendations

3.6. Тестування рекомендаційної системи

Тести, що реалізовані в даному пункті потрібні для того, щоб показати, що вся рекомендаційна система працює разом, як одна цілісна схема. Вони перевіряють, чи правильно поєднуються дані з бази, модуль поведінкових рекомендацій і онтологія, а також чи узгоджуються їхні результати між собою.

Основний сенс цих тестів у тому, щоб наочно продемонструвати три речі: система формує топ-5 рекомендацій з урахуванням реальної поведінки користувача; потім доповнює цей список ще п'ятьма стравами з онтології; і, нарешті, змінює обидва списки, коли змінюється історія замовлень. Це дає просте, але переконливе підтвердження, що розроблена модель рекомендацій не лише описана в теорії, а й реально працює в коді.

3.6.1. Мета кінцевого інтеграційного тесту

Мета тестів - показати, що:

1. Система коректно формує топ-5 рекомендацій з урахуванням поведінки користувача;

2. Ці рекомендації коректно доповнюються п'ятьма пов'язаними стравами з онтології;

3. Після зміни поведінки користувача змінюється як базовий список А (топ-5), так і похідний онтологічний список В, що підтверджує узгоджену роботу обох шарів.

Клас RecommendationOntologyTests реалізує інтеграційні тести гібридної рекомендаційної системи, у якій поєднуються два рівні:

- Рівень А - поведінкові рекомендації на основі даних реляційної БД (історія замовлень, переглядів та оцінок користувача);
- Рівень В - семантичне розширення рекомендацій на основі онтології в GraphDB (зв'язки isSimilarTo між стравами).

3.6.2. Клас тестів RecommendationOntologyTests

Лістинг 3.14

```
class RecommendationOntologyTests(TestCase):
    """
    Інтеграційні тести рекомендаційної системи:

    A - базові рекомендації з Django-БД,
    B - розширені рекомендації з урахуванням онтології (GraphDB, репозиторій
Test).
    """
```

Цей клас задає рамку інтеграційних тестів. Використання django.test.TestCase забезпечує автоматичне створення й очищення тестової БД перед запуском тестів, що дозволяє відтворити завершені сценарії "від дій користувача до рекомендацій". Коментар у docstring чітко фіксує дихотомію А/В і визначає, що в тестах буде перевірятися взаємодія між реляційною БД та онтологічним репозиторієм.

3.6.3. Статична ініціалізація даних setUpTestData

Лістинг 3.15

```
@classmethod
    def setUpTestData(cls):
        # 1. Роль клієнта
        cls.role_customer = Role.objects.create(name="Customer")

        # 2. Власник закладу
        cls.owner = User.objects.create_user(
            email="owner@test.com",
            name="Owner",
            surname="Test",
            patronymic="",
            phone="+380000000000",
            password="owner_password",
            id_role=cls.role_customer,
        )

        # 3. Партнер (заклад)
        cls.partner = Partner.objects.create(
            title="Тестовий заклад",
            owner=cls.owner,
            photo="partner_photos/test_partner.jpg",
        )

        # 4. Меню
        cls.menu = Menu.objects.create(
            title="Тестове меню",
            id_menu=cls.partner,
        )
        # 5. Тестові позиції меню з PK 1..10 (синхронні до
MenuItem_1..MenuItem_10 в онтології)
        cls.menu_items = {}
        items = [
            (1,  "Шаурма"),
            (2,  "Цезар"),
            (3,  "Фірмова"),
            (4,  "По чом в Одесі"),
```

```
            (5,  "Струнка Мар'яна"),
            (6,  "БД"),
            (7,  "Чорна гора"),
            (8,  "Айс лате"),
            (9,  "Солодкі вафлі"),
            (10, "Солоні вафлі"),
        ]
        for item_id, dish_name in items:
            obj = MenuItems.objects.create(
                id=item_id,
                dishes=dish_name,
                price=100,
                ingredients="тестові інгредієнти",
                id_menu=cls.menu,
                photo="menu_photos/test.jpg",
            )
            cls.menu_items[item_id] = obj
```

Метод setUpTestData виконується один раз для всього класу й створює спільний для тестів статичний контекст: роль "клієнта", власника закладу, одного партнера (заклад) і одного меню, а також 10 позицій меню з наперед визначеними первинними ключами 1..10.

Важливий момент: значення поля id у моделі MenuItems синхронізуються з суфіксами ресурсів MenuItem_1..MenuItem_10 в онтології GraphDB. Це дозволяє при отриманні URI виду ...#MenuItem_7 однозначно знайти відповідну страву в реляційній БД за її PK=7. Саме завдяки цій відповідності тест може перевіряти, що онтологічні рекомендації В узгоджені з базовими об'єктами Django.

3.6.4. Підготовка динамічного контексту setUp

Лістинг 3.16

```
def setUp(self):
        # Очищаємо тільки динамічні сутності, пов'язані з поведінкою
користувача
        Order.objects.all().delete()
        OrderMenuItems.objects.all().delete()
```

```
        UserInteraction.objects.all().delete()
        Rating.objects.all().delete()
        UserMenuInterest.objects.all().delete()

        # Тестовий користувач-клієнт
        self.test_user = User.objects.create_user(
            email="customer@test.com",
            name="Customer",
            surname="Test",
            patronymic="",
            phone="+380000000001",
            password="customer_password",
            id_role=self.role_customer,
        )
```

Метод setUp виконується перед кожним тестом. Він очищає саме ті таблиці, які відображають динамічну поведінку користувача (замовлення, позиції замовлень, взаємодії, рейтинги, рівні інтересу). Після цього створюється новий тестовий користувач.

Таким чином, кожен тест починається з «чистої» історії взаємодій, що дозволяє в першому тесті змоделювати початкову поведінку, а в другому - радикально змінити цю поведінку й спостерігати, як система реагує на зміну даних.

3.6.5. Формування замовлення create_order_with_items

Лістинг 3.17

```
def create_order_with_items(self, user, items_spec):
        """
        Створення замовлення з позиціями.
        items_spec: список кортежів (menu_item_id, quantity)
        """
        order = Order.objects.create(
            status="Delivered",
            get_type="delivery",
            address="Тестова адреса",
            partner=self.partner,
            client=user,
```

```
            courier=None,
            cook=None,
        )

        for menu_item_id, qty in items_spec:
            omi = OrderMenuItems.objects.create(
                menu_item=self.menu_items[menu_item_id],
                quantity=qty,
            )
            order.menu_items.add(omi)

        return order
```

Цей допоміжний метод створює сутності замовлення та його позиції. Параметр items_spec задає, які страви і в якій кількості замовив користувач.

Цей метод моделює «явні» покупки користувача, які є важливими сигналами для обчислення його інтересів і, відповідно, впливають на впорядкування рекомендацій рівня А. Використання цього методу в обох тестах забезпечує відтворюваність сценаріїв поведінки.

3.6.6. Моделювання взаємодій і рейтингів create_interactions_and_ratings

Лістинг 3.18

```
def create_interactions_and_ratings(self, user, interactions_spec):
        """
        interactions_spec: список словників:
        {
            "menu_item_id": 1,
            "actions": ["view", "purchase", "rate"],
            "rating": 5  # опційно
        }
        """
        for spec in interactions_spec:
            menu_item = self.menu_items[spec["menu_item_id"]]

            for action in spec.get("actions", []):
                UserInteraction.objects.create(
```

```
                user=user,
                menu_item=menu_item,
                action_type=action,
                timestamp=timezone.now(),
            )

        if "rating" in spec:
            Rating.objects.create(
                user=user,
                menu_item=menu_item,
                rating_value=spec["rating"],
            )
```

Цей метод описує повну поведінку користувача: для кожної страви створюються події перегляду, покупки й оцінювання, а також числовий рейтинг. Завдяки цьому тест не обмежується лише фактами замовлень, а враховує різні типи взаємодій, що ближче до реального використання системи.

У тесті ці дані потім перетворюється в рівень інтересу до меню й впливають на те, які саме п’ять страв потраплять до топ-5 списку А. Це демонструє, що система рекомендацій враховує комплексну поведінку, а не лише поодинокі покупки.

3.6.7. Обчислення інтересу до меню recalculate_user_interest

Лістинг 3.19

```
def recalculate_user_interest(self, user):
        """
        Примусовий перерахунок UserMenuInterest для одного меню
        на основі UserInteraction (спрощена версія логіки з моделі).
        """
        interactions = UserInteraction.objects.filter(
            user=user,
            menu_item__id_menu=self.menu,
        )

        weights = {
            "view": 1.0,
            "purchase": 5.0,
```

```
        "rate": 3.0,
    }

    total_interest = 0
    for interaction in interactions:
        total_interest += weights.get(interaction.action_type, 0)

    umi, _ = UserMenuInterest.objects.get_or_create(

        user=user,
        menu=self.menu,
        defaults={"interest_level": 1},

    )

    umi.interest_level = int(total_interest)
    umi.save()
```

Цей блок реалізує алгоритм обчислення «інтересу» користувача до меню, який агрегує всі взаємодії з використанням ваг для різних типів дій. Результат записується в модель UserMenuInterest.

Функція get_recommendations_for_user надалі використовує цей показник при формуванні рекомендацій. В рамках тесту це ключова ланка, яка пов’язує сирові події (перегляди, покупки, рейтинги) з кінцевим списком A (топ-5 рекомендацій), що дає змогу наочно продемонструвати, як поведінка користувача впливає на отриманий список страв.

3.6.8. Тест 1 - початкові рекомендації та онтологічні зв’язки

Код тесту №1 зображений в лістингу А.3.

Цей метод реалізує перший інтеграційний сценарій.

Послідовність кроків:

1. Формується початкова історія замовлень та взаємодій, орієнтована на страви 1-3.

2. Обчислюється UserMenuInterest, після чого викликається get_recommendations_for_user, формується список A і виводиться в консоль лише топ 5 назв страв.

3. Викликається get_enhanced_recommendations,яка, використовуючи fetch_similar_item_ids, отримує з GraphDB пари «рекомендована страва → схожа страва». Топ 5 таких пар виводиться у форматі A: <назва> → B: <назва>.Через асерти перевіряється:

- непорожність A і B;
- належність main_item.id до множини рекомендацій A;
- наявність усіх similar_item.id у таблиці MenuItems.

Цей блок демонструє **базову коректність** гібридного підходу: онтологія не породжує «чужі» елементи, а лише семантично розширює вже знайдені поведінкові рекомендації.

3.6.9. Тест 2 -зміна поведінки та оновлення рекомендацій

Код тесту №2 зображений в лістингу А.4.

Другий тест ілюструє **адаптивність** системи.

Спочатку формується початкова історія (аналогічна до першого тесту, але з меншим набором дій), отримується список A1 (топ-5) і виводиться. Потім повністю скидається історія й задається нова поведінка користувача, спрямована на інші страви (4,5,9,10); обчислюється A2 та виводиться оновлений топ-5.

За допомогою асертів фіксується, що:

- множина рекомендацій реально змінилася (A1 ≠ A2);
- у новому списку з’явилися очікувані ID.

Після цього знову викликається get_enhanced_recommendations, формується B2 (топ-5 онтологічних пар A2 → B2) та виводиться. Перевірка існування similar_item у MenuItems гарантує, що онтологія коректно працює з новим контекстом.

Таким чином, цей тест демонструє, що зміна поведінки користувача призводить до узгодженої зміни як поведінкових, так і онтологічних рекомендацій, що є ключовою вимогою до гібридних рекомендаційних систем у дипломній роботі.

3.6.10. Результат виконання тестування

Результати тестування показали, що розроблена рекомендаційна система коректно виконує заданий робочий сценарій. У першому тесті система формує послідовний топ-5 рекомендацій на основі історії взаємодій користувача, а потім успішно розширює цей список п’ятьма стравами, отриманими з онтології за зв’язками подібності. У другому тесті продемонстровано, що після зміни поведінки користувача змінюється склад базових рекомендацій і відповідним чином оновлюється онтологічно розширений список, що підтверджується успішним проходженням обох тестів без помилок.

Вигляд отриманого результату виконання тестів:

```
=== ТЕСТ 1: Початкові рекомендації та онтологічні зв'язки ===

A) Базові рекомендації Django (топ-5, тільки назви):
  A: Шаурма
  A: Цезар
  A: Фірмова
  A: По чом в Одесі
  A: Струнка Мар'яна

B) Розширені рекомендації (топ-5, A(рекомендації на основі дій користувача) → B(рекомендації на основі онтологічних зв'язків)):
  A: Шаурма → B: Фірмова
  A: Шаурма → B: БД
  A: Шаурма → B: Чорна гора
  A: Цезар → B: По чом в Одесі
  A: Цезар → B: Струнка Мар'яна
```

```
[ПІДСУМОК ТЕСТУ 1] Топ-5 базових рекомендацій (A) та топ-5 онтологічно
розширених рекомендацій (B) сформовані коректно: B отримано шляхом
семантичного розширення A за зв'язками isSimilarTo.
.
=== ТЕСТ 2: Зміна поведінки користувача та рекомендацій ===

A1) Початкові базові рекомендації Django (топ-5):

  A1: Шаурма
  A1: Цезар
  A1: Фірмова
  A1: По чом в Одесі
  A1: Струнка Мар'яна

A2) Оновлені базові рекомендації Django після зміни поведінки (топ-5):
  A2: По чом в Одесі
  A2: Струнка Мар'яна
  A2: Солодкі вафлі
  A2: Солоні вафлі
  A2: Шаурма

B2) Розширені рекомендації після зміни поведінки (топ-5, A2 → B2):
A1) Початкові базові рекомендації Django (топ-5):

  A1: Шаурма
  A1: Цезар
  A1: Фірмова
  A1: По чом в Одесі
  A1: Струнка Мар'яна

A2) Оновлені базові рекомендації Django після зміни поведінки (топ-5):

  A2: По чом в Одесі
  A2: Струнка Мар'яна
  A2: Солодкі вафлі
  A2: Солоні вафлі
  A2: Шаурма
```

```
B2) Розширені рекомендації після зміни поведінки (топ-5, A2 → B2):

  A2: По чом в Одесі → B2: Цезар
  A2: По чом в Одесі → B2: Струнка Мар'яна
  A2: Струнка Мар'яна → B2: Цезар
  A2: Струнка Мар'яна → B2: По чом в Одесі
  A2: Солодкі вафлі → B2: Айс латте

[ПІДСУМОК ТЕСТУ 2] Після зміни історії взаємодій змінився склад топ-5 базових
рекомендацій (A1 → A2), а також відповідний топ-5 онтологічно розширених
рекомендацій (B2), що підтверджує узгоджену роботу поведінкової та
онтологічної частин системи.
```

Також на рисунку 3.26 можна побачити вигляд отриманого результату виконання тестів в IDE Visual Studio:

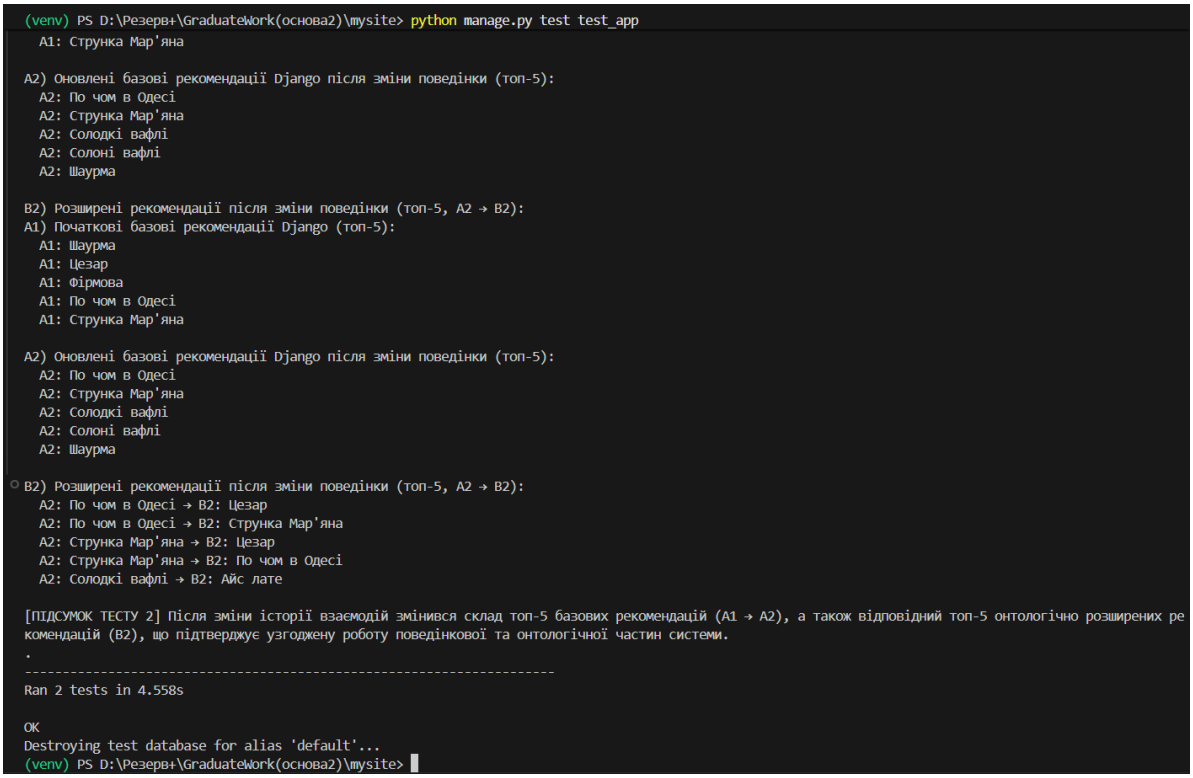

Рис. 3.26. Вигляд отриманого результату виконання тестів в IDE Visual Studio

3.7. Висновок до розділу

У розділі 3 представлено практичну реалізацію онтологічного підходу для веб-

додатку «маркетплейс – логістичне рішення для закладів харчування», починаючи з детального аналізу предметної області: користувачі та ролі, заклади, меню, страви, замовлення, логістика та взаємодії. На основі цієї моделі в редакторі Protégé було створено онтологію, розроблено класи (User, Establishment, Menu, MenuItem, Order тощо), властивості та обмеження, після чого онтологію було завантажено в GraphDB як граф знань RDF.

Описано алгоритм та програмну реалізацію інтеграції між додатком Django, реляційною базою даних та GraphDB, зокрема механізм формування RDF-трійок, виконання SPARQL-запитів та побудову підсистеми рекомендацій, яка поєднує поведінкові дані із семантичними зв'язками (наприклад, isSimilarTo між стравами). Результати тестування демонструють коректність передачі даних, узгоджену роботу поведінкової та онтологічної частин, а також здатність системи генерувати онтологічно розширені рекомендації, що підтверджує практичну ефективність запропонованого підходу.

ВИСНОВКИ

У магістерській роботі вирішується проблема інтеграції онтологій у веб-застосунок торговельної платформи для закладів громадського харчування з використанням стеку Django, GraphDB, RDF/OWL та SPARQL. Теоретична частина формує цілісне бачення ролі онтологій у представленні знань: розглядаються формальні основи RDF/RDFS/OWL, класифікація онтологій, принципи онтологічного моделювання предметної області та методи використання графа знань для семантичного пошуку, персоналізації та рекомендацій у веб-системах. На цій основі обґрунтовано вибір онтологічного підходу як засобу подолання обмежень традиційного реляційного моделювання та підвищення узгодженості та повторного використання знань у розподіленому середовищі.

Окремо показано, що поєднання реляційних та графових моделей даних дозволяє здійснювати як надійну транзакційну обробку, так і гнучкий семантичний аналіз. Розроблені моделі відображення таблиць у онтологічні класи, алгоритми початкової міграції та синхронізації даних на основі подій, а також запропоновані варіанти архітектурної інтеграції демонструють, як онтологічний рівень може розвиватися разом з веб-застосунок, не вимагаючи радикальної реструктуризації існуючої інфраструктури. Це створює передумови для подальшого розширення системи за рахунок нових типів знань, додаткових ролей користувачів або сервісів без порушення цілісності даних.

Практична частина роботи показала, що онтологічну модель можна інтегрувати в реальний веб-додаток без руйнування існуючої архітектури: було побудовано OWL-онтологію торговельної сфери, реалізовано механізми передачі даних та синхронізації між реляційною базою даних та GraphDB, а також розроблено підсистему рекомендацій, яка поєднує поведінкові дані із семантичними зв'язками між стравами та закладами. Результати тестування підтвердили коректність алгоритмів синхронізації та здатність системи генерувати онтологічно покращені рекомендації, що покращує якість обслуговування без суттєвих змін базової логіки веб-додатку; таким чином, було досягнуто мети створення моделей, методів та алгоритмів для ефективної інтеграції онтологій у веб-додатки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Berners-Lee, T., Hendler, J., Lassila, O. “The Semantic Web”, Scientific American, 284(5):34-43, 2001.
2. Hitzler, P., Krötzsch, M., Rudolph, S. “Foundations of Semantic Web Technologies”. Chapman & Hall/CRC, 2009.
3. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.). “The Description Logic Handbook: Theory, Implementation, and Applications”. Cambridge University Press, 2003.
4. W3C. “RDF 1.1 Concepts and Abstract Syntax”, W3C Recommendation, 25 February 2014. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/rdf11-concepts/
5. W3C. “RDF Schema 1.1”, W3C Recommendation, 25 February 2014. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/rdf-schema/
6. W3C. “RDF 1.2 Schema”, W3C Working Draft, 17 December 2025. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/rdf12-schema/
7. W3C. “OWL 2 Web Ontology Language Primer (Second Edition)”, W3C Recommendation, 11 December 2012. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/owl2-primer/
8. W3C. “OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax”, W3C Recommendation, 11 December 2012. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/owl2-syntax/
9. Motik, B., Patel-Schneider, P., Parsia, B. et al. “OWL 2 Web Ontology Language - Quick Reference Guide”. W3C Note, 27 October 2009. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/owl2-quick-reference/
10. Horridge, M., Knublauch, H., Rector, A., Stevens, R., Wroe, C. “A Practical Guide To Building OWL Ontologies Using Protégé and CO-ODE Tools”, University of

Manchester, 2004. Електронний ресурс. - Режим доступу: https://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html

11. Noy, N.F., McGuinness, D.L. “Ontology Development 101: A Guide to Creating Your First Ontology”, Stanford KSL Technical Report KSL-01-05, 2001. Електронний ресурс. - Режим доступу: https://protege.stanford.edu/publications/ontology_development/ontology101.pdf
12. W3C. “SPARQL 1.1 Query Language”, W3C Recommendation, 21 March 2013. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/sparql11-query/
13. W3C. “SPARQL 1.1 Update”, W3C Recommendation, 21 March 2013. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/sparql11-update/
14. Ontotext. “Using the RDF4J REST API — GraphDB 11.1 Documentation”, 2025. Електронний ресурс. - Режим доступу: https://graphdb.ontotext.com/documentation/11.1/rdf4j-rest-api.html
15. Ontotext. “Data Modeling with RDF(S) — GraphDB Documentation”, 2024. Електронний ресурс. - Режим доступу: https://graphdb.ontotext.com/documentation/11.1/rdfs.html
16. Ontotext. “GraphDB JavaScript Client (graphdb.js) - README”, 2019. Електронний ресурс. - Режим доступу: https://github.com/Ontotext-AD/graphdb.js
17. Bonduel, M. “RDF Triplestores and SPARQL Endpoints”, LDAC Summer School Lecture Notes, 2019. Електронний ресурс. - Режим доступу: https://linkedbuildingdata.net/ldac2019/summerschool/files/07_Bonduel_triplestores_SPARQL_endpoints.pdf
18. Django Software Foundation. “Django 5.x Documentation: Models and the Django ORM”. Електронний ресурс. - Режим доступу: https://docs.djangoproject.com/

19. Django Software Foundation. “Signals — Django Documentation”. Електронний ресурс. - Режим доступу: https://docs.djangoproject.com/en/stable/topics/signals/

20. Librarian Studies. “Semantic Web and Linked Data”, Librarianship Studies Blog, 2020. Електронний ресурс. - Режим доступу: https://www.librarianshipstudies.com/2020/05/semantic-web-and-linked-data.html

21. Klyne, G., Carroll, J.J. “Resource Description Framework (RDF): Concepts and Abstract Syntax”, W3C Recommendation, 10 February 2004. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/rdf-concepts/

22. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F. “From SHIQ and RDF to OWL: The Making of a Web Ontology Language”, Journal of Web Semantics, 1(1):7-26, 2003.

23. Antoniou, G., van Harmelen, F. “A Semantic Web Primer”. 3rd ed., MIT Press, 2012.

24. Guarino, N., Oberle, D., Staab, S. “What Is an Ontology?”, in Staab, S., Studer, R. (eds.), Handbook on Ontologies, Springer, pp. 1-17, 2009.

25. Studer, R., Benjamins, V.R., Fensel, D. “Knowledge Engineering: Principles and Methods”, Data & Knowledge Engineering, 25(1-2):161-197, 1998.

26. Staab, S., Studer, R. (eds.). “Handbook on Ontologies”. 2nd ed., Springer, 2009.

27. Bizer, C., Heath, T., Berners-Lee, T. “Linked Data - The Story So Far”, International Journal on Semantic Web and Information Systems, 5(3):1-22, 2009.

28. Gandon, F., Schreiber, G. “RDF 1.1 XML Syntax”, W3C Recommendation, 25 February 2014. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/rdf-syntax-grammar/

29. Hogan, A. et al. “Knowledge Graphs”, ACM Computing Surveys, 54(4):1-37, 2021.

30. Papazoglou, M.P., van den Heuvel, W.J. “Service Oriented Architectures: Approaches, Technologies and Research Issues”, The VLDB Journal, 16(3):389-415, 2007.

31. Burke, R. “Hybrid Recommender Systems: Survey and Experiments”, User Modeling and User-Adapted Interaction, 12(4):331-370, 2002.

32. Adomavicius, G., Tuzhilin, A. “Toward the Next Generation of Recommender Systems: A Survey of the State of the Art and Possible Extensions”, IEEE Transactions on Knowledge and Data Engineering, 17(6):734-749, 2005.

33. Ricci, F., Rokach, L., Shapira, B. (eds.). “Recommender Systems Handbook”. 2nd ed., Springer, 2015.

34. Al-Mubaid, H. “A Comprehensive Survey on Ontology-Based Recommender Systems”, International Journal of Computer Applications, 176(1):1-8, 2020.

35. Cánovas-Izquierdo, J.L., Cabot, J. “On the Use of RDF for the Management of Model Versions”, in Proceedings of the 2012 ICSE Workshop on Modeling in Software Engineering (MiSE), pp. 21-26, 2012.

36. Prud’hommeaux, E., Seaborne, A. “SPARQL Query Language for RDF”, W3C Recommendation, 15 January 2008. Електронний ресурс. - Режим доступу: https://www.w3.org/TR/rdf-sparql-query/

37. Garijo, D., Gil, Y. “A New Approach for Publishing Scientific Workflows: Research Objects and RO-Hub”, in Proceedings of IEEE eScience 2013, pp. 199-206, 2013.

38. Carroll, J.J. et al. “Jena: Implementing the Semantic Web Recommendations”, in Proceedings of the 13th International World Wide Web Conference (WWW), pp. 74-83, 2004.

39. Broekstra, J., Kampman, A., van Harmelen, F. “Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema”, in Proceedings of the International Semantic Web Conference (ISWC), pp. 54-68, 2002.

40. Buil-Aranda, C., Hogan, A., Umbrich, J., Vandenbussche, P-Y. “SPARQL Web Querying Infrastructure: Ready for Action?”, in Proceedings of the 12th International Semantic Web Conference (ISWC), pp. 277-293, 2013.

ДОДАТКИ

Додаток А

Лістинг А.1. Файл для імпорту існуючих даних веб-застосунку в онтологію даних

```
from django.core.management.base import BaseCommand
from test_app.models import (
    Role, User, Partner, Menu, MenuItems,
    Order, OrderMenuItems,
    UserMenuInterest, UserInteraction, Rating,
)
from SPARQLWrapper import SPARQLWrapper, POST


class Command(BaseCommand):
    help = 'Load all existing data from the database into the ontology'

    def handle(self, *args, **kwargs):
        sparql =
SPARQLWrapper("http://localhost:7200/repositories/4_Repo/statements")
        sparql.setMethod(POST)

        def execute_query(query):
            sparql.setQuery(query)
            sparql.query()

        # 1. Перенос даних про ролі
        # реалізація функціоналу переносу даних ролей у онтологію

        # 2. Перенос даних про користувачів (User)
        for user in User.objects.all():
            query = f"""
                PREFIX :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
                PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
                INSERT {{
                    :User_{user.id} a :User ;
                        :Name "{user.name}"^^xsd:string ;
                        :Surname "{user.surname}"^^xsd:string ;
                        :Patronymic "{user.patronymic or ''}"^^xsd:string ;
                        :Email "{user.email}"^^xsd:string ;
                        :Phone "{user.phone}"^^xsd:string ;
                        :Username "{user.username}"^^xsd:string ;
                        :hasRole :Role_{user.id_role_id} .
                }}
                WHERE {{
                    FILTER NOT EXISTS {{ :User_{user.id} a :User . }}
                }}
            """
            execute_query(query)

        # 3. Перенос даних про партнерів
        # реалізація функціоналу переносу даних партнерів у онтологію
```

```
        # разом із зв’язками з кухарями та кур’єрами

        # 4. Перенос даних про меню
        # реалізація функціоналу переносу даних меню у онтологію

        # 5. Перенос даних про елементи меню (MenuItems)
        for menu_item in MenuItems.objects.all():
            query = f"""
                PREFIX :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
                PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
                INSERT {{
                    :MenuItem_{menu_item.id} a :MenuItem ;
                        :dishe "{menu_item.dishes}"^^xsd:string ;
                        :price "{menu_item.price}"^^xsd:decimal ;
                        :ingredients "{menu_item.ingredients}"^^xsd:string ;
                        :hasMenuItems :Menu_{menu_item.id_menu_id} .
                }}
                WHERE {{
                    FILTER NOT EXISTS {{ :MenuItem_{menu_item.id} a :MenuItem
. }}
                }}
            """
            execute_query(query)

        # 6. Перенос даних про OrderMenuItems
        # реалізація функціоналу переносу даних OrderMenuItems у онтологію

        # 7. Перенос даних про замовлення (Order)
        for order in Order.objects.prefetch_related('menu_items').all():
            query = f"""
                PREFIX:
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
                PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
                INSERT {{
                    :Order_{order.id} a :Order ;
                        :orderStatus "{order.status}"^^xsd:string ;
                        :address "{order.address or ''}"^^xsd:string ;
                        :hasOrderClient :User_{order.client_id} ;
                        :hasOrderCourier :User_{order.courier_id} ;
                        :hasOrderCook :User_{order.cook_id} ;
                        :hasOrderEstablishment
:Establishment_{order.partner_id} .
                }}
                WHERE {{
                    FILTER NOT EXISTS {{ :Order_{order.id} a :Order . }}
                }}
            """
            execute_query(query)
# Додавання зв'язків із OrderMenuItems
            # реалізація функціоналу додавання зв’язків замовлення з
позиціями замовлення
        # Перенос даних про рейтинги
```

```
        # реалізація функціоналу переносу даних Rating у онтологію

        # Перенос даних про взаємодії користувачів
        # реалізація функціоналу переносу даних UserInteraction у онтологію

        # Перенос даних про інтереси користувачів до меню
        # реалізація функціоналу переносу даних UserMenuInterest у онтологію
        self.stdout.write(self.style.SUCCESS('All data has been successfully
loaded into the ontology.'))
```

Лістинг А.2. Код файлу signals.py

```
# Додавання зв'язків із OrderMenuItems
            # реалізація функціоналу додавання зв’язків замовлення з
позиціями замовлення
        # Перенос даних про рейтинги
        # реалізація функціоналу переносу даних Rating у онтологію

        # Перенос даних про взаємодії користувачів
        #  реалізація функціоналу переносу даних UserInteraction у
онтологію

        # Перенос даних про інтереси користувачів до меню
        #  реалізація функціоналу переносу даних UserMenuInterest у
онтологію
        self.stdout.write(self.style.SUCCESS('All    data    has    been
successfully loaded into the ontology.'))

from django.core.management.base import BaseCommand
from django.db.models.signals import post_save, pre_delete
from django.dispatch import receiver
from test_app.models import (
    Role, User, Partner, Menu, MenuItems,
    Order, OrderMenuItems,
    UserMenuInterest, UserInteraction, Rating,
)
from SPARQLWrapper import SPARQLWrapper, POST

# Налаштування SPARQL-запитів
sparql                                                              =
SPARQLWrapper("http://localhost:7200/repositories/3_Repo/statements")
sparql.setMethod(POST)

def execute_query(query):
    """Виконує SPARQL-запит"""
    sparql.setQuery(query)
    sparql.query()


# --- Ролі ---
#   реалізація   функціоналу   синхронізації   ролей   з   онтологією
(CREATE/UPDATE/DELETE)
```

```
# --- Користувачі (User) ---
@receiver(post_save, sender=User)
def sync_user_to_ontology(sender, instance, **kwargs):
    query = f"""
        PREFIX :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
        PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
        DELETE WHERE {{ :User_{instance.id} ?p ?o . }};
        INSERT DATA {{
            :User_{instance.id} a :User ;
                :Name "{instance.name}"^^xsd:string ;
                :Surname "{instance.surname}"^^xsd:string ;
                :Patronymic "{instance.patronymic or ''}"^^xsd:string ;
                :Email "{instance.email}"^^xsd:string ;
                :Phone "{instance.phone}"^^xsd:string ;
                :Username "{instance.username}"^^xsd:string ;
                :hasRole :Role_{instance.id_role_id} .
        }}
    """
    execute_query(query)


@receiver(pre_delete, sender=User)
def delete_user_from_ontology(sender, instance, **kwargs):
    query = f"""
        PREFIX :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
        DELETE WHERE {{ :User_{instance.id} ?p ?o . }}
    """
    execute_query(query)

# --- Рейтинги ---
# реалізація функціоналу синхронізації рейтингів з онтологією

# --- Взаємодії користувачів ---
#  реалізація функціоналу синхронізації взаємодій користувачів з
онтологією

# --- Інтереси користувачів до меню ---
#  реалізація функціоналу синхронізації інтересів користувачів з
онтологією

# --- Партнери ---
#  реалізація функціоналу синхронізації партнерів (Establishment) з
онтологією
# включно з додаванням зв’язків hasCook та hasCourier

# --- Меню ---
# реалізація функціоналу синхронізації меню з онтологією

# --- Замовлення (Order) ---
@receiver(post_save, sender=Order)
```

```
def sync_order_to_ontology(sender, instance, **kwargs):
    query = f"""
        PREFIX                                                                  :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
        PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
        DELETE WHERE {{ :Order_{instance.id} ?p ?o . }};
        INSERT DATA {{
            :Order_{instance.id} a :Order ;
                :orderStatus "{instance.status}"^^xsd:string ;
                :address "{instance.address or ''}"^^xsd:string ;
                :hasOrderClient :User_{instance.client_id} ;
                :hasOrderCourier :User_{instance.courier_id} ;
                :hasOrderCook :User_{instance.cook_id} ;
                :hasOrderEstablishment
:Establishment_{instance.partner_id} .
        }}
    """
    execute_query(query)

    # Додавання зв'язків із OrderMenuItems
    # реалізація функціоналу додавання зв’язків замовлення з елементами
замовлення (OrderMenuItems)

Саша Солнцев, [22.12.2025 9:32]
@receiver(pre_delete, sender=Order)
def delete_order_from_ontology(sender, instance, **kwargs):
    query = f"""
        PREFIX                                                                  :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
        DELETE WHERE {{ :Order_{instance.id} ?p ?o . }}
    """
    execute_query(query)

# --- Елементи замовлення (OrderMenuItems) ---
# реалізація функціоналу синхронізації OrderMenuItems з онтологією

# --- Елементи меню (MenuItems) ---
@receiver(post_save, sender=MenuItems)
def sync_menu_item_to_ontology(sender, instance, **kwargs):
    query = f"""
        PREFIX                                                                  :
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
        PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
        DELETE WHERE {{ :MenuItem_{instance.id} ?p ?o . }};
        INSERT DATA {{
            :MenuItem_{instance.id} a :MenuItem ;
                :dishe "{instance.dishes}"^^xsd:string ;
                :price "{instance.price}"^^xsd:decimal ;
                :ingredients "{instance.ingredients}"^^xsd:string ;
                :hasMenuItems :Menu_{instance.id_menu_id} .
        }}
    """
    execute_query(query)
```

```
@receiver(pre_delete, sender=MenuItems)
def delete_menu_item_from_ontology(sender, instance, **kwargs):
    query = f"""
        PREFIX:
<http://www.semanticweb.org/solnc/ontologies/2024/10/Ontology#>
        DELETE WHERE {{ :MenuItem_{instance.id} ?p ?o . }}
    """
    execute_query(query)

class Command(BaseCommand):
    help = 'Load all existing data from the database into the ontology'

    def handle(self, *args, **kwargs):
      self.stdout.write(self.style.SUCCESS('Real-time  synchronization  with
ontology enabled.'))
```

Лістинг А.3. Тест онтології №1

```
def test_1_initial_recommendations_and_ontology_links(self):
        """
        Тест 1.
        А: базові рекомендації за початковою історією (Django-БД).
        В: розширені рекомендації на основі онтології (зв'язки isSimilarTo).
        """
        print("\n=== ТЕСТ 1: Початкові рекомендації та онтологічні зв'язки
===")
        # 1. Початкові замовлення
        base_items = [
            (1, 3),   # Шаурма ×3
            (2, 2),   # Цезар ×2
            (3, 1),   # Фірмова ×1
        ]
        self.create_order_with_items(self.test_user, base_items)

        # 2. Взаємодії та рейтинги
        interactions_spec = [
            {"menu_item_id": 1, "actions": ["view", "purchase", "rate"],
"rating": 5},
            {"menu_item_id": 2, "actions": ["view", "purchase"], "rating":
4},
            {"menu_item_id": 3, "actions": ["view"], "rating": 4},
        ]
        self.create_interactions_and_ratings(self.test_user,
interactions_spec)
        # 3. Перерахунок інтересу до меню
        self.recalculate_user_interest(self.test_user)
        # 4. Базові рекомендації (А)
        recommended_items = get_recommendations_for_user(self.test_user)
        self.assertTrue(
            recommended_items.exists(),
            "Список базових рекомендацій порожній",
```

```
        )

        # беремо тільки перші 5
        top5_recommended = list(recommended_items[:5])
        print("\nA) Базові рекомендації Django (топ-5, тільки назви):")
        for item in top5_recommended:
            print(f"  A: {item.dishes}")
        recommended_ids = set(recommended_items.values_list("id", flat=True))
        self.assertTrue(
            recommended_ids.intersection({1, 2, 3}),
            f"Очікувались рекомендації серед {{1,2,3}}, отримано:
{recommended_ids}",
        )

        # 5. Розширені рекомендації з урахуванням онтології (B)
        enhanced_recommendations =
get_enhanced_recommendations(self.test_user)
        self.assertTrue(
            enhanced_recommendations,
            "Список розширених рекомендацій (з онтологією) порожній",
        )
        # беремо тільки перші 5 пар
        top5_enhanced = enhanced_recommendations[:5]

        print("\nB) Розширені рекомендації (топ-5, A(рекомендації на основі
дій користувача) → B(рекомендації на основі онтологічних зв'язків)):")
        for rec in top5_enhanced:
            main_item = rec["main_item"]
            similar_item = rec["similar_item"]
            print(
                f"  A: {main_item['name']} → B: {similar_item['name']}"
            )

        # 6. Перевірка мепінгу ID з онтології до Django-БД
        for rec in enhanced_recommendations:
            main_item = rec["main_item"]
            similar_item = rec["similar_item"]

            self.assertIn(
                main_item["id"],
                recommended_ids,
                "main_item у розширених рекомендаціях не входить до базових
рекомендацій",
            )

            self.assertTrue(
                MenuItems.objects.filter(id=similar_item["id"]).exists(),
                f"similar_item з id={similar_item['id']} відсутній у БД
MenuItems",
            )
        print("\n[ПІДСУМОК ТЕСТУ 1] Топ-5 базових рекомендацій (A) та топ-5
онтологічно "
```

```
            "розширених рекомендацій (B) сформовані коректно: B отримано
шляхом "
            "семантичного розширення A за зв'язками isSimilarTo.")
```

Лістинг А.4. Тест онтології №2

```
def test_2_recommendations_change_after_behavior_update(self):
        """
        Тест 2.
        Перевірка, що після зміни історії взаємодій:
        - змінюються топ 5 базових рекомендацій (A1 → A2),
        - відповідно змінюються топ 5 онтологічних рекомендацій (B2).
        """

        print("\n=== ТЕСТ 2: Зміна поведінки користувача та рекомендацій
===")

        # 1. Початкова поведінка: Шаурма, Цезар
        self.create_order_with_items(self.test_user, [(1, 2), (2, 2)])
        self.create_interactions_and_ratings(
            self.test_user,
            [
                {"menu_item_id": 1, "actions": ["view", "purchase"],
"rating": 5},
                {"menu_item_id": 2, "actions": ["view"], "rating": 4},
            ],
        )
        self.recalculate_user_interest(self.test_user)

        initial_recommended = get_recommendations_for_user(self.test_user)
        initial_ids = list(initial_recommended.values_list("id", flat=True))
        top5_initial = list(initial_recommended[:5])

        print("\nA1) Початкові базові рекомендації Django (топ 5):")
        for item in top5_initial:
            print(f"  A1: {item.dishes}")

        # 2. Скидаємо історію та моделюємо нову поведінку
        Order.objects.all().delete()
        OrderMenuItems.objects.all().delete()
        UserInteraction.objects.all().delete()
        Rating.objects.all().delete()
        UserMenuInterest.objects.all().delete()

        new_items = [
            (4, 2),   # По чом в Одесі
            (5, 2),   # Струнка Мар'яна
            (9, 1),   # Солодкі вафлі
            (10, 1),  # Солоні вафлі
        ]
        self.create_order_with_items(self.test_user, new_items)
        self.create_interactions_and_ratings(
            self.test_user,
```

```
            [
                {"menu_item_id": 4, "actions": ["view", "purchase"],
"rating": 5},
                {"menu_item_id": 5, "actions": ["view", "purchase"],
"rating": 5},
                {"menu_item_id": 9, "actions": ["view"], "rating": 4},
                {"menu_item_id": 10, "actions": ["view"], "rating": 4},
            ],
        )
        self.recalculate_user_interest(self.test_user)

        updated_recommended = get_recommendations_for_user(self.test_user)
        updated_ids = list(updated_recommended.values_list("id", flat=True))
        top5_updated = list(updated_recommended[:5])

        print("\nA2) Оновлені базові рекомендації Django після зміни
поведінки "
            "(топ 5):")
        for item in top5_updated:
            print(f"  A2: {item.dishes}")
        # 3. Перевірка зміни рекомендацій
        self.assertNotEqual(
            set(initial_ids),
            set(updated_ids),
            "Рекомендації не змінилися після суттєвої зміни історії
взаємодій",
        )
        self.assertTrue(
            set(updated_ids).intersection({4, 5, 9, 10}),
            f"Очікувалось зміщення рекомендацій до {{4,5,9,10}}, отримано:
{updated_ids}",
        )
        # 4. Розширені рекомендації (B2) після зміни поведінки
        updated_enhanced = get_enhanced_recommendations(self.test_user)
        self.assertTrue(
            updated_enhanced,
            "Після зміни поведінки список розширених рекомендацій порожній",
        )
        top5_updated_enhanced = updated_enhanced[:5]
        print("\nB2) Розширені рекомендації після зміни поведінки "
            "(топ 5, A2 → B2):")
        for rec in top5_updated_enhanced:
            main_item = rec["main_item"]
            similar_item = rec["similar_item"]
            print(
                f"  A2: {main_item['name']} → B2: {similar_item['name']}"
            )

Саша Солнцев, [22.12.2025 10:01]
for rec in updated_enhanced:
            self.assertTrue(

MenuItems.objects.filter(id=rec["similar_item"]["id"]).exists(),
```

```
                "Схожа страва від онтології не має відповідника в таблиці
MenuItems",
            )
        print("\n[ПІДСУМОК ТЕСТУ 2] Після зміни історії взаємодій змінився
склад топ 5 "
            "базових рекомендацій (A1 → A2), а також відповідний топ 5 "
            "онтологічно розширених рекомендацій (B2), що підтверджує
узгоджену "
            "роботу поведінкової та онтологічної частин системи.")
```

Другий тест ілюструє адаптивність системи.