

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

(повне найменування інституту, факультету)

Автоматизованих систем обробки інформації і управління

(повна назва кафедри)

«До захисту допущено»

В.о. завідувача кафедри

_____ *Олександр ПАВЛОВ*
(підпис) (ініціали, прізвище)

“ ” 2021 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Програмне забезпечення
комп'ютеризованих систем»

спеціальності «121 Інженерія програмного забезпечення»

на тему Бібліотека для логування змін в моделях Django-застосувань

Виконав: студент IV курсу, групи

ІП-72 Вакуленко Кирило

Віталійович

(прізвище, ім'я, по батькові)

(підпис)

Керівник

доц., к.т.н., Ліщук К.І.

посада, науковий ступінь, вчене звання, прізвище, і ім'я, по батькові

(підпис)

**Консультант
з графічної
документації**

доц., к.т.н., Ліщук К.І.

посада, науковий ступінь, вчене звання, прізвище, і ім'я, по батькові

(підпис)

Рецензент:

посада, науковий ступінь, вчене звання, прізвище, ім'я, по батькові

(підпис)

Засвідчую, що у цьому дипломному проєкті
немає запозичень з праць інших авторів без
відповідних посилань.

Студент _____
(підпис)

Київ – 2021 року

**Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”**

Факультет (інститут) Інформатики та обчислювальної техніки
(повна назва)

Кафедра автоматизованих систем обробки інформації і управління
(повна назва)

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 Інженерія програмного забезпечення

Освітньо-професійна програма – Програмне забезпечення
комп'ютеризованих систем

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

Олександр ПАВЛОВ
(підпис)

“ ” _____ 2021 р.

**ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ СТУДЕНТУ**

Вакуленку Кирилу Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема проєкту «Бібліотека для логування змін в моделях
Django-застосувань»

керівник проєкту доц., к.т.н., Ліщук К.І.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “11” травня 2021 р. №1139-с

2. Термін подання студентом проєкту «08» червня 2021 року

3. Вихідні дані до проєкту

Технічне завдання

4. Зміст пояснювальної записки

1) Аналіз вимог до програмного забезпечення: основні визначення та терміни, опис предметного середовища, огляд існуючих технічних рішень та відомих програмних продуктів, розробка функціональних та нефункціональних вимог

2) Моделювання та конструювання програмного забезпечення: моделювання та аналіз програмного забезпечення, засоби розробки, технічні рішення, архітектура програмного забезпечення

3) Розгортання та впровадження програмного забезпечення

4) Керівництво користувача, методика випробувань програмного продукту

5. Перелік графічного матеріалу

1) *Схема структурна варіантів використання*

2) *Схема структурна класів програмного забезпечення*

3) *Креслення вигляду екранних форм*

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання «14» березня 2021 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проєкту	Термін виконання етапів проєкту	Примітка
1.	<i>Вивчення рекомендованої літератури</i>	<i>21.03.2021</i>	
2.	<i>Аналіз існуючих методів розв'язання задачі</i>	<i>27.03.2021</i>	
3.	<i>Постановка та формалізація задачі</i>	<i>05.04.2021</i>	
4.	<i>Аналіз вимог до програмного забезпечення</i>	<i>12.04.2021</i>	
5.	<i>Алгоритмізація задачі</i>	<i>17.04.2021</i>	
6.	<i>Моделювання програмного забезпечення</i>	<i>21.04.2021</i>	
7.	<i>Обґрунтування використовуваних технічних засобів</i>	<i>23.04.2021</i>	
8.	<i>Розробка архітектури програмного забезпечення</i>	<i>29.04.2021</i>	
9.	<i>Розробка програмного забезпечення</i>	<i>05.05.2021</i>	
10.	<i>Налагодження програми</i>	<i>07.05.2021</i>	
11.	<i>Виконання графічних документів</i>	<i>12.05.2021</i>	
12.	<i>Оформлення пояснювальної записки</i>	<i>15.05.2021</i>	
13.	<i>Подання ДП на попередній захист</i>	<i>17.05.2021</i>	
14.	<i>Подання ДП рецензенту</i>	<i>01.06.2021</i>	
15.	<i>Подання ДП на основний захист</i>	<i>08.06.2021</i>	

Студент

_____ Кирило ВАКУЛЕНКО
(підпис)

Керівник

_____ Катерина ЛІЩУК
(підпис)

[illegible]

АНОТАЦІЯ

Структура та обсяг роботи. Пояснювальна записка дипломного проєкту складається з 4 розділів, містить 168 сторінки 90 таблиць 29 рисунків, 10 джерел.

Мета. Метою дипломного проєкту є створення бібліотеки, що надає можливість відстежувати зміни, що відбуваються внаслідок взаємодії користувачів з системою та адаптувати пропонуване рішення до потреб та особливостей мікросервісної архітектури.

У першому розділі описано предметну область, відомі технічні рішення, відмінність від наявних аналогів, функціональні та не функціональні вимоги.

У другому розділі описано архітектуру системи відстеження змін та прототипу системи, що її використовує, структуру класів плагіну виробника та споживача повідомлень.

У третьому розділі описано перевірку функціональності системи відповідно до варіантів використання.

У четвертому розділі описано процес розгортання програмного забезпечення та роботи з ним.

БАЗИ ДАНИХ, БІБЛІОТЕКА, БРОКЕРИ ПОВІДОМЛЕНЬ,
ВЗАЄМОДІЯ КОРИСТУВАЧІВ З СИСТЕМОЮ

					КПІ.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		5

ABSTRACT

Structure and scope of work. The explanatory note of the diploma project consists of 4 sections, contains 168 pages 90 tables 29 figures, 10 sources.

Goal. The aim of the diploma project is to create a library that provides the ability to track changes that occur due to user interaction with the system and adapt this solution to the needs and characteristics of microservice architecture.

The first section describes the subject area, known technical solutions, differences from existing analogues, functional and non-functional requirements.

The second section describes the architecture of the change tracking system and the prototype of the system that uses it, the structure of the plug-in classes of the producer and consumer of messages.

The third section describes how to check the functionality of the system according to the use cases.

The fourth section describes the process of deploying and working with the software.

DATABASES, LIBRARY, MESSAGE BROKERS, USER INTERACTION
WITH THE SYSTEM

					КПІ.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		6

Пояснювальна записка до дипломного проєкту

на тему: Бібліотека для логування змін в моделях Django-застосувань

Київ – 2021 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	10
ВСТУП	11
1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	12
1.1 ЗАГАЛЬНІ ПОЛОЖЕННЯ	12
1.2 ЗМІСТОВНИЙ ОПИС І АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	13
1.3 АНАЛІЗ УСПІШНИХ ІТ-ПРОЕКТІВ	19
1.3.1 Аналіз відомих технічних рішень	19
1.4 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	21
1.4.1 Розроблення функціональних вимог	21
1.4.2 Розроблення нефункціональних вимог	48
1.4.3 Постановка комплексу завдань модулю	48
1.5 ВИСНОВКИ ПО РОЗДІЛУ	49
2 МОДЕЛЮВАННЯ ТА КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	50
2.1 АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	50
2.1.1 Система відстеження змін	50
2.1.2 Прототип вебсайту	51
2.2 КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	54
2.2.1 Django-плагін виробника повідомлень	54
2.2.2 Django-плагін споживача повідомлень	75
2.3 ВИСНОВКИ ПО РОЗДІЛУ	84
3 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	85
3.1 АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	85
3.2 ВИСНОВКИ ДО РОЗДІЛУ	97

4	ВПРОВАДЖЕННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	98
4.1	РОЗГОРТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	98
4.2	РОБОТА З ПРОГРАМНИМ ЗАБЕЗПЕЧЕННЯМ.....	98
4.3	ВИСНОВКИ ДО РОЗДІЛУ	99
	ВИСНОВКИ	100
	ПЕРЕЛІК ПОСИЛАНЬ	101
	ДОДАТОК А ЗВІТ З ПЕРЕВІРКИ НА ПЛАГІАТ	103

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Бібліотека – це сукупність функцій, класів, модулів, що надають функціонал, який розробники використовують у своїх проєктах задля уникнення розробки вже наявних рішень для проблем, що постають при розробці.

Фреймворк – це платформа, що є фундаментом для створення програмного забезпечення. Фреймворк, як і бібліотека, містить в собі набір класів, модулів і так далі. Відмінність між ними полягає в тому, що бібліотеки вбудовуються у програмні рішення, щоб розв’язати певну локальну задачу, а фреймворк вже є готовим програмним рішенням в яке розробник вбудовує власний код, та яке розв’язує такі глобальні питання як архітектура застосунку та внутрішня структура коду [1].

Об’єктно-реляційне відображення – це підхід до розв’язання проблеми представлення структури та даних реляційної бази даних в рамках об’єктноорієнтованої мови програмування. Це рішення дозволяє відображати структуру, дані таблиць та взаємозв’язки між таблицями за допомогою класів та їх об’єктів всередині мови програмування.

Реляційні бази даних – це набір даних, які зберігаються відповідно до реляційної моделі даних, тобто структура бази даних складається з таблиць, які складаються з колонок та зберігають певну інформацію.

СУБД – система управління базами даних [2].

Брокер повідомлень – це проміжне програмне забезпечення, що виступає посередником у спілкуванні між декількома системами.

ВСТУП

На сьогодні в Інтернеті розміщена велика кількість вебсайтів, якими користуються багато людей. Сайти, зазвичай, є об'єднанням двох застосунків – клієнта, з яким взаємодіють люди, та сервера, з яким взаємодіє клієнт, кожен з яких розробляється за допомогою певної мови програмування та певного фреймворку. Одним з популярних фреймворків для створення серверної частини є Django для мови програмування Python.

Дані, з якими взаємодіють люди, повинні певним чином зберігатися для подальшого використання. Існують різні підходи до збереження інформації, одним з яких є використання реляційних баз даних. В рамках Django серверів одним з найпоширеніших способів є саме цей підхід.

Користувачі взаємодіють з застосунками різними способами. Вони можуть переглядати вміст цих сторінок, вводити певні дані, натискати на кнопки й тому подібне. В будь-якому випадку люди або бачать інформацію, яку розмістили на вебсайті інші користувачі, або ж самі додають свою до вже наявної. В результаті певних взаємодій особи з клієнтом дані потрапляють до сервера, який, після обробки, зберігає їх у базі даних, або видаляє звідти.

Сучасні вебдодатки є дуже складними системами, як з боку розробки, так і з боку використання. Ці системи є дуже чутливими до зміни, додавання, видалення даних, адже люди взаємодіють з цінною інформацією такою як бюджети, для фінансових сфер, історією хворих, для сфери медицини, й так далі. Саме тому дуже важливо відстежувати та зберігати інформацію про те, хто як і коли взаємодіяв з даними.

1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Загальні положення

Відстеження змін в базі даних, які виникають в наслідку взаємодії користувачів з системою, є нетривіальною задачею, яка потребує втручання у різні рівні застосунку, наприклад рівень HTTP-запиту, для визначення користувача, який вносить зміни, або рівень взаємодії з базою даних, для вилучення інформації про зміни в таблицях.

Ця задача стає більш комплексною, коли її потрібно вирішити в рамках мікросервісної системи, яка складається з багатьох підсистем, кожна з яких може бути створена за допомогою різних технологій та різних підходів. В рамках однієї системи один з мікросервісів може розроблятися за допомогою мови JavaScript, з використанням бібліотеки Express, а інший за допомогою мови Python та платформи Django.

Не всі дії користувача можливо відстежити. Якщо користувач викликає збережену процедуру, то ми не зможемо відстежити хто саме змінив дані в базі даних, адже збережені процедури не мають доступу до рівня HTTP-запиту.

Вище згадані особливості потрібно враховувати при розробці системи відстеження змін в базі даних, а отже необхідно накласти ряд обмежень на саму систему, що створюється, та на програмне забезпечення, що буде її використовувати.

В рамках даної роботи розробляється програмне забезпечення, яке може бути використано в мікросервісній системі, для сервісів, що розробляються на платформі Django, та які використовують вбудовану в Django об'єктно-реляційну проєкцію, як систему управління баз даних застосовують PostgreSQL.

1.2 Змістовний опис і аналіз предметної області

В сучасному світі існує безліч мов програмування, кожна з яких застосовується для розв'язання певних задач. Одні застосовуються для аналізу даних, інші для системного програмування, розробки вебсайтів і так далі. В будь-якому випадку, незалежно від сфери застосування, для створення програмного забезпечення наряду з базовим функціоналом мови програмування застосовуються бібліотеки та фреймворки створені для використання в тій чи іншій мові.

Django – це фреймворк для мови програмування Python, що використовується для створення вебсайтів. За допомогою Django можливо створювати як цілі застосунки, так і застосовувати його для розробки серверної частини додатку в той час, як клієнт буде розроблений іншою мовою програмування. Платформа надає наступний функціонал:

- взаємодія з базою даних, за допомогою вбудованої ORM та модулів для роботи з базою;
- побудова HTML документів за допомогою системи шаблонів;
- аутентифікація користувача;
- платформа для кешування, яка підтримує різні методи кешування;
- створення системи для адміністратора, з можливістю розширення функціоналу;
- набір функцій та класів для створення контролерів;
- набір функцій та класів для побудови маршрутизації запитів;
- набір функцій та класів для серіалізації даних.

Django має характерну структуру, яка є усталеною на рівні фреймворку, а отже різні проєкти, що створюються за допомогою цієї платформи, мають дуже схожу структуру з незначними відмінностями. В структурі передбачено наявність кореневого модуля, що містить основні

налаштування застосунку, а також декілька модулів, що в термінології Django називаються app.

Кожен app є окремим застосунком, який відповідає за певний функціонал в рамках усього проєкту. Сукупність усіх app та кореневого модуля є усім вебзастосунком.

Кожен конкретний app має загальну для всіх app структуру. Він містить файли, або модулі з характерними назвами models, views, templates [3]. Ці назви відповідають складовим частинам архітектурного шаблону MVC (модель, представлення, контролер).

В модулі views реалізовується архітектурний компонент контролер, ця частина відповідає за взаємодію з двома іншими компонентами та призначена для передачі даних між цими рівнями.

Модуль templates відповідає за рівень представлення, слугує для відображення даних, та містить HTML-шаблони Django, що дозволяють динамічно формувати HTML-документи.

Модуль models відповідає рівню моделі та слугує для взаємодії з базою даних та бізнес-логікою.

Створення таблиць в базі даних, з використанням Django, відбувається в декілька етапів. Спочатку створюються класи, що називають моделями, які розширюють функціонал базового класу django.db.models.Model. Ці класи пов'язані з певним app, всередині якого вони створюються, та розміщуються в модулі models.

На наступному рисунку зображено приклад створення моделей в Django.

```

from django.db import models

class User(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)

class Car(models.Model):
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
    brand = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=10, decimal_places=2)

```

Рисунок 1.1 – Приклад моделей в Django

На Рисунку 1.1 можна побачити як створюються моделі на прикладі моделей User та Car, та як створюються поля та задаються їх типи. Після наступних кроків ці моделі будуть відображені у відповідні таблиці, з відповідними назвами й полями.

Наступний крок – виклик вбудованої команди для створення міграцій. Після відпрацювання команди буде отримано файли з кодом, який відповідає за внесення змін у дані та структури бази даних. Виклик отриманого коду за допомогою спеціальної команди призведе до проведення міграції в базі даних, в результаті якої буде змінено структуру та/або дані. Моделі в Django є реалізацією об'єктно-реляційного відображення.

Django надає можливість створювати та підключати до своїх проєктів різні плагіни. Одна з причин поширеності цього фреймворку є велика кількість доступних плагінів з відкритим кодом. Деякі з них можуть бути або просто підключені до застосунку, і одразу виконувати поставлені задачі, а деякі використовуються як бібліотеки. Плагіни можуть розширювати будь-який з рівнів застосунку. Simple JWT, наприклад, втручається в рівень HTTP-запиту та змінює вбудований спосіб аутентифікації на аутентифікацію за допомогою JSON Web Token, django-dirtyfields розширює функціонал на рівні моделей та надає додаткові методи для взаємодії з ними [4], а в рамках

даної дипломної роботи розробляється плагін, який розширює функціонал на рівні моделей.

Django містить функціонал так званих сигналів. Сигнали в Django – це реалізація паттерну “Спостерігач”. Сигнали є видавцями, на яких розробник може підписувати власні функції, в які передаються параметри, що визначені сигналом, на який вони підписуються [5]. Django містить ряд вбудованих сигналів, а також надає можливість створювати власні.

На сьогодні велика кількість вебзастосунків розробляється з використанням мікросервісної архітектури. Ідея полягає у розбитті системи на підсистеми, кожна з яких є окремим сервісом. Такий підхід має наступні переваги:

- кожен сервіс має власну кодову базу, отже знаходиться в окремому репозиторії, що зменшує конфлікти у коді;
- кожен сервіс є окремим проєктом, що дає можливість зручно розподіляти відповідальність команд розробників між проєктами;
- існує можливість відносно легко переробити одну з частин системи використовуючи іншу технологію, якщо в процесі розробки виявиться, що наявне рішення не відповідає вимогам.

Існують різні підходи для розв’язання задачі комунікації мікросервісів один з одним, наприклад взаємодія за допомогою протоколу HTTP, але такий спосіб не є оптимальним, оскільки виникають проблеми з продуктивністю, адже HTTP-протокол є достатньо повільним через його надійність, бо витрачається час на встановлення з’єднання між сервісами, підтвердження результатів будь-якої взаємодії між ними й так далі. Якщо ж мікросервісів в системі дуже багато і вони постійно взаємодіють один з одним неоліки швидкості можуть стати критичними для функціонування систем, тому в деяких випадках краще застосовувати інші підходи, такі як комунікація за допомогою брокерів повідомлень, які не мають проблем зі швидкістю передачі даних.

Одним з таких брокерів є Kafka. Для опису принципу роботи Kafka, потрібно виділити наступні терміни в рамках термінології цього брокеру повідомлень:

- виробник (producer) – це застосунок, що відправляє повідомлення у Kafka, таких виробників може бути декілька;
- тема (topic) – це ідентифікатор джерела даних, кожен виробник відправляє повідомлення в певну тему;
- споживач (consumer) – це застосунок, який читає повідомлення з певної теми.

Кожен виробник повідомлень відправляє повідомлення у Kafka, в певну тему, кожен споживач, який підписаний на цю тему, читає повідомлення з потоку даних для подальшої обробки. Кожен споживач має прив'язку до певної групи. Споживачі, що читають повідомлення з однієї теми та знаходяться у різних групах отримують повідомлення незалежно один від одного. В ситуації, коли споживачі знаходяться в одній групі та читають повідомлення з однієї теми, повідомлення будуть розподілятися між споживачами за певною стратегією, наприклад по колу, тобто дані будуть рівномірно розподілятися між споживачами через те, що перше повідомлення отримає перший споживач друге другий, і так далі.

Розбиття споживачів на групи потрібне для того, щоб надати можливість збільшити ефективність вилучення даних з теми, оскільки може скластися ситуація, що повідомлення надходять в тему занадто швидко, і споживач не встигає їх обробляти, через що утворюється накопичення необроблених даних. Якщо сталася така ситуація, то можна додати до групи нових споживачів, що дозволить прискорити процес обробки даних.

Для того, щоб повідомлення могли бути розбиті на декілька частин і відповідно прочитані декількома споживачами, потрібно розбити тему на розділи, наприклад, для двох розділів потрібно запустити двох споживачів, які читають з однієї теми та знаходяться в одній групі [6]. Якщо ж

					КПІ.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		17

споживачів буде більше ніж розділів, то один зі споживачів не буде отримувати дані. Якщо розділів у темі більше, ніж споживачів, то деякі споживачі будуть читати з більшої кількості розділів відносно інших [7].

Для кожного споживача існує окрема черга повідомлень, до якої прив'язане значення зміщення, яке відповідає за індекс останнього прочитаного повідомлення, а отже якщо виробники повідомлень відправляють дані у тему, а один з споживачів в цей час вимкнений, то як тільки він підключиться до системи він зможе розпочати читати повідомлення починаючи з останнього прочитаного повідомлення.

В рамках даної роботи розробляється програмна система, що складається з прототипу вебсайту, який розробляється за допомогою мікросервісної архітектури, кожен мікросервіс якої розробляється за допомогою Django, та системи для відстеження змін у реляційній базі даних PostgreSQL, яку потрібно підключити до наявного вебдодатку.

Система для відстеження змін складається з Django-плагіну, який є бібліотекою, та надає мікросервісам можливість відправляти повідомлення, брокеру повідомлень Apache Kafka та сервісу, що виступає у ролі споживача повідомлень. При розробці системи мають бути враховані особливості та можливості Django для того, щоб система не змінювала загальну структуру та не перешкоджала наявному функціоналу.

Прототип вебсайту розробляється як система для розміщення повідомлень про продаж певних товарів. Система містить в собі базовий функціонал для демонстрації можливостей системи для відстеження змін та тестування коректності її роботи, а отже не потребує детального огляду та аналізу.

1.3 Аналіз успішних ІТ-проектів

1.3.1 Аналіз відомих технічних рішень

Застосунки, що розробляються з використанням PostgreSQL як СУБД, мають можливість розв'язати проблему використовуючи вбудовані можливості даної СУБД. Для цього потрібно створити додаткову таблицю, що буде зберігати результати змін та створити тригер, що буде спрацьовувати перед додаванням, видаленням та редагуванням рядків в таблиці [8]. Дане рішення надає можливість відстежувати наступну інформацію:

- час та дату, коли відбулася зміна;
- назву таблиці, де відбулася зміна;
- тип зміни;
- ім'я користувача, що вніс зміну;
- старі значення колонок запису;
- нові значення колонок запису.

Перевагами цього підходу є те, що він універсальний для будь-якої системи, що використовує СУБД PostgreSQL, та може бути застосований незалежно від мови програмування та платформи. Також цей підхід надає можливість відстежувати будь-які зміни в таблицях, адже тригери працюють на рівні бази даних, а не коду, а отже не може бути такого випадку, що застосунок, або розробник змінює інформацію в базі даних в обхід тригерів.

Недоліками даного підходу є те, що інформація про користувача, що вніс зміни обмежується рівнем користувача бази даних. Зазвичай така інформація є непотрібною для вебдодатків, адже користувачі в системі визначаються на рівні back end частини, а на рівні бази даних існує один, або декілька користувачів. Також цей спосіб не підходить для мікросервісної системи, адже зміни зберігаються на рівні бази даних. Кожен мікросервіс, зазвичай, має власну базу даних, отже інформація про зміни в системі буде

розподілена між мікросервісами, і це позбавить систему можливості централізовано оброблювати дані про зміни.

Django-застосунки можуть використати можливості вбудованої ORM для відстеження змін базі даних. Для цього необхідно розширити функціонал класу `django.db.models.Model`. Цей клас є базовим для усіх моделей, що створюють розробники й містить методи, що відповідають за створення, видалення, редагування записів в базі даних. В результаті розширення функціоналу утворюється новий базовий клас. Моделі, що будуть розширювати функціонал отриманого класу мають можливість зберігати таку ж інформацію, що і в попередньому підході. Також відстеження змін можна реалізувати використовуючи механізм сигналів Django. Для цього потрібно підписатися на сигнали, що спрацьовують при збереженні та видаленні об'єкту моделі. Результати змін можуть бути збережені як в тій самій таблиці, де відбуваються зміни, так і в окремій таблиці відведеній під зміни [9].

Перевагою даного підходу є те, що він надає можливість відстежувати користувачів, що вносять зміни у систему, адже Django безпосередньо працює з рівнем HTTP-запиту, в якому міститься необхідна інформація про користувача.

Недоліком цього рішення є те, що він не дає можливості відстежувати усі зміни, що відбуваються в базі даних, адже існує багато функціоналу як в Django, так і поза його межами, що надає можливість вносити зміни в базу даних, оминаючи функціонал відстеження даних. Прикладами можуть слугувати збережні процедури та bulk-операції з базою даних. Також наявні рішення не беруть до уваги особливості мікросервісної архітектури, а отже не надає можливостей для централізованої обробки та збереження даних.

Проаналізувавши наявні рішення можна виділити проблему використання цих способів в системі з мікросервісною архітектурою. Такі застосунки потребують централізованого збереження змін в системі, адже це

					КП.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		20

надає можливість обробляти отримані дані в одному місці. В даній дипломній роботі буде розроблено архітектурне рішення, яке враховує особливості мікросервісної архітектури.

1.4 Аналіз вимог до програмного забезпечення

В рамках даного дипломного проєкту розробляється декілька програмних систем – бібліотека відстеження змін та прототип вебсайту для розміщення товарів, який використовує розроблену бібліотеку. Отже, необхідно розробити вимоги для обох цих систем.

1.4.1 Розроблення функціональних вимог

В таблицях 1.1 – 1.12 описано функціональні вимоги до прототипу вебсайту.

Таблиця 1.1 – Опис функціональної вимоги REQ-1

Унікальний ідентифікатор	REQ-1
Назва	Реєстрація
Опис	В системі має бути передбачена можливість додавання користувачів до системи. Для цього користувачу потрібно створити логін, який має бути унікальним в рамках системи, та пароль.

Таблиця 1.2 – Опис функціональної вимоги REQ-2

Унікальний ідентифікатор	REQ-2
Назва	Аутентифікація
Опис	В системі має бути передбачена можливість перевірки чи справді користувач є тим, за кого себе видає та авторизувати його в системі. Для цього користувач має надати свій пароль та логін.

Таблиця 1.3 – Опис функціональної вимоги REQ-3

Унікальний ідентифікатор	REQ-3
Назва	Перегляд списку товарів
Опис	В системі має бути передбачена можливість переглядати список наявних товарів. Для цього користувач має бути авторизованим як звичайний користувач. Список має містити в собі наступну інформацію по кожному товару: - назва товару; - ціна товару в умовних грошових одиницях; - логін власника.

Таблиця 1.4 – Опис функціональної вимоги REQ-4

Унікальний ідентифікатор	REQ-4
Назва	Видалення товару
Опис	В системі має бути передбачена можливість видаляти товари з системи. Для цього користувач має бути авторизованим як звичайний користувач та має бути власником товару.

Таблиця 1.5 – Опис функціональної вимоги REQ-5

Унікальний ідентифікатор	REQ-5
Назва	Детальний опис товару
Опис	В системі має бути передбачена можливість переглянути детальну інформацію про товар. Для цього користувач має бути авторизованим як звичайний користувач. Детальний опис товару має містити наступну інформацію: - назва товару; - ціна товару в умовних грошових одиницях; - логін власника; - опис товару.

Таблиця 1.6 – Опис функціональної вимоги REQ-6

Унікальний ідентифікатор	REQ-6
Назва	Створення товару
Опис	В системі має бути передбачена можливість створити товар. Для цього користувач має бути авторизованим як звичайний користувач. Для створення товару користувач має ввести наступні дані: - назва товару; - опис товару; - ціна товару в умовних грошових одиницях. Користувач є власником створеного товару.

Таблиця 1.7 – Опис функціональної вимоги REQ-7

Унікальний ідентифікатор	REQ-7
Назва	Редагування товару
Опис	В системі має бути передбачена можливість відредагувати товар. Для цього користувач має бути авторизованим як звичайний користувач та має бути власником товару. Користувач повинен мати можливість редагувати назву, опис, ціну товару.

Таблиця 1.8 – Опис функціональної вимоги REQ-8

Унікальний ідентифікатор	REQ-8
Назва	Збереження інформації про видалення товару
Опис	В системі має бути передбачена можливість зберігати інформацію про видалення товарів. Зберігається наступна інформація: - ідентифікатор користувача, що видалив товар; - дата та час, коли відбулося видалення; - назва таблиці в базі даних, де відбулося видалення; - ідентифікатор запису.

Таблиця 1.9 – Опис функціональної вимоги REQ-9

Унікальний ідентифікатор	REQ-9
Назва	Збереження інформації про редагування товару

Продовження таблиці 1.9

Опис	В системі має бути передбачена можливість зберігати інформацію про зміну товарів. Зберігається наступна інформація: - ідентифікатор користувача, що редагував товар; - дата та час, коли відбулося редагування; - назва таблиці в базі даних, де відбулося редагування; - ідентифікатор запису. Має бути передбачена можливість зберігати інформацію про зміни в колонках таблиці. Має зберігатися наступна інформація: - назви колонок, що змінилися; - старі значення у строковому представленні; - нові значення у строковому представленні; - тип даних. Мають відстежуватися колонки, які містять наступну інформацію про товар: - опис товару; - ціна товару; - назва товару.
------	--

Таблиця 1.10 – Опис функціональної вимоги REQ-10

Унікальний ідентифікатор	REQ-10
Назва	Збереження інформації про створення товару
Опис	В системі має бути передбачена можливість зберігати інформацію про створення товарів. Зберігається наступна інформація: - ідентифікатор користувача, що створив товар; - дата та час, коли відбулося створення; - назва таблиці в базі даних, де відбулося створення; - ідентифікатор запису.

Таблиця 1.11 – Опис функціональної вимоги REQ-11

Унікальний ідентифікатор	REQ-11
Назва	Перегляд списку зі змінами

Продовження таблиці 1.11

Опис	В системі має бути передбачена можливість перегляду списку зі змінами, що відбулися в системі. Для цього користувач має бути авторизованим як адміністратор. В списку повинна бути представлена наступна інформація: - тип зміни (видалення, створення, редагування); - логін користувача, що вніс зміну; - дата та час, коли відбулася зміна; - назва таблиці бази даних, в якій відбулася зміна; - ідентифікатор запису. Для змін з типом редагування повинна також бути надана наступна інформація, якщо вона наявна: - назви колонок таблиці, де відбулося редагування; - старі значення, що знаходилося в колонках; - нові значення, які з'явилися в результаті редагування.
------	---

Таблиця 1.12 – Опис функціональної вимоги REQ-12

Унікальний ідентифікатор	REQ-12
Назва	Фільтрація списку зі змінами
Опис	В системі має бути передбачена можливість фільтрації списку зі змінами. Для цього користувач має бути авторизованим як адміністратор. Користувач має можливість фільтрувати список за наступними критеріями: - початкова дата, коли відбулися зміни; - кінцева дата, коли відбулися зміни; - логін користувача, який вніс зміни. В результаті фільтрації користувач побачить інформацію зі списку зі змінами, відфільтровану за обраними критеріями.

В таблицях 1.13 – 1.17 описано функціональні вимоги до системи відстеження змін.

Таблиця 1.13 – Опис функціональної вимоги REQ-13

Унікальний ідентифікатор	REQ-13
Назва	Підключення моделі до системи відстеження змін
Опис	Плагін має надавати можливість розробнику певної системи підключати модель до системи відстеження змін. Розробник системи може обрати один, або декілька типів подій, що будуть відстежуватися, з наступного списку: - видалення запису; - створення запису; - редагування запису. Також плагін має надавати можливість вказувати назви колонок, зміни в яких будуть відстежуватися в результаті редагування запису. За замовчуванням, якщо розробник явно не вказує типи подій, що мають відстежуватися, та не вказує список колонок, плагін має підключати модель до системи з налаштуваннями на відстеження усіх типів подій та усіх колонок.

Таблиця 1.14 – Опис функціональної вимоги REQ-14

Унікальний ідентифікатор	REQ-14
Назва	Відправлення повідомлення про створення запису
Опис	Плагін має надавати можливість застосунку, що його використовує, відправляти повідомлення про створення запису в таблиці баз даних до системи відстеження змін. Повідомлення має містити наступну інформацію: - тип зміни – створення запису; - ідентифікатор користувача, що створив запис; - дата та час, коли відбулося створення; - назва таблиці бази даних, в якій відбулося створення; - ідентифікатор запису.

Таблиця 1.15 – Опис функціональної вимоги REQ-15

Унікальний ідентифікатор	REQ-15
Назва	Відправлення повідомлення про редагування запису
Опис	Плагін має надавати можливість відправляти повідомлення про редагування запису в таблиці до системи відстеження змін. Повідомлення має містити наступну інформацію: - тип зміни – редагування; - ідентифікатор користувача; - дата та час, коли відбулося редагування; - назва таблиці бази даних, в якій відбулося редагування; - ідентифікатор запису. Плагін має надавати можливість додавати до повідомлення інформацію про зміни в колонках. Для кожної колонки має надаватися наступна інформація: - назва колонки; - тип даних; - старе значення у строковому представлені; - нове значення у строковому представлені.

Таблиця 1.16 – Опис функціональної вимоги REQ-16

Унікальний ідентифікатор	REQ-16
Назва	Відправлення повідомлення про видалення запису
Опис	Плагін має надавати можливість застосунку, що його використовує, відправляти повідомлення про видалення запису в таблиці баз даних до системи відстеження змін. Повідомлення має містити наступну інформацію: - тип зміни – видалення запису; - ідентифікатор користувача, що видалив запис; - дата та час, коли відбулося видалення; - назва таблиці бази даних, в якій відбулося видалення; - ідентифікатор запису.

Таблиця 1.17 – Опис функціональної вимоги REQ-17

Унікальний ідентифікатор	REQ-17
Назва	Збереження змін, що відбуваються в системі
Опис	Система відстеження змін має надавати можливість іншій системі, що її використовує, зберігати зміни, які відбуваються в таблицях баз даних її сервісів. Має зберігатися наступна інформація: - тип зміни (видалення, створення, редагування); - ідентифікатор користувача, що вніс зміну; - дата та час внесення зміни; - назва таблиці бази даних, де відбулася зміна; - ідентифікатор запису.

В системі прототипу вебсайту було виділено три типи акторів:

- неавторизований користувач;
- звичайний користувач;
- адміністратор.

В таблицях 1.18 – 1.27 описано варіанти використання прототипу вебсайту.

Таблиця 1.18 – Опис варіанта використання UC-1

Унікальний ідентифікатор	UC-1
Назва	Реєстрація
Опис	Користувач реєструється в системі
Учасники	Неавторизований користувач
Передумови	- користувач не є авторизованим в системі; - користувач знаходиться на сторінці реєстрації.
Постумови	- в системі створено обліковий запис звичайного користувача; - користувач є авторизованим в системі як звичайний користувач; - користувач знаходиться на сторінці зі списком товарів.
Сценарій	- користувач вводить логін у відповідне поле; - користувач вводить пароль у відповідне поле; - користувач натискає на кнопку реєстрації; - у випадку коректного введення даних користувач потрапляє на сторінку зі списком товарів.

Таблиця 1.19 – Опис варіанта використання UC-2

Унікальний ідентифікатор	UC-2
Назва	Вхід у систему звичайного користувача
Опис	Користувач входить в систему як звичайний користувач
Учасники	Неавторизований користувач
Передумови	- користувач не є авторизованим у системі; - користувач знаходиться на сторінці входу у систему для звичайного користувача.
Постумови	- користувач є авторизованим в системі як звичайний користувач. - користувач знаходиться на сторінці зі списком товарів.
Сценарій	- користувач вводить логін у відповідне поле; - користувач вводить пароль у відповідне поле; - користувач натискає на кнопку входу у систему; - у випадку успішної ідентифікації, аутентифікації здійснюється авторизація користувача та він потрапляє на сторінку зі списком товарів.

Таблиця 1.20 – Опис варіанта використання UC-3

Унікальний ідентифікатор	UC-3
Назва	Вхід у систему адміністратора
Опис	Користувач авторизується як адміністратор.
Учасники	Неавторизований користувач
Передумови	- користувач не є авторизованим у системі; - користувач знаходиться на сторінці входу у систему для адміністратора.
Постумови	- користувач є авторизованим в системі як адміністратор. - користувач знаходиться на сторінці зі списком змін у системі.
Сценарій	- користувач вводить логін у відповідне поле; - користувач вводить пароль у відповідне поле; - користувач натискає на кнопку входу у систему; - у випадку успішної ідентифікації та аутентифікації здійснюється авторизація користувача та він потрапляє на сторінку зі списком змін у системі.

Таблиця 1.21 – Опис варіанта використання UC-4

Унікальний ідентифікатор	UC-4
Назва	Перегляд списку товарів
Опис	Звичайний користувач переглядає список товарів.
Учасники	Звичайний користувач
Передумови	Користувач є авторизованим в системі як звичайний користувач.
Постумови	Користувач знаходиться на головній сторінці вебсайту.
Сценарій	- користувач переходить на головну сторінку вебсайту; - користувач бачить список товарів.

Таблиця 1.22 – Опис варіанта використання UC-5

Унікальний ідентифікатор	UC-5
Назва	Видалення товару
Опис	Звичайний користувач видаляє товар зі списку товарів
Учасники	Звичайний користувач
Передумови	- користувач є авторизованим в системі як звичайний користувач; - користувач знаходиться на сторінці зі списком товарів; - користувач є власником товару, який збирається видалити.

Продовження таблиці 1.22

Постумови	- користувач знаходиться на сторінці зі списком товарів; - товар видалено з системи.
Сценарій	- користувач натискає на кнопку видалення товару; - товар видаляється з системи.

Таблиця 1.23 – Опис варіанта використання UC-6

Унікальний ідентифікатор	UC-6
Назва	Детальний перегляд товару
Опис	Звичайний користувач переглядає детальну інформацію по товару
Учасники	Звичайний користувач
Передумови	- користувач є авторизованим в системі як звичайний користувач; - користувач знаходиться на сторінці зі списком товарів.
Постумови	- користувач знаходиться на сторінці з детальним описом товару;
Сценарій	- користувач натискає на кнопку детального опису товару; - користувач потрапляє на сторінку з детальним описом товару.

Таблиця 1.24 – Опис варіанта використання UC-7

Унікальний ідентифікатор	UC-7
Назва	Редагування товару
Опис	Звичайний користувач редагує інформацію про товар
Учасники	Звичайний користувач
Передумови	- користувач є авторизованим в системі як звичайний користувач; - користувач знаходиться на сторінці з детальним описом товару; - користувач є власником товару, який збирається редагувати.
Постумови	- користувач знаходиться на сторінці з детальним описом товару; - інформація про товар оновлена.
Сценарій	- користувач натискає на кнопку редагування; - користувач опиняється на сторінці редагування товару; - користувач редагує інформацію про товар; - користувач натискає на кнопку збереження змін.

Таблиця 1.25 – Опис варіанта використання UC-8

Унікальний ідентифікатор	UC-8
Назва	Створення товару
Опис	Звичайний користувач створює товар
Учасники	Звичайний користувач
Передумови	- користувач є авторизованим в системі як звичайний користувач; - користувач знаходиться на сторінці зі списком товарів.
Постумови	- користувач знаходиться на сторінці зі списком товарів; - товар додано в систему.
Сценарій	- користувач натискає на кнопку створення нового товару; - користувач опиняється на сторінці створення товару; - користувач вводить назву, опис, ціну товару; - користувач натискає на кнопку створення товару.

Таблиця 1.26 – Опис варіанта використання UC-9

Унікальний ідентифікатор	UC-9
Назва	Перегляд змін в системі
Опис	Адміністратор системи переглядає список змін в системі.
Учасники	Адміністратор
Передумови	Користувач є авторизованим в системі як адміністратор.
Постумови	Користувач знаходиться на сторінці зі списком змін
Сценарій	- користувач переходить на головну сторінку вебсайту; - користувач опиняється на сторінці зі списком змін в системі.

Таблиця 1.27 – Опис варіанта використання UC-10

Унікальний ідентифікатор	UC-10
Назва	Фільтрація змін в системі
Опис	Адміністратор відфільтровує список змін в системі.
Учасники	Адміністратор
Передумови	- користувач є авторизованим в системі як адміністратор; - користувач знаходиться на сторінці зі списком змін.

Продовження таблиці 1.27

Постумови	- користувач знаходиться на сторінці зі списком змін; - список змін відфільтрований за обраними критеріями.
Сценарій	- користувач натискає на кнопку фільтрації; - користувач обирає критерії для фільтрації; - користувач натискає на кнопку застосування фільтрації.

Варіанти використання прототипу вебсайту зображено у документі «Схема структурна варіантів використання».

В системі відстеження змін було виділено два типи акторів:

- розробник Django-системи;
- Django-система.

В таблицях 1.28 – 1.32 описано варіанти використання системи відстеження змін.

Таблиця 1.28 – Опис варіанта використання UC-11

Унікальний ідентифікатор	UC-11
Назва	Підключення моделі до системи відстеження змін
Опис	Розробник системи підключає модель до системи відстеження змін
Учасники	Розробник Django-системи
Передумови	- встановлено відповідний пагін; - вказано налаштування для підключення сервісу до системи відстеження змін.
Постумови	Модель підключена до системи відстеження змін
Сценарій	- розробник вказує назву моделі, яку потрібно приєднати; - вказує типи подій, що треба відстежити; - вказує колонки, що треба відстежити.

Таблиця 1.29 – Опис варіанта використання UC-12

Унікальний ідентифікатор	UC-12
Назва	Відправлення повідомлення про створення запису
Опис	Django-система відправляє повідомлення про створення запису в базі даних у систему відстеження змін.
Учасники	Django-система
Передумови	Модель, що прив'язана до таблиці, підключена до системи відстеження змін з налаштуванням на відстеження створення запису
Постумови	Повідомлення відправлене у систему відстеження змін
Сценарій	- Django-система створює запис у базі даних; - відправляється повідомлення про створення запису.

Таблиця 1.30 – Опис варіанта використання UC-13

Унікальний ідентифікатор	UC-13
Назва	Відправлення повідомлення про редагування запису
Опис	Django-система відправляє повідомлення про редагування запису в базі даних у систему відстеження змін.

Продовження таблиці 1.30

Учасники	Django-система
Передумови	Модель, що прив'язана до таблиці, підключена до системи відстеження змін з налаштуванням на відстеження редагування запису
Постумови	Повідомлення відправлене у систему відстеження змін
Сценарій	- Django-система редагує запис у базі даних; - відправляється повідомлення про редагування запису.

Таблиця 1.31 – Опис варіанта використання UC-14

Унікальний ідентифікатор	UC-14
Назва	Відправлення повідомлення про видалення запису
Опис	Django-система відправляє повідомлення про видалення запису в базі даних у систему відстеження змін.
Учасники	Django-система
Передумови	Модель, що прив'язана до таблиці, підключена до системи відстеження змін з налаштуванням на відстеження видалення запису
Постумови	Повідомлення відправлене у систему відстеження змін

Продовження таблиці 1.31

Сценарій	- Django-система видаляє запис у базі даних; - відправляється повідомлення про видалення запису.
----------	---

Таблиця 1.32 – Опис варіанта використання UC-15

Унікальний ідентифікатор	UC-15
Назва	Збереження повідомлення
Опис	Django-система зберігає повідомлення про зміну
Учасники	Django-система
Передумови	В систему надійшло повідомлення про зміну
Постумови	Повідомлення збережене
Сценарій	- повідомлення вилучається з черги повідомлень; - повідомлення зберігається.

Варіанти використання системи відстеження змін зображено у документі «Схема структурна варіантів використань».

На рисунку 1.2 зображено матрицю залежності між функціональними вимогами та варіантами використання.

	REQ-1	REQ-2	REQ-3	REQ-4	REQ-5	REQ-6	REQ-7	REQ-8	REQ-9	REQ-10	REQ-11	REQ-12	REQ-13	REQ-14	REQ-15	REQ-16	REQ-17
UC-1																	
UC-2																	
UC-3																	
UC-4																	
UC-5																	
UC-6																	
UC-7																	
UC-8																	
UC-9																	
UC-10																	
UC-11																	
UC-12																	
UC-13																	
UC-14																	
UC-15																	

Рисунок 1.2 – Матриця залежності між функціональними вимогами та варіантами використання

1.4.2 Розроблення нефункціональних вимог

Передбачено наступні нефункціональні вимоги:

- прототип вебсайту має бути розроблений з використанням мікросервісної архітектури;
- мікросервіси, що підключають пагін мають бути розроблені з використанням Django;
- мікросервіси, що підключають пагін використовують вбудовану Django-ORM;
- мікросервіси використовують PostgreSQL як СУБД.

1.4.3 Постановка комплексу завдань модулю

Основною метою розробки є створення архітектурного рішення, що надає можливість відстежувати зміни, що відбуваються внаслідок взаємодії користувачів з системою та адаптувати запропоноване рішення до потреб та особливостей мікросервісної архітектури.

В рамках цієї роботи потрібно розробити систему для відстеження змін та прототип вебзастосунку для розміщення товарів. Прототип вебзастосунку необхідно розробити для того, щоб продемонструвати можливості системи відстеження змін.

Необхідно відзначити обмеження можливостей системи відстеження змін. Система не надає можливості відстежувати зміни, що відбуваються при прямій взаємодії з базою даних, наприклад при виклику збережених процедур. Система відстежує лише ті зміни, що відбуваються в результаті виклику методів save та delete класу django.db.models.Model.

Описані вище обмеження не є критичними для системи, оскільки в більшості випадків Django взаємодіє з базою даних використовуючи функціонал в результаті якого система зберігає зміни.

В рамках розробки системи відстеження змін необхідно виділити наступні задачі:

- реалізація Django-плагіну, що відправляє повідомлення про зміни в системі до брокера повідомлень;
- реалізація Django-плагіну споживача повідомлень, що зберігає повідомлення про зміни.

В рамках розробки прототипу вебсайту потрібно виділити наступні задачі:

- реалізація реєстрації, авторизації користувача;
- реалізація створення, редагування, видалення та перегляду товарів;
- реалізація збереження змін, що відбуваються в таблиці товарів;
- реалізація перегляду змін;
- реалізація фільтрації змін.

1.5 Висновки по розділу

Було проведено аналіз середовища розробки та детально його описано.

Було досліджено наявні архітектурні рішення, виокремлено їх переваги та недоліки та визначено як можна їх покращити.

Було проаналізовано та визначено функціональні вимоги, не функціональні вимоги та сформульовано задачі модулю.

					КПІ.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		49

2 МОДЕЛЮВАННЯ ТА КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Архітектура програмного забезпечення

В рамках даної дипломної роботи розробляється декілька систем — система для відстеження змін та прототип вебсайту, що її використовує.

2.1.1 Система відстеження змін

Архітектура системи відстеження змін спеціалізована для використання у мікросервісних системах. Система складається з трьох складових:

- виробники повідомлень;
- брокер повідомлень;
- споживач повідомлень.

Виробники повідомлень відстежують зміни в системі, формують та відправляють повідомлення про зміни в брокер повідомлень. Кожен мікросервіс, до якого підключено відповідний Django-плагін, має можливість використовувати функціонал виробника повідомлення. Споживач повідомлень вилучає повідомлення з брокера, обробляє їх та зберігає у базі даних. Функціонал споживача прив'язується до одного з сервісів, що існують у системі, для зручного використання інформації про зміни, що зберігаються у базі даних відповідного мікросервіса. Незважаючи на те, що функціонал споживача повідомлень може бути доданий до вже наявного сервісу, споживач повідомлень та сервіс працюють як два різних сервіси.

На рисунку 2.1 зображено схему діаграми послідовності, що відображає послідовність відстеження зміни в базі даних для одного мікросервіса. Актором в даній схемі виступає мікросервіс, який використовує Django-плагін для відстеження змін в своїй базі даних. Необхідно відзначити,

що дана схема актуальна для усіх мікросервісів в системі, що використовують функціонал виробника повідомлень.



Рисунок 2.1 – Діаграма послідовності відстеження зміни в мікросервісі
Представлена архітектура надає наступні переваги:

- до системи відстеження змін підключаються окремі мікросервіси, що надає можливість зручно додавати та вилучати функціонал відстеження змін в окремих частинах системи;
- зміни зберігаються централізовано, що надає можливість швидко та зручно обробляти їх;
- дані передаються через брокер повідомлень, що забезпечує швидкість відстеження змін, а також не сповільнює мікросервіси.

2.1.2 Прототип вебсайту

Вебсайт складається з клієнтської та серверної частин. Серверна частина розробляється з використанням мікросервісної архітектури.

В системі передбачено наступні сервіси:

- authentication service (сервіс аутентифікації);
- products service (сервіс товарів);
- admin service (сервіс адміністратора);
- message broker (брокер повідомлень);
- message consumer (споживач повідомлень);
- API gateway (маршрутизатор запитів).

Маршрутизатор запитів слугує точкою входу для клієнту, він направляє запити, що надходять з клієнтського застосунку до інших сервісів.

Сервіс аутентифікації відповідає за керування користувачами. Надає функціонал для додавання, аутентифікації користувачів, може повертати інформацію по всім користувачам в системі.

Сервіс товарів відповідає за взаємодію користувачів з товарами. Містить функціонал для створення, видалення, редагування, перегляду товарів. Цей сервіс підключено до системи відстеження змін, отже він містить функціонал виробника повідомлень, та надсилає повідомлення в брокер повідомлень.

Споживач повідомлень — це сервіс з системи відстеження змін, який відповідає за вилучення повідомлень про створення записів з брокеру повідомлень та збереження їх у базі даних.

Сервіс адміністратора відповідає за перегляд, фільтрацію даних, що надходять з системи відстеження змін.

Клієнт взаємодіє лише з маршрутизатором запитів. Маршрутизатор запитів перенаправляє запит клієнта до одного з трьох сервісів — сервісу аутентифікації, сервісу товарів, або сервісу адміністратора. В системі передбачено три бази даних. Сервіс аутентифікації та сервіс товарів мають по окремій базі даних. Сервіс адміністратора та споживач повідомлень мають спільну базу даних, оскільки обидва взаємодіють з інформацією про зміни в системі. Споживач повідомлень та сервіс товарів взаємодіють один з одним

					КПІ.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		52

за допомогою споживача повідомлень в рамках функціонування системи для відстеження змін. На рисунку 2.2 зображено схему взаємодії мікросервісів.

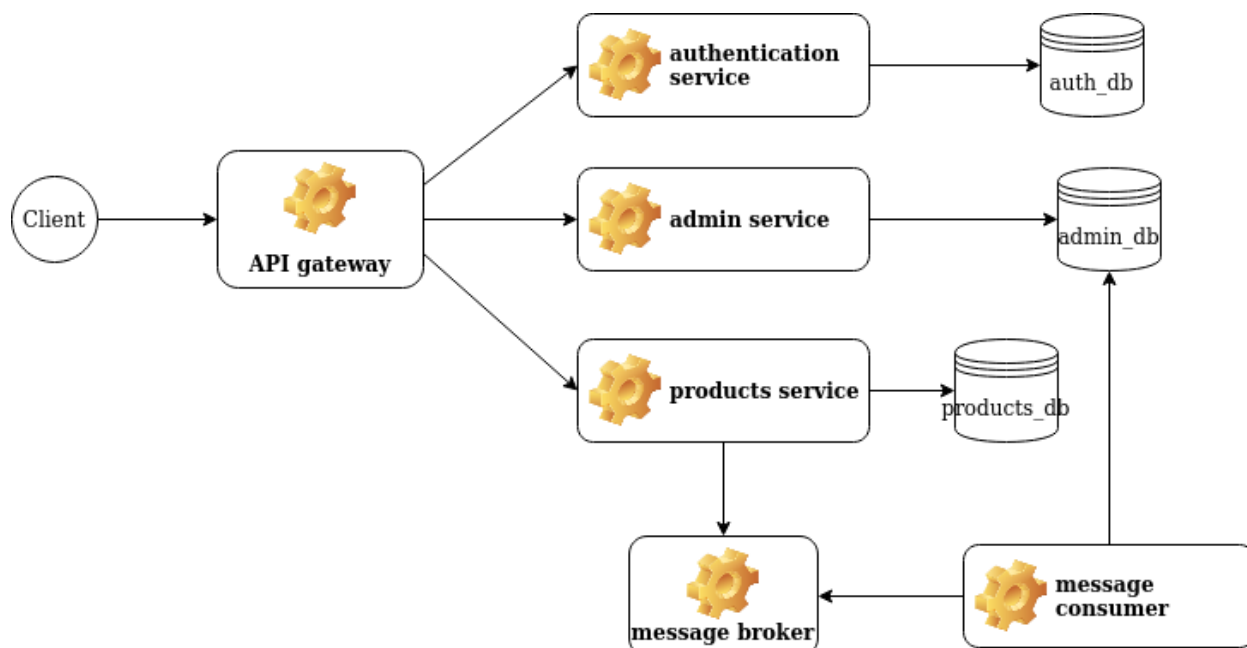


Рисунок 2.2 – Діаграма взаємодії мікросервісів

Оскільки в даній системі передбачено реєстрацію користувача, а доступ до ресурсів надається тільки після входу користувача в систему, необхідно описати процес аутентифікації користувача при запитах до певного сервісу. На рисунку 2.3 представлено діаграму аутентифікації користувача для будь-якого запиту до системи, що потребує аутентифікації. Актором в даній схемі виступає клієнтський застосунок.

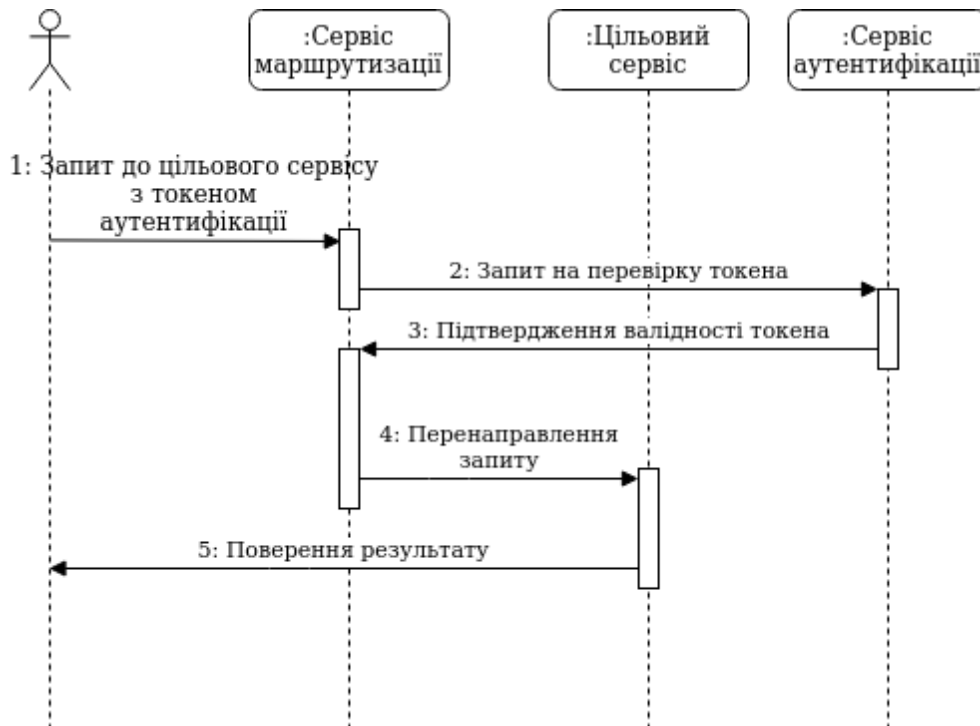


Рисунок 2.3 – Діаграма послідовності аутентифікації користувача

2.2 Конструювання програмного забезпечення

В рамках системи відстеження змін необхідно розробити Django-плагін виробника повідомлень та Django-плагін споживача повідомлень.

2.2.1 Django-плагін виробника повідомлень

Завдання виробника полягає у відстеженні подій створення, видалення, редагування моделей. При редагуванні моделей необхідно відстежувати нові та старі значення полів, що змінюються. Також необхідно визначити інтерфейс для підключення моделей до системи відстеження змін, з можливістю конфігурування типів змін та набору полів, що будуть відстежуватися.

Подія редагування є більш комплексною за інші типи подій, тому спочатку необхідно розробити спосіб для відстеження редагування, а вже потім, використовуючи як базу отриманий підхід, створити функціонал для відстеження видалення та створення записів. Платформа Django надає

широкий простір для вирішення поставленої задачі, отже потрібно розглянути декілька наявних варіантів.

Перший підхід — розширити функціонал базового класу `django.db.models.Model`. Для цього необхідно перевизначити методи `__init__` та `save`. При ініціалізації об'єкту в методі `__init__` потрібно зберегти початковий стан моделі, а при виклику методу `save`, після відпрацювання батьківського методу `save`, порівняти початкові та поточні значення моделі. Цей підхід має певні недоліки:

- необхідно враховувати те, що після ініціалізації об'єкту моделі значення полів більше не відповідають реальним значенням в базі даних, а отже якщо в процесі роботи з моделлю значення в базі даних будуть змінені, то при відстеженні зміни початковий стан моделі може не відповідати реальним значенням, що знаходяться в таблиці на момент збереження;

- необхідно враховувати те, що Django надає багато методів для роботи з моделями, а отже необхідно розширити деякі з них для коректної роботи, наприклад метод `refresh_from_db`, що перезаписує поля моделі поточними значеннями з бази даних;

- розробнику системи необхідно розширювати функціонал моделі, що надається плагіном, а не `django.db.models.Model`, що додатково навантажує процес підключення моделі до системи відстеження змін.

Можливою модифікацією цього способу є визначення початкового стану моделі в момент перед викликом батьківського методу `save`, що позбавляє систему від перших двох недоліків, але тоді з'являється необхідність додаткового запиту до бази даних в момент збереження, а також залишається третій недолік.

Другий спосіб — використання сигналів моделей. Django надає можливість підписуватися на події змін в базі даних [10]. Розглянемо сигнал `pre_save`. Функція, що підписується на цей сигнал, спрацює в момент перед збереженням інформації в базі даних, а отже надає можливість

отримати початковий стан моделі. Даний підхід позбавляє необхідності розширювати функціонал базової моделі Django, але залишається проблема з необхідністю робити додатковий запит до бази даних перед збереженням. Також цей підхід має новий недолік — відстеження зміни відбувається в момент перед збереженням інформації, а отже немає гарантії того, що після відправлення повідомлення, відповідна зміна буде внесена в базу даних.

Третій спосіб — використати сигнали `pre_save` та `post_save`. В момент `pre_save` до об'єкту моделі буде додаватися поле, що містить об'єкт з поточним станом рядку в таблиці бази даних. В момент `post_save` буде формуватися повідомлення і відправлятися в брокер повідомлень. Даний підхід має наступні переваги:

- не має необхідності розширювати функціонал моделі;
- не має необхідності відстежувати стан запису в таблиці в процесі взаємодії з об'єктом;
- повідомлення відправляється після збереження в базі даних, отже не має можливості відправити повідомлення про зміну, яка не відбулася.

Також цей підхід має недоліки:

- додатковий запит до бази даних перед збереженням;
- необхідно динамічно додавати поле до об'єкту.

Перший недолік є вимушеним, оскільки в будь-якому випадку, якщо існує необхідність дізнатися точні дані, що знаходяться в базі даних на момент збереження, необхідно витягнути їх з бази даних. Другий недолік не буде явно вираженим для розробника системи.

Враховуючи недоліки та переваги наведених способів, для розробки плагіну буде застосовано третій підхід, оскільки його переваги надають можливість створити зручний інтерфейс, а також забезпечують достовірність даних, що передаються в брокер повідомлень. Оскільки для відстеження редагувань моделей було обрано використання сигналів, то для відстеження

видалення та створення записів буде також використано відповідні сигнали — `post_delete` та `post_save`.

Для спілкування між виробником та споживачем використовується брокер повідомлень Apache Kafka. Формат повідомлень, що передаються — JSON. Даний формат обирається через те, що він є текстовим форматом. Використання бінарного формату дозволило б зменшити об'єм даних, що передаються, але позбавило б можливості розуміти вміст повідомлення без додаткових перетворень. В даній системі об'єм даних, що передається не є критичним, тому бінарний формат не використовується. В таблиці 2.1 представлено структуру повідомлення.

Таблиця 2.1 — Структура повідомлення про зміну

Назва поля	Тип поля	Опис
actor_id	Рядок	Ідентифікатор користувача, що вніс зміну
action	Рядок	Тип зміни, що відбувся (delete, insert, update)
row_id	Рядок	Ідентифікатор рядка таблиці, що змінився
table_name	Рядок	Назва таблиці, в якій відбулася зміна
changes	Список	Список змін полів, що відбулися при редагуванні (update), для delete, insert передається пустий список
action_datetime	Час	Дата та час, коли відбулася зміна

Для відстеження поля `actor_id` необхідно розробити спосіб відстеження поточного користувача системи. Відстеження змін відбувається на рівні Django моделей, а робота з користувачем системи відбувається на рівні обробки запиту. Django не надає вбудованого функціоналу для доступу до запиту з рівня моделей. Для вирішення цієї проблеми можна використати Django middleware.

Django middleware — це набір функцій, що послідовно оброблюють запит перед тим, як він опиниться на рівні контролерів. Django містить набір вбудованих middleware, а також надає можливість створювати власні.

Django-плагін виробника повідомлень буде містити middleware, що буде зберігати поточний запит користувача, для подальшого оброблення на рівні моделі. Недоліком даного підходу є те, що доводиться зберігати глобальну змінну для відстеження користувача. Іншим підходом було б явно передавати поточного користувача з контролеру в рівень моделей, але це б ускладнило інтерфейс для підключення моделей, а також розробникам системи довелося б реалізовувати більшу частину функціоналу самостійно.

2.2.1.1 Опис класів та модулів

Модуль `model_registry` містить основний клас плагіну виробника.

`ModelRegistry` — клас, що є точкою входу для розробника, який підключає сервіс до системи відстеження змін. Містить методи, які відповідають за підключення моделі та конфігурування змін та полів, що мають відстежуватися.

В таблицях 2.2 — 2.3 наведено опис класу `ModelRegistry`.

Таблиця 2.2 — Опис полів класу ModelRegistry

Назва поля	Опис
message_sender_cls	Поле класу, що містить клас, який відповідає за відправку повідомлення.
message_builder_cls	Поле класу, що містить клас, який відповідає за створення повідомлення.
changes_tracker_cls	Поле класу, що містить клас, який відповідає за відстеження змін моделі.
model	Поле об'єкту, що містить клас моделі, яка підключається до системи відстеження змін.
actions	Поле об'єкту, що містить множину дій, що мають відстежуватися.
fields	Поле об'єкту, що містить множину полів, що мають відстежуватися при редагуванні запису.

Таблиця 2.3 — Опис методів класу ModelRegistry

Назва методу	Параметри	Опис
register_model	model — модель, що підключається до системи відстеження змін, actions — множина змін, що відстежується для обраної моделі (insert, update, delete), fields — множина полів, що мають відстежуватися при редагуванні моделі.	Метод класу, що слугує для підключення моделі до системи відстеження змін та конфігурування змін та полів, що мають відстежуватися. В результаті роботи методу модель буде підключена до системи відстеження змін.
validate_model_registry		Перевіряє поля, якими було ініціалізовано об'єкт класу ModelRegistry. При некоректних значеннях викликається ValueError з відповідним повідомленням.
connect_actions		Під'єднує функції для відстеження подій, що знаходяться в полі actions для моделі model.

Продовження таблиці 2.3

get_connectors		Створює та повертає словник, ключами якого є типи подій, а значеннями є об'єкти типу AbstractReceiverConnector.
create_insert_connector		Створює та повертає об'єкт класу InsertActionReceiverConnector.
create_delete_connector		Створює та повертає об'єкт класу DeleteActionReceiverConnector.
create_update_connector		Створює та повертає об'єкт класу UpdateActionReceiverConnector.

Модуль receiver_connectors містить класи, що відповідають за створення та підключення функцій до сигналів Django.

AbstractReceiverConnector — базовий клас для класів, що під'єднують відстеження подій для моделі.

В таблицях 2.4 — 2.5 описано клас AbstractReceiverConnector.

Таблиця 2.4 — Опис полів класу AbstractReceiverConnector

Назва поля	Опис
model	Поле об'єкту, що містить клас моделі, яка підключається до системи відстеження змін.
message_sender_cls	Поле об'єкту, що містить клас, який відповідає за відправку повідомлення.
message_builder_cls	Поле об'єкту, що містить клас, який відповідає за створення повідомлення.

Таблиця 2.5 — Опис методів класу AbstractReceiverConnector

Назва методу	Параметри	Опис
connect		Абстрактний метод для підключення функції, що відстежує зміни до Django сигналу.

DeleteActionReceiverConnector — клас, що слугує для під'єднання відстеження події видалення для моделі.

В таблицях 2.6 — 2.7 описано клас DeleteActionReceiverConnector.

Таблиця 2.6 — Опис полів класу DeleteActionReceiverConnector

Назва поля	Опис
model	Поле об'єкту, що містить клас моделі, для якої потрібно відстежувати подію.
message_sender_cls	Поле об'єкту, що містить клас, який відповідає за відправку повідомлення.
message_builder_cls	Поле об'єкту, що містить клас, який відповідає за створення повідомлення.

Таблиця 2.7 — Опис методів класу DeleteActionReceiverConnector

Назва методу	Параметри	Опис
connect		Під'єднує модель та функцію, що відстежує подію видалення до сигналу post_delete
build_send_message_receiver		Повертає функцію, що відстежує подію видалення.

InsertActionReceiverConnector — клас, що слугує для під'єднання відстеження події створення для моделі.

В таблицях 2.8 — 2.9 описано клас InsertActionReceiverConnector.

Таблиця 2.8 — Опис полів класу InsertActionReceiverConnector

Назва поля	Опис
model	Поле об'єкту, що містить клас моделі, для якої потрібно відстежувати подію створення.
message_sender_cls	Поле об'єкту, що містить клас, який відповідає за відправку повідомлення.
message_builder_cls	Поле об'єкту, що містить клас, який відповідає за створення повідомлення.

Таблиця 2.9 — Опис методів класу InsertActionReceiverConnector

Назва методу	Параметри	Опис
connect		Під'єднує модель та функцію, що відстежує подію створення до сигналу post_save.
build_send_message_receiver		Повертає функцію, що відстежує подію створення.

UpdateActionReceiverConnector — клас, що слугує для під'єднання відстеження події редагування для моделі.

В таблицях 2.10 — 2.11 описано клас UpdateActionReceiverConnector.

Таблиця 2.10 — Опис полів класу UpdateActionReceiverConnector

Назва поля	Опис
model	Поле об'єкту, що містить клас моделі, для якої потрібно відстежувати подію редагування.
message_sender_cls	Поле об'єкту, що містить клас, який відповідає за відправку повідомлення.
message_builder_cls	Поле об'єкту, що містить клас, який відповідає за створення повідомлення.
fields	Поле об'єкту, що містить множину полів, зміни яких потрібно відстежувати.

Таблиця 2.11 — Опис методів класу UpdateActionReceiverConnector

Назва методу	Параметри	Опис
connect		Під'єднує модель та функцію, що відстежує подію створення до сигналу post_save, а також під'єднує функцію set_original_instance до pre_save сигналу.
build_send_message_receiver		Повертає функцію, що відстежує подію редагування.

В таблиці 2.12 описано функції, що знаходяться в модулі receiver_connectors.

Таблиця 2.12 — Опис функцій модуля receiver_connectors

Назва функції	Параметри	Опис
send_delete_action_message	sender — клас об'єкту моделі, що викликала сигнал post_delete, instance — об'єкт що викликав сигнал post_delete.	Функція, що підписується на сигнал post_delete. Слугує для створення та відправлення повідомлення про видалення запису.

Продовження таблиці 2.12

send_insert_action_message	sender — клас об'єкту моделі, що викликала сигнал post_save, instance — об'єкт, що викликав сигнал post_save, created — параметр, що містить значення істини, якщо сигнал викликано після створення запису в таблиці, інакше повертає хибу.	Функція, що підписується на сигнал post_save. Слугує для створення та відправлення повідомлення про створення запису.
set_original_instance	sender — клас об'єкту моделі, що викликала сигнал pre_save, instance — об'єкт що викликав сигнал pre_save.	Функція, що підписується на сигнал pre_save. Слугує для отримання та встановлення поточного стану запису в базі даних на об'єкт моделі.

Продовження таблиці 2.12

send_update_action_message	sender — клас об'єкту моделі, що викликала сигнал post_save, instance — об'єкт, що викликав сигнал post_save, created — параметр, що містить значення істини, якщо сигнал викликано після створення запису в таблиці, інакше повертає хибу, update_fields — список полів, що були вказані як ті, що потрібно оновити.	Функція, що підписується на сигнал post_save. Слугує для відстеження змінених полів в результаті редагування запису, створення повідомлення про редагування та відправлення повідомлення.
----------------------------	---	--

Модуль messages містить класи, що слугують для відображення структури повідомлення про зміну.

В таблицях 2.13 — 2.14 описано класи ModelChangeMessage та FieldChange.

Таблиця 2.13 — Опис полів класу ModelChangeMessage

Назва поля	Опис
actor_id	Ідентифікатор користувача, що вніс зміну.
action	Тип зміни, що відбулася (insert, update, delete).
row_id	Ідентифікатор запису, що змінився.
table_name	Назва таблиці, де відбулася зміна.
changes	Список змін, що відбулися в полях.
action_datetime	Дата та час, коли відбулися зміни.

Таблиця 2.14 — Опис полів класу FieldChange

Назва поля	Опис
Field	Назва поля, що змінилося.
old_value	Значення, що знаходилося в таблиці до внесення зміни.
new_value	Нове значення поля в таблиці.
field_type	Тип поля.

В модулі message_serializer міститься клас, що відповідає за перетворення повідомлення в формат JSON.

MessageSerializer — клас, що слугує для перетворення повідомлення в JSON-строку.

В таблицях 2.15 — 2.16 описано клас MessageSerializer.

Таблиця 2.15 — Опис полів класу MessageSerializer

Назва поля	Опис
json_encoder_cls	Поле класу, що містить клас, який відповідає за серіалізацію словників в JSON.
message	Поле об'єкту, що містить повідомлення.

Таблиця 2.16 — Опис методів класу MessageSerializer

Назва методу	Параметри	Опис
to_json		Серіалізує повідомлення у JSON та повертає отримане значення.

В модулі `message_sender` знаходиться клас, що відповідає за відправлення повідомлення у брокер повідомлень.

`MessageSender` — клас, що відповідає за серіалізацію та відправлення повідомлення у брокер повідомлень.

В таблицях 2.17 — 2.18 описано клас `MessageSender`.

Таблиця 2.17 — Опис полів класу MessageSender

Назва поля	Опис
message_serializer_cls	Поле класу, що містить клас, який відповідає за серіалізацію повідомлення.
message	Поле об'єкту, що містить повідомлення, яке потрібно відправити у брокер повідомлень.

Таблиця 2.18 — Опис методів класу MessageSender

Назва методу	Параметри	Опис
send_message		Відправляє повідомлення у брокер повідомлень.

В модулі message_builders знаходяться класи, що відповідають за створення повідомлення.

MessageBuilder — клас, що відповідає за створення повідомлення.

В таблицях 2.19 — 2.20 описано клас MessageBuilder.

Таблиця 2.19 — Опис полів класу MessageBuilder

Назва поля	Опис
instance	Поле об'єкту, що містить об'єкт, який було змінено/створено/видалено.

Таблиця 2.20 — Опис методів класу MessageBuilder

Назва методу	Параметри	Опис
build_message	action — зміна, що відбулася, changes — список змін, що відбулися з полями	Створює повідомлення про зміну та повертає його.
get_actor_id		Повертає ідентифікатор користувача, що вніс зміну.
get_row_id		Повертає ідентифікатор запису, що було створено/видалено/змінено.

Продовження таблиці 2.20

get_table_name		Повертає назву таблиці бази даних, де відбулася зміна.
get_action_datetime		Повертає дату та час, коли відбулася зміна.

ChangeBuilder — клас, що відповідає за створення частини повідомлення з інформацією про зміну в полі запису.

В таблицях 2.21 — 2.22 описано клас ChangeBuilder.

Таблиця 2.21 — Опис полів класу ChangeBuilder

Назва поля	Опис
instance	Об'єкт, в записі якого відбулася зміна.
field	Назва поля, значення якого змінилося.
old_value	Старе значення поля, що змінилося.

Таблиця 2.22 — Опис методів класу ChangeBuilder

Назва методу	Параметри	Опис
build_change		Створює та повертає об'єкт типу FieldChange.
get_field		Повертає назву поля, що змінилося.
get_old_value		Повертає старе значення поля.
get_new_value		Повертає нове значення поля.
get_field_type		Повертає тип поля.

В модулі `current_user` знаходиться функціонал для відстеження поточного користувача.

`CurrentRequestUserMiddleware` — клас, що виступає `middleware`, який користувач плагіну має використовувати для відстеження поточного користувача.

В таблицях 2.23 — 2.24 описано клас `CurrentRequestUserMiddleware`.

Таблиця 2.23 — Опис полів класу `CurrentRequestUserMiddleware`

Назва поля	Опис
<code>current_request</code>	Поле класу, що зберігає посилання на об'єкт поточного запиту.
<code>get_response</code>	Поле об'єкту, що містить функцію, яка приймає об'єкт запиту та повертає результат об'єкт відповіді.

Таблиця 2.24 — Опис методів класу `CurrentRequestUserMiddleware`

Назва методу	Параметри	Опис
<code>__call__</code>	<code>request</code> — об'єкт запиту	Встановлює в поле <code>current_request</code> значення параметру <code>request</code> . Після відпрацювання функції <code>get_response</code> , встановлює порожнє значення.

В таблиці 2.25 описано функції, що знаходяться в модулі `current_user`.

Таблиця 2.25 — Опис функцій модуля current_user

Назва функції	Параметри	Опис
get_current_user		Повертає поточного користувача, що знаходиться в полі current_request класу CurrentRequestUserMiddleware.

В модулі changes_tracker знаходяться класи, що відповідають за відстеження змін полів при редагуванні записів.

ChangesTrackerResult — клас, що відповідає за збереження результату відстеження змін полів.

В таблиці 2.26 описано клас ChangesTrackerResult.

Таблиця 2.26 — Опис полів класу ChangesTrackerResult

Назва поля	Опис
tracked_changes	Поле об'єкту, що містить інформацію про зміни в полях.
is_instance_updated	Поле об'єкту, що містить значення істини, якщо відбулися зміни в записі, інакше хибя.

ChangesTracker — клас, що відповідає за визначення полів моделі, що були змінені в результаті редагування запису.

В таблицях 2.27 — 2.28 описано клас ChangesTracker.

Таблиця 2.27 — Опис полів класу ChangesTracker

Назва поля	Опис
change_builder_cls	Поле класу, що містить клас, який відповідає за створення частини повідомлення про зміну поля.
instance	Поле об'єкту, що містить об'єкт, який було змінено.
original_instance	Поле об'єкту, що містить об'єкт, який відповідає поточному значенню запису в базі даних.
fields_to_track	Поле об'єкту, що містить множину полів, що мають відстежуватися при редагуванні запису.

Таблиця 2.28 — Опис методів класу ChangesTracker

Назва методу	Параметри	Опис
get_changes		Повертає зміни, що потрібно відстежувати відповідно до полів вказаних в полі fields_to_track.
get_fields_changes		Знаходить усі поля, що були змінені в записі та повертає список з відповідними значеннями.

2.2.2 Django-плагін споживача повідомлень

Споживач повідомлень призначений для вилучення повідомлень з брокеру повідомлень та збереження їх у базі даних. Плагін містить команду, що дозволяє запускати окремий процес споживача повідомлень. Може бути підключений до вже наявного Django-застосунок, або використовуватися як окремий застосунок. Такий підхід надає можливість використовувати вже наявні в плагіні моделі для роботи з даними.

Оскільки в брокер повідомлень можуть надсилатися будь-які дані не має гарантії, що вони матимуть формат, який очікує застосунок, отже необхідно перевірити дані на відповідність формату повідомлення, що має надсилатися виробником повідомлень. Споживач використовує сторонню бібліотеку `marshmallow` для валідації даних, що вилучаються з брокеру повідомлень.

Для відстеження обробки повідомлень потрібно завести декілька рівнів логування:

- `debug` — логується інформація про надходження повідомлень та про помилки валідації повідомлень;
- `error` — логується інформація про помилки валідації повідомлень;
- `no log` — інформація про обробку повідомлень не логується.

Оскільки в задачі даного плагіну входить збереження даних необхідно розробити архітектуру таблиць, що будуть містити інформацію про зміни в записах.

На рисунку 2.4 зображено схему бази даних споживача повідомлень.

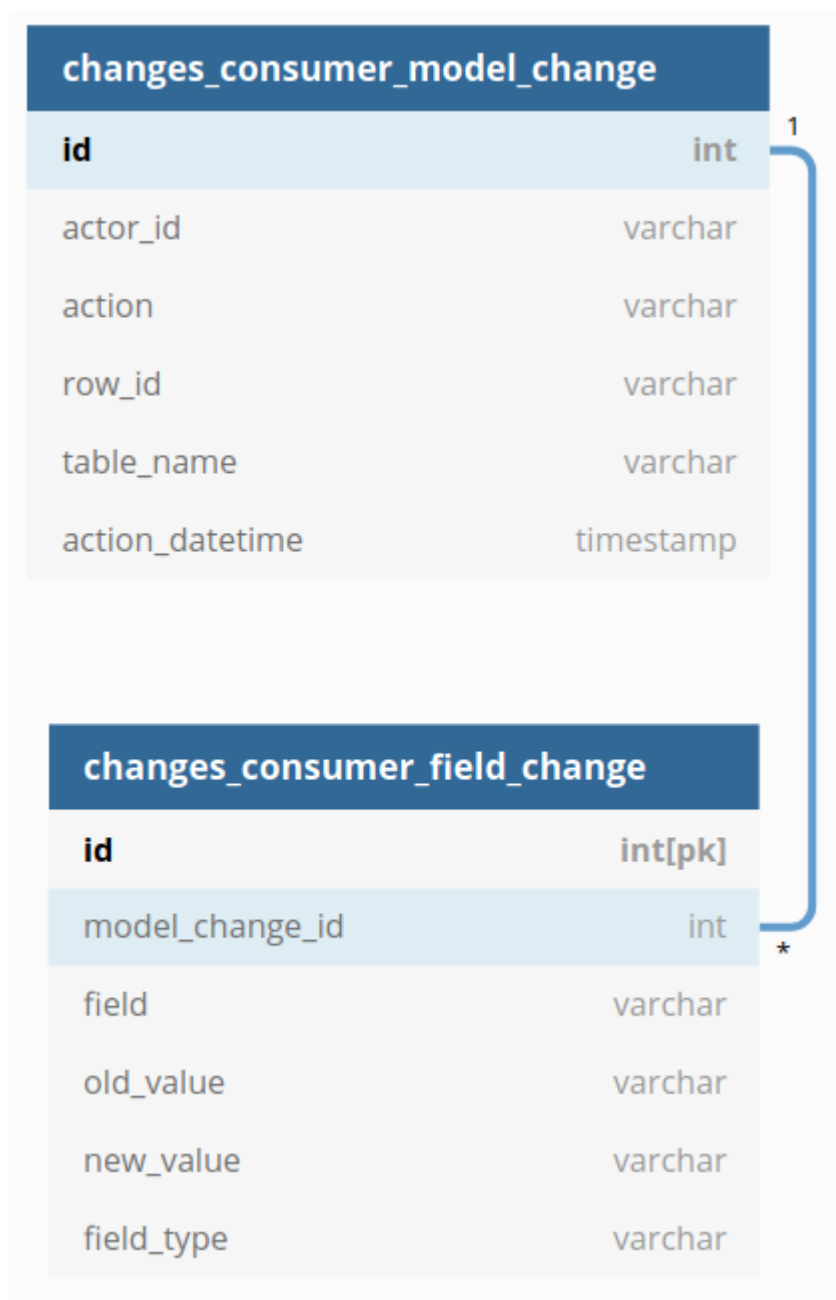


Рисунок 2.4 – Схема бази даних споживача повідомлень

Таблиця `changes_consumer_model_change` містить набір даних, який є спільним для подій будь-якого типу. Первинним ключем для цієї таблиці є поле `id`. Таблиця `changes_consumer_field_change` містить інформацію про зміни в полях записів. Має первинний ключ `id`, а також зовнішній ключ `model_change_id`, що посиляється на запис в таблиці `changes_consumer_model_change`, поле

changes_consumer_field_change.model_change_id відповідає полю
changes_consumer_model_change.id.

2.2.2.1 Опис модулів та класів

run_consumer — модуль, що слугує командою, яка запускає споживача повідомлень. Містить відповідний клас команди.

Command — клас, що розширює функціонал вбудованого класу Django BaseCommand. Слугує для запуску споживача повідомлень.

В таблиці 2.29 описано клас Command.

Таблиця 2.29 — Опис методів класу Command

Назва методу	Параметри	Опис
handle	Приймає набір параметрів, що передаються з командної строки	Запускає споживача повідомлень.
get_loggers		Повертає словник, ключами якого є константи, що відповідають рівню логуванню, а значеннями є об'єкти логерів відповідного рівня.

loggers — модуль, що містить класи логерів.

AbstractConsumerLogger — базовий клас для усіх логерів.

В таблиці 2.30 описано клас AbstractConsumerLogger.

Таблиця 2.30 — Опис методів класу AbstractConsumerLogger

Назва методу	Параметри	Опис
log_validation_error	error — об'єкт помилки, що створився в результаті помилки валідації повідомлення з брокеру повідомлень.	Абстрактний метод для виведення помилки валідації повідомлення з брокеру повідомлень.
log_message	message — об'єкт повідомлення, що вилучено з брокеру повідомлень.	Абстрактний метод для виведення повідомлення, що вилучено з брокеру повідомлень.

DebugLevelLogger — клас, що використовується для логування процесу обробки повідомлення при рівні логування debug.

В таблицях 2.31 — 2.32 описано клас DebugLevelLogger.

Таблиця 2.31 — Опис полів класу DebugLevelLogger

Назва поля	Опис
logger	Поле об'єкту, що містить вбудований Python-логер.

Таблиця 2.32 — Опис методів класу DebugLevelLogger

Назва методу	Параметри	Опис
log_validation_error	error — об'єкт помилки, що створився в результаті помилки валідації повідомлення з брокеру повідомлень.	Виводить повідомлення про помилку валідації повідомлення з брокеру повідомлень.

Продовження таблиці 2.32

log_message	message — об'єкт повідомлення, що вилучено з брокеру повідомлень.	Виводить інформацію про повідомлення, що було вилучено з брокеру повідомлень.
-------------	---	---

ErrorLevelLogger — клас, що використовується для логування процесу обробки повідомлення при рівні логування error.

В таблицях 2.33 — 2.34 описано клас ErrorLevelLogger.

Таблиця 2.33 — Опис полів класу ErrorLevelLogger

Назва поля	Опис
logger	Поле об'єкту, що містить вбудований Python-логер.

Таблиця 2.34 — Опис методів класу ErrorLevelLogger

Назва методу	Параметри	Опис
log_validation_error	error — об'єкт помилки, що створився в результаті помилки валідації повідомлення з брокеру повідомлень.	Виводить повідомлення про помилку валідації повідомлення з брокеру повідомлень.
log_message	message — об'єкт повідомлення, що вилучено з брокеру повідомлень.	Повертає порожнє значення, не виводить повідомлення.

NoLoggingLevelLogger — клас, що використовується для логування процесу обробки повідомлення при рівні логування no log.

В таблиці 2.35 описано клас NoLoggingLevelLogger.

					КП.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		79

Таблиця 2.35 — Опис методів класу NoLoggingLevelLogger

Назва методу	Параметри	Опис
log_validation_error	error — об'єкт помилки, що створився в результаті помилки валідації повідомлення з брокеру повідомлень.	Повертає порожнє значення, не виводить повідомлення.
log_message	message — об'єкт повідомлення, що вилучено з брокеру повідомлень.	Повертає порожнє значення, не виводить повідомлення.

В модулі consumer знаходиться клас споживача повідомлень.

Consumer — клас споживача повідомлень, що слугує для вилучення повідомлень з брокеру повідомлень.

В таблицях 2.36 — 2.37 описано клас Consumer.

Таблиця 2.36 — Опис полів класу Consumer

Назва поля	Опис
consumer	Поле об'єкту, що містить об'єкт класу KafkaConsumer.
logger	Поле об'єкту, що містить об'єкт логера.

Таблиця 2.37 — Опис методів класу Consumer

Назва методу	Параметри	Опис
run		Розпочинає процес вилучення та обробки повідомлень з брокеру повідомлень.

Модуль models містить класи моделей, що відображають сутності з бази даних.

ModelChange — клас, що є відображенням таблиці changes_consumer_model_change.

В таблиці 2.38 описано клас ModelChange.

Таблиця 2.38 — Опис полів класу ModelChange

Назва поля	Опис
actor_id	Ідентифікатор користувача, що вніс зміну.
action	Тип зміни, що відбулася.
row_id	Ідентифікатор запису в таблиці, де відбулася зміна.
table_name	Назва таблиці бази даних, де відбулася зміна.
action_datetime	Дата та час, коли відбулася зміна.

FieldChange — клас, що є відображенням таблиці changes_consumer_field_change.

В таблиці 2.39 описано клас FieldChange.

Таблиця 2.39 — Опис полів класу FieldChange

Назва поля	Опис
model_change	містить посилання на об'єкт класу ModelChange.
field	Назва поля, що змінилось.
old_value	Старе значення поля.
new_value	Нове значення поля.
field_type	Тип поля.

Модуль schemas містить класи, що відповідають за валідацію повідомлень, що приходять з брокеру повідомлень.

Клас ModelChangeMessageSchema — клас, що слугує для валідації повідомлення з брокеру повідомлень.

В таблиці 2.40 описано клас ModelChangeMessageSchema.

Таблиця 2.40 — Опис полів класу ModelChangeMessageSchema

Назва поля	Опис
actor_id	Поле об'єкту, яке вказує на перевірку того, що повідомлення містить ідентифікатор користувача, що вніс зміну.
action	Поле об'єкту, яке вказує на перевірку того, що повідомлення містить тип події, що відбулася.
row_id	Перевіряє, що повідомлення містить ідентифікатор запису, що змінився.
table_name	Поле об'єкту, яке вказує на перевірку того, що повідомлення містить назву таблиці, де відбулася зміна.

Продовження 2.40

action_datetime	Поле об'єкту, яке вказує на перевірку того, що повідомлення містить дату та час, коли відбулася зміна.
changes	Поле об'єкту, яке вказує на перевірку того, що повідомлення містить список з даними про зміну полів запису.

Клас `FieldChangeSchema` — клас, що є доповненням до класу `ModelChangeMessageSchema` для валідації вкладених повідомлень про зміни в полях запису.

В таблиці 2.41 описано клас `FieldChangeSchema`.

Таблиця 2.41 — Опис полів класу `FieldChangeSchema`

Назва поля	Опис
field	Поле об'єкту, яке вказує на перевірку того, що дані про зміну поля містять назву поля, що змінилось.
old_value	Поле об'єкту, яке вказує на перевірку того, що дані про зміну поля містять старе значення поля, що змінилось.
new_value	Поле об'єкту, яке вказує на перевірку того, що дані про зміну поля містять нове значення поля, що змінилось.
field_type	Поле об'єкту, яке вказує на перевірку того, що дані про зміну поля містять тип даних поля, що змінилось.

Модуль `message_saver` містить клас, що відповідає за збереження даних про зміну в базу даних.

MessageSaver — клас, що обробляє повідомлення та зберігає отримані дані в базу даних.

В таблицях 2.42 — 2.43 описано клас MessageSaver.

Таблиця 2.42 — Опис полів класу

Назва поля	Опис
Message	Повідомлення, що містить дані для збереження.

Таблиця 2.43 — Опис методів класу

Назва методу	Параметри	Опис
save		Зберігає повідомлення у базі даних.

2.3 Висновки по розділу

В даному розділі було описано архітектуру системи відстеження змін, а також системи, що її використовує. Було описано яким чином частини системи взаємодіють один з одним. Було описано різні проблеми, які постають при розробці системи відстеження змін, а саме її складових — Django-плагіну виробника повідомлень та Django-плагіну споживача повідомлень. Було наведено можливі варіанти реалізації функціоналу, виокремлено переваги та недоліки різних підходів, а також надано пояснення, чому було обрано певний підхід до розв'язання задач. Було описано модулі та класи Django-плагіну споживача повідомлень та Django-плагіну виробника повідомлень.

3 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Аналіз програмного забезпечення

В процесі тестування програмного забезпечення була перевірена функціональність системи відповідно до варіантів використання. В таблицях 3.1 – 3.15 описано результати тестування.

Таблиця 3.1 – Перевірка можливості реєстрації користувача

Мета тесту	Перевірка можливості реєстрації користувача.
Початковий стан	Відкрита сторінка реєстрації користувача.
Вхідні дані	Логін та пароль користувача.
Схема проведення тесту	Ввести логін та пароль в відповідні поля вводу та натиснути на кнопку підтвердження.
Очікуваний результат	В системі створено нового користувача. Користувача перенаправляють на сторінку авторизації звичайного користувача.
Стан програмного продукту після проведення випробувань	В системі створено нового користувача. Користувача перенаправляють на сторінку авторизації звичайного користувача.

Таблиця 3.2 – Перевірка можливості входу у систему звичайного користувача

Мета тесту	Перевірка можливості входу у систему звичайного користувача.
Початковий стан	Неавторизований користувач знаходиться на сторінці авторизації звичайного користувача.
Вхідні дані	Логін та пароль користувача.
Схема проведення тесту	Ввести логін та пароль в відповідні поля вводу та натиснути на кнопку підтвердження.
Очікуваний результат	Користувач є авторизованим в системі як звичайний користувач, знаходиться на сторінці зі списком усіх товарів.
Стан програмного продукту після проведення випробувань	Користувач є авторизованим в системі як звичайний користувач, знаходиться на сторінці зі списком усіх товарів.

Таблиця 3.3 – Перевірка можливості входу у систему адміністратора

Мета тесту	Перевірка можливості входу у систему адміністратора.
Початковий стан	Неавторизований користувач знаходиться на сторінці авторизації звичайного адміністратора.
Вхідні дані	Логін та пароль користувача.

Продовження таблиці 3.3

Схема проведення тесту	Ввести логін та пароль в відповідні поля вводу та натиснути на кнопку підтвердження.
Очікуваний результат	Користувач є авторизованим в системі як адміністратор, знаходиться на сторінці зі списком усіх товарів.
Стан програмного продукту після проведення випробувань	Користувач є авторизованим в системі як адміністратор, знаходиться на сторінці зі списком усіх товарів.

Таблиця 3.4 – Перевірка можливості перегляду списку товарів

Мета тесту	Перевірка можливості перегляду списку товарів.
Початковий стан	Користувач є авторизованим в системі як звичайний користувач
Вхідні дані	
Схема проведення тесту	Перейти на головну сторінку вебсайту
Очікуваний результат	Користувач знаходиться на головній сторінці сайту зі списком всіх товарів.
Стан програмного продукту після проведення випробувань	Користувач знаходиться на головній сторінці сайту зі списком всіх товарів.

Таблиця 3.5 – Перевірка можливості видалення товару

Мета тесту	Перевірка можливості видалення товару.
Початковий стан	Користувач є авторизованим як звичайний користувач. Користувач знаходиться на сторінці зі списком всіх товарів. В списку товарів присутні товари, що належать користувачу.
Вхідні дані	
Схема проведення тесту	Користувач натискає на кнопку видалення продукту.
Очікуваний результат	Користувач знаходиться на сторінці зі списком продуктів. Список товарів не містить товару, що був видалений.
Стан програмного продукту після проведення випробувань	Користувач знаходиться на сторінці зі списком продуктів. Список товарів не містить товару, що був видалений.

Таблиця 3.6 – Перевірка можливості детального перегляду товару

Мета тесту	Перевірка можливості детального перегляду товару.
Початковий стан	Користувач є авторизованим як звичайний користувач. Користувач знаходиться на сторінці зі списком всіх товарів. В списку товарів присутні товари.
Вхідні дані	

Продовження таблиці 3.6

Схема проведення тесту	Користувач натискає на кнопку детального перегляду товару.
Очікуваний результат	Користувач знаходиться на сторінці детального перегляду товару. Сторінка містить інформацію про ціну, опис, логін власника, назву товару.
Стан програмного продукту після проведення випробувань	Користувач знаходиться на сторінці детального перегляду товару. Сторінка містить інформацію про ціну, опис, логін власника, назву товару.

Таблиця 3.7 – Перевірка можливості редагування товару

Мета тесту	Перевірка можливості редагування товару
Початковий стан	Користувач знаходиться на сторінці детального перегляду товару. Користувач є власником товару.
Вхідні дані	Назва, опис, ціна товару
Схема проведення тесту	Користувач натискає на кнопку редагування товару. Користувач опиняється на сторінці редагування товару. Користувач оновлює відповідні поля з назвою, описом, ціною товару. Користувач натискає кнопку збереження.

Продовження таблиці 3.7

Очікуваний результат	Інформація про товар оновлена. Користувач знаходиться на сторінці з детальним описом товару.
Стан програмного продукту після проведення випробувань	Інформація про товар оновлена. Користувач знаходиться на сторінці з детальним описом товару.

Таблиця 3.8 – Перевірка можливості створення товару

Мета тесту	Перевірка можливості створення товару.
Початковий стан	Користувач знаходиться на сторінці зі списком усіх товарів.
Вхідні дані	Назва, ціна, опис товару
Схема проведення тесту	Користувач натискає на кнопку створення товару, користувач опиняється на сторінці створення товару. Користувач вводить дані у відповідні поля про назву, опис, ціну товару. Користувач натискає кнопку створення.
Очікуваний результат	Товар додано до системи. Користувач знаходиться на сторінці зі списком усіх продуктів. Новий продукт присутній у списку.
Стан програмного продукту після проведення випробувань	Товар додано до системи. Користувач знаходиться на сторінці зі списком усіх продуктів. Новий продукт присутній у списку.

Таблиця 3.9 – Перевірка можливості перегляду змін в системі

Мета тесту	Перевірка можливості перегляду змін в системі.
Початковий стан	Користувач авторизований в системі як адміністратор.
Вхідні дані	
Схема проведення тесту	Користувач переходить на сторінку зі списком змін.
Очікуваний результат	Користувач знаходиться на сторінці зі списком змін. Кожен елемент списку містить інформацію про тип зміни, користувача, що вніс зміну, дату та час, коли внесено зміну, назву таблиці, де відбулася зміна, ідентифікатор запису, де відбулася зміна. Для подій оновлення міститься інформація про назву, тип, старе значення, нове значення полів.
Стан програмного продукту після проведення випробувань	Користувач знаходиться на сторінці зі списком змін. Кожен елемент списку містить інформацію про тип зміни, користувача, що вніс зміну, дату та час, коли внесено зміну, назву таблиці, де відбулася зміна, ідентифікатор запису, де відбулася зміна. Для подій оновлення міститься інформація про назву, тип, старе значення, нове значення полів.

Таблиця 3.10 – Перевірка можливості фільтрації змін в системі

Мета тесту	Перевірка можливості фільтрації змін в системі.
Початковий стан	Користувач авторизований в системі, як адміністратор. Користувач знаходиться на сторінці зі списком змін.
Вхідні дані	Початкова дата, коли внесено зміну, кінцева дата, коли внесено зміну, логін користувача, що вніс зміну.
Схема проведення тесту	Користувач натискає на кнопку фільтрації змін, вводить вхідні дані у відповідні поля натискає на кнопку застосування фільтрації.
Очікуваний результат	Користувач знаходиться на сторінці зі списком змін. Список змін відфільтрований відповідно до вхідних даних.
Стан програмного продукту після проведення випробувань	Користувач знаходиться на сторінці зі списком змін. Список змін відфільтрований відповідно до вхідних даних.

Таблиця 3.11 – Перевірка можливості підключення моделі до системи відстеження змін

Мета тесту	Перевірка можливості підключення моделі до системи відстеження змін.
Початковий стан	Розробник системи
Вхідні дані	Модель, яку необхідно підключити до системи, список полів, що необхідно відстежити, список змін, що необхідно відстежити.
Схема проведення тесту	Розробник підключає модель Product до системи відстеження змін для подій створення, видалення, редагування, для усіх полів.
Очікуваний результат	Модель підключено до системи відстеження змін з відповідними налаштуваннями.
Стан програмного продукту після проведення випробувань	Модель підключено до системи відстеження змін з відповідними налаштуваннями.

Таблиця 3.12 – Перевірка можливості відправлення повідомлення про створення запису

Мета тесту	Перевірка можливості відправлення повідомлення про створення запису.
Початковий стан	Модель Product підключено до системи відстеження змін. Користувач є авторизованим в системі, як звичайний користувач. Користувач знаходиться на сторінці зі списком усіх продуктів.
Вхідні дані	Назва, ціна, опис товару.
Схема проведення тесту	Користувач натискає на кнопку створення товару. Користувач опиняється на сторінці створення товару. Користувач вводить вхідні дані у відповідні поля. Користувач натискає на кнопку створення товару.
Очікуваний результат	Товар додано до системи. Повідомлення про створення товару відправлено в брокер повідомлень.
Стан програмного продукту після проведення випробувань	Товар додано до системи. Повідомлення про створення товару відправлено в брокер повідомлень.

Таблиця 3.13 – Перевірка можливості відправлення повідомлення про редагування запису

Мета тесту	Перевірка можливості відправлення повідомлення про редагування запису.
Початковий стан	Модель Product підключено до системи відстеження змін. Користувач є авторизованим в системі, як звичайний користувач. Користувач знаходиться на сторінці з детальним переглядом товару. Користувач є власником товару.
Вхідні дані	Назва, ціна, опис товару.
Схема проведення тесту	Користувач натискає на кнопку редагування товару. Користувач опиняється на сторінці редагування товару. Користувач вводить вхідні дані у відповідні поля. Користувач натискає на кнопку редагування товару.
Очікуваний результат	Товар оновлено в системі. Повідомлення про редагування товару відправлено в брокер повідомлень.
Стан програмного продукту після проведення випробувань	Товар оновлено в системі. Повідомлення про редагування товару відправлено в брокер повідомлень.

Таблиця 3.14 – Перевірка можливості відправлення повідомлення про видалення запису

Мета тесту	Перевірка можливості відправлення повідомлення про видалення запису.
Початковий стан	Модель Product підключено до системи відстеження змін. Користувач є авторизованим в системі, як звичайний користувач. Користувач знаходиться на сторінці зі списком усіх продуктів. В списку містяться товари, що належать користувачу.
Вхідні дані	
Схема проведення тесту	Користувач натискає на кнопку видалення товару.
Очікуваний результат	Товар видалено з системи. Повідомлення про видалення товару відправлено в брокер повідомлень.
Стан програмного продукту після проведення випробувань	Товар видалено з системи. Повідомлення про видалення товару відправлено в брокер повідомлень.

Таблиця 3.15 – Перевірка можливості збереження повідомлення

Мета тесту	Перевірка можливості збереження повідомлення.
Початковий стан	Споживач повідомлень підключено до брокеру повідомлень.
Вхідні дані	Повідомлення про зміну.
Схема проведення тесту	Повідомлення про зміну відправляється в брокер повідомлень.
Очікуваний результат	Повідомлення зчитується споживачем повідомлень з брокеру повідомлень. Повідомлення зберігається у базі даних.
Стан програмного продукту після проведення випробувань	Повідомлення зчитується споживачем повідомлень з брокеру повідомлень. Повідомлення зберігається у базі даних.

3.2 Висновки до розділу

Було проведено тестування функціоналу системи та надано результати перевірки.

4 ВПРОВАДЖЕННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Розгортання програмного забезпечення

Перед безпосереднім розгортанням необхідно встановити наступне додаткове програмне забезпечення:

- Docker;
- Docker-compose;
- Python3.7;
- Node.js;
- Nginx.

Після встановлення необхідних компонентів потрібно окремо запустити компоненти системи. Для цього необхідно виконати наступні кроки:

- запустити сервіс nginx використовуючи відповідний скрипт setup.sh;
- запустити admin-service використовуючи відповідний скрипт start.sh;
- запустити authentication-service використовуючи відповідний скрипт start.sh;
- запустити products-service використовуючи відповідний скрипт start.sh;
- запустити consumer використовуючи відповідний скрипт run_consumer.sh;
- запустити kafka використовуючи docker-compose;
- запустити клієнтський додаток використовуючи прт.

4.2 Робота з програмним забезпеченням

Роботу з програмним забезпеченням описано в відповідних документах “Керівництво програміста” та “Керівництво користувача”.

					КПІ.ІП-7205.045490.02.81	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		98

4.3 Висновки до розділу

Було описано вимоги до розгортання програмного забезпечення та описано процес розгортання програмного забезпечення.

					КПІ.ІП-7205.045490.02.81	Арк.
						99
Змн.	Арк.	№ докум.	Підпис	Дата		

ВИСНОВКИ

В результаті виконання дипломного проєкту було розроблено бібліотеку для відстеження змін в Django-застосуваннях. Для перевірки функціонування бібліотеки було створено прототип вебсайту для розміщення товарів. Було проведено тестування роботи системи.

У першому розділі було надано опис предметної області, проведено аналіз наявних технічних рішень, виокремлено переваги рішення, що використовується у даній роботі, у порівнянні з наявними аналогами. Було описано функціональні та не функціональні вимоги до програмного забезпечення, створено структурні схеми варіантів використання.

У другому розділі було описано архітектуру системи відстеження змін, описано структуру плагіну споживача повідомлень та структуру плагіну виробника повідомлень. Було проведено аналіз можливих шляхів розв'язання поставлених задач. Було описано класи та модулі споживача повідомлень та виробника повідомлень.

У третьому розділі було детально описано результати тестування програмного забезпечення.

У четвертому розділі було описано процес розгортання програмного забезпечення.

ПЕРЕЛІК ПОСИЛАНЬ

- 1) Multi threaded design guidelines for libraries: Part I - Ayende @ Rahien [Електронний ресурс] – Режим доступу до ресурсу: https://ayende.com/blog/159201/multi-threaded-design-guidelines-for-libraries-part-i?key=05ccc8b6-9a75-48a6-bc9d-dd3b9fbee4a&utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+AyendeRahien+%28Ayende+%40+Rahien%29&utm_content=Google+Reader.
- 2) Система управління базами даних — Вікіпедія [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B0%D0%BC%D0%B8_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85.
- 3) Writing your first Django app, part 1 | Django documentation | Django [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.djangoproject.com/en/3.1/intro/tutorial01/>.
- 4) Welcome to django-dirtyfields's documentation! – django-dirtyfields 0.8.1 documentation [Електронний ресурс] – Режим доступу до ресурсу: <https://django-dirtyfields.readthedocs.io/en/develop/>.
- 5) Design Patterns : Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, J. Ralph, V. John. – Harlow: Pearson Education Limited, 1997. – 416 с.
- 6) Apache Kafka – мой конспект / Хабр [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: <https://habr.com/ru/post/354486/>.
- 7) 4. Kafka Consumers: Reading Data from Kafka - Kafka: The Definitive Guide [Book] [Електронний ресурс] – Режим доступу до ресурсу:

<https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html>.

8) Tracking changes in PostgreSQL - Cybertec [Електронний ресурс] – Режим доступу до ресурсу: <https://www.cybertec-postgresql.com/en/tracking-changes-in-postgresql/>.

9) Quick Start – django-simple-history 3.0.0.post6+g666b6a6 documentation [Електронний ресурс] – Режим доступу до ресурсу: https://django-simple-history.readthedocs.io/en/latest/quick_start.html.

10) Melé A. Django 2 by Example: Build powerful and reliable Python web applications from scratch / Antonio Melé., 2018. – 526 с. – (2nd Revised edition).

ДОДАТОК А ЗВІТ З ПЕРЕВІРКИ НА ПЛАГІАТ

Факультет інформатики та обчислювальної техніки
Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ Олександр ПАВЛОВ

“ ____ ” _____ 2021 р.

БІБЛІОТЕКА ДЛЯ ЛОГУВАННЯ ЗМІН В МОДЕЛЯХ DJANGO-

ЗАСТОСУВАНЬ

Технічне завдання

КПШ.ІІІ-7205.045490.03.91

“ПОГОДЖЕНО”

Керівник проєкту:

_____ К.І. Ліщук

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ К.В. Вакуленко

Київ – 2021 року

ЗМІСТ

1	НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ.....	3
2	ПІДСТАВА ДЛЯ РОЗРОБКИ.....	4
3	ПРИЗНАЧЕННЯ РОЗРОБКИ	5
4	ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	6
4.1	ВИМОГИ ДО ФУНКЦІОНАЛЬНИХ ХАРАКТЕРИСТИК.....	6
4.2	ВИМОГИ ДО НАДІЙНОСТІ.....	7
4.3	УМОВИ ЕКСПЛУАТАЦІЇ	7
4.4	ВИМОГИ ДО СКЛАДУ І ПАРАМЕТРІВ ТЕХНІЧНИХ ЗАСОБІВ	7
4.5	ВИМОГИ ДО ІНФОРМАЦІЙНОЇ ТА ПРОГРАМНОЇ СУМІСНОСТІ	7
4.6	ВИМОГИ ДО МАРКУВАННЯ ТА ПАКУВАННЯ.....	7
4.7	ВИМОГИ ДО ТРАНСПОРТУВАННЯ ТА ЗБЕРІГАННЯ.....	7
4.8	СПЕЦІАЛЬНІ ВИМОГИ	7
5	ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ	8
6	СТАДІЇ І ЕТАПИ РОЗРОБКИ.....	9
7	ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ	10

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Бібліотека для логування змін в моделях Django-застосувачів.

Галузь застосування:

Наведене технічне завдання поширюється на розробку бібліотеки для логування змін в моделях Django-застосувачів КП.ІП-7205.045490, котра використовується для відстеження та збереження змін, що відбулися в базі даних, в результаті взаємодії користувачів із системою та призначена для використання платформами, що розроблені за допомогою фреймворку Django.

					КП.ІП-7205.045490.03.91	Арк.
						3
Змн.	Арк.	№ докум.	Підпис	Дата		

2 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки бібліотеки для логування змін в моделях Django-застосувань є завдання на дипломне проектування, затверджене кафедрою автоматизованих систем обробки інформації і управління Національного технічного університету України «Київський політехнічний інститут» ім.Ігоря Сікорського (КПІ ім. Ігоря Сікорського).

					КПІ.ІП-7205.045490.03.91	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		4

3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для розробників, перед якими стоїть завдання відстежувати зміни в базах даних у мікросервісних системах, що розробляються з використанням фреймворку Django.

Метою розробки є надання можливості легування змін баз даних в Django системах, та централізованого збереження отриманих даних.

					КПІ.ІП-7205.045490.03.91	Арк.
						5
Змн.	Арк.	№ докум.	Підпис	Дата		

4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вимоги до функціональних характеристик

4.1.1 Програмне забезпечення повинно забезпечувати виконання наступних основних функцій:

4.1.1.1 Для користувача:

- реєстрація в прототипі системи для додавання товарів, що використовує функціонал бібліотеки;
- авторизація в системі;
- створення нових товарів;
- редагування власних товарів;
- видалення власних товарів;
- перегляд усіх товарів;
- детальний перегляд товару.

4.1.1.2 Для адміністратора системи:

- авторизація в системі адміністратора;
- перегляд змін, що внесли звичайні користувачі;
- фільтрація змін за логіном та датами коли відбулися зміни.

4.1.2 Розробку виконати на платформі Ubuntu

4.1.3 Додаткові вимоги:

- мова користувацького інтерфейсу прототипу системи – англійська;
- клієнтська частина має бути розроблена з використанням Vue.js.

4.2 Вимоги до надійності

4.2.1 Передбачити контроль введення інформації.

4.2.2 Передбачити захист від некоректних дій користувача.

4.2.3 Забезпечити цілісність інформації в базі даних.

4.3 Умови експлуатації

4.3.1 Умови експлуатації згідно СанПін 2.2.2.542 – 96.

4.3.2 Обслуговування

4.4 Вимоги до складу і параметрів технічних засобів

4.4.1.1 Програмне забезпечення повинно функціонувати на Unix-сумісних персональних комп'ютерах.

4.5 Вимоги до інформаційної та програмної сумісності

4.5.1 Програмне забезпечення повинно працювати під управлінням операційних систем сімейства Unix.

4.6 Вимоги до маркування та пакування

Вимоги до маркування та пакування не пред'являються.

4.7 Вимоги до транспортування та зберігання

Вимоги до транспортування та зберігання не пред'являються.

4.8 Спеціальні вимоги

Спеціальні вимоги не пред'являються.

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

5.1 Програмні модулі, котрі розробляються, повинні бути задокументовані, тобто тексти програм повинні містити всі необхідні коментарі.

5.2 Програмне забезпечення повинно мати довідникову систему.

5.3 У склад супроводжувальної документації повинні входити наступні документи:

5.3.1 Пояснювальна записка не менше ніж на 60 аркушах формату А4 (без додатків 5.3.2 - 5.3.6).

5.3.2 Технічне завдання.

5.3.3 Керівництво користувача.

5.3.4 Керівництво програміста.

5.3.5 Програма та методика тестування

5.4 Графічна частина повинна бути виконана на 3 аркушах формату А3, котрі включаються у якості додатків до пояснювальної записки.

5.4.1 Схема структурна варіантів використання.

5.4.2 Схема структурна класів програмного забезпечення.

5.4.3 Креслення вигляду екранних форм.

6 СТАДІЇ І ЕТАПИ РОЗРОБКИ

Таблиця 6.1 – Стадії та етапи розробки

№	Назва етапу	Строк	Звітність
1.	Вивчення рекомендованої літератури	21.03.2021	
2.	Аналіз існуючих методів розв'язання задачі	27.03.2021	Аналіз наявних аналогів
3.	Постановка та формалізація задачі	05.04.2021	Комплексу завдань
4.	Аналіз вимог до програмного забезпечення	12.04.2021	Функціональні вимоги
5.	Алгоритмізація задачі	17.04.2021	Діаграма послідовності
6.	Моделювання програмного забезпечення	21.04.2021	Діаграма класів
7.	Обґрунтування використовуваних технічних засобів	23.04.2021	Письмовий звіт
8.	Розробка архітектури програмного забезпечення	29.04.2021	Схема архітектури
9.	Розробка програмного забезпечення	05.05.2021	Текст програмного забезпечення
10.	Налагодження програми	07.05.2021	Результати тестування
11.	Виконання графічних документів	12.05.2021	Графічні матеріали
12.	Оформлення пояснювальної записки	15.05.2021	Пояснювальна записка
13.	Подання ДП на попередній захист	17.05.2021	
14.	Подання ДП рецензенту	01.06.2021	
15.	Подання ДП на основний захист	08.06.2021	

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

7.1 Види випробувань

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

					КПІ.ІП-7205.045490.03.91	Арк.
						10
Змн.	Арк.	№ докум.	Підпис	Дата		

Факультет інформатики та обчислювальної техніки
Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ Олександр ПАВЛОВ

“ ____ ” _____ 2021 р.

**БІБЛІОТЕКА ДЛЯ ЛОГУВАННЯ ЗМІН В МОДЕЛЯХ DJANGO-
ЗАСТОСУВАНЬ**

Програма та методика тестування

КПШ.ІП-7205.045490.04.51

“ПОГОДЖЕНО”

Керівник проєкту:

_____ К.І. Ліщук

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ К.В. Вакуленко

Київ – 2021 року

ЗМІСТ

1	ОБ'ЄКТ ВИПРОБУВАНЬ.....	3
2	МЕТА ТЕСТУВАННЯ.....	4
3	МЕТОДИ ТЕСТУВАННЯ.....	5
4	ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ	6

					КПІ.ІП-7205.045490.04.51	Арк.
						2
Змн.	Арк.	№ докум.	Підпис	Дата		

1 ОБ'ЄКТ ВИПРОБУВАНЬ

Об'єктами випробувань є прототип вебсайту для розміщення товарів та підключена до нього система відстеження змін.

					КПІ.ІП-7205.045490.04.51	Арк.
						3
Змн.	Арк.	№ докум.	Підпис	Дата		

2 МЕТА ТЕСТУВАННЯ

Тестування проводиться для перевірки коректності роботи програмного забезпечення. В процесі тестування необхідно перевірити наступний функціонал:

- реєстрація користувача;
- вхід у систему звичайного користувача;
- вхід у систему адміністратора;
- перегляд списку товарів;
- видалення товару;
- детальний перегляд товару;
- редагування товару;
- створення товару;
- перегляд списку зі змінами;
- збереження інформації про створення товару;
- збереження інформації про видалення товару;
- збереження інформації про редагування товару;
- збереження інформації про поля, що були змінені;
- фільтрація списку зі змінами.

					КПІ.ІП-7205.045490.04.51	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		4

3 МЕТОДИ ТЕСТУВАННЯ

Тестування виконується методом “білої скриньки” шляхом мануальної перевірки роботи системи.

В процесі тести наступних типів:

- функціональні для перевірки коректності роботи API серверної частини;
- UI-тестування для перевірки коректності роботи інтерфейсу користувача.

					КПІ.ІП-7205.045490.04.51	Арк.
						5
Змн.	Арк.	№ докум.	Підпис	Дата		

4 ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Першою чергою проводяться функціональні тести бекенд частини застосунку, використовуючи програму Postman. Перевірка виконується наступним чином:

- в точки входу системи подаються очікувані значення;
- в точки входу системи подаються невірні поля та значення.

Потім проводиться UI-тестування шляхом перевірки відповідності функціонування системи функціональним вимогам.

					КПІ.ІП-7205.045490.04.51	Арк.
						6
Змн.	Арк.	№ докум.	Підпис	Дата		

Факультет інформатики та обчислювальної техніки
Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ Олександр ПАВЛОВ

“ ____ ” _____ 2021 р.

БІБЛІОТЕКА ДЛЯ ЛОГУВАННЯ ЗМІН В МОДЕЛЯХ DJANGO-

ЗАСТОСУВАНЬ

Керівництво користувача

КПІ.ІП-7205.045490.05.34

“ПОГОДЖЕНО”

Керівник проєкту:

_____ К.І. Ліщук

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ К.В. Вакуленко

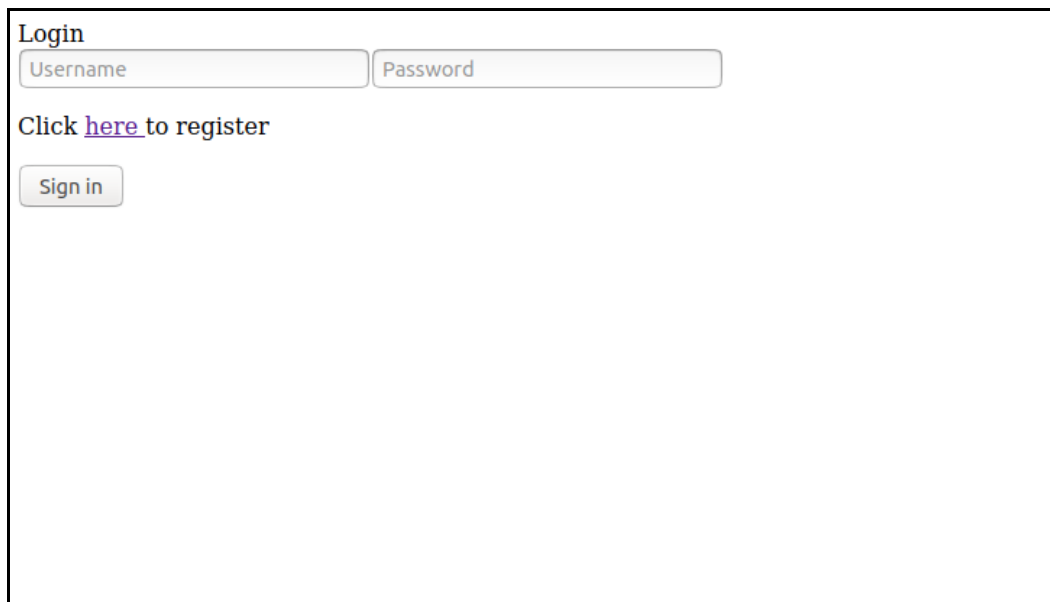
Київ – 2021 року

ЗМІСТ

1	КЕРІВНИЦТВО ЗВИЧАЙНОГО КОРИСТУВАЧА	3
2	КЕРІВНИЦТВО АДМІНІСТРАТОРА.....	11

1 КЕРІВНИЦТВО ЗВИЧАЙНОГО КОРИСТУВАЧА

Користувач переходить на головну сторінку сайту використовуючи браузер. На рисунку 1.1 зображено стартову сторінку.



Login

Username Password

Click [here](#) to register

Sign in

Рисунок 1.1 – Сторінка авторизації користувача

Для того, щоб зареєструватися у системі користувачу необхідно перейти на сторінку реєстрації за посиланням, що знаходиться на сторінці авторизації. Після цього необхідно ввести логін та пароль у відповідні поля та натиснути кнопку “Register”. На рисунку 1.2 зображено сторінку реєстрації користувача.

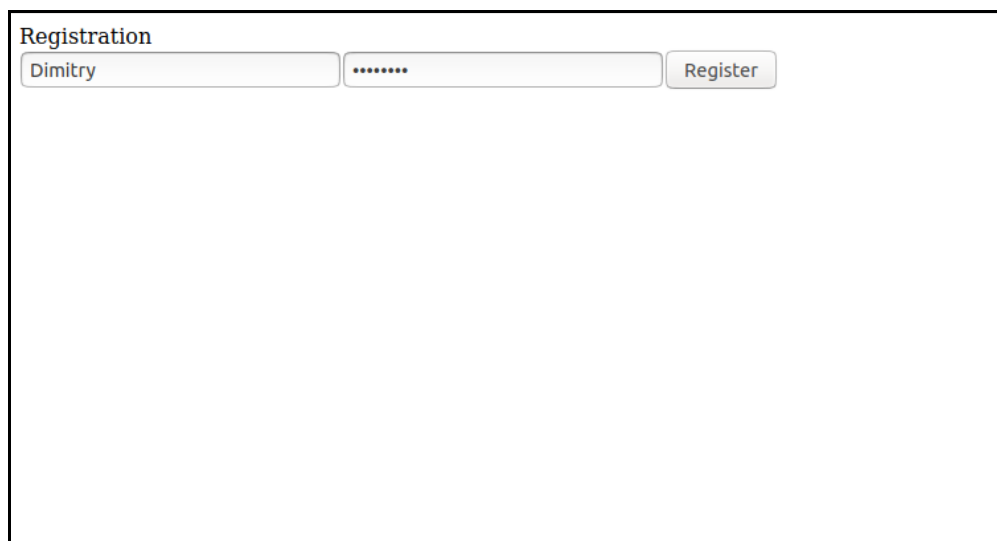
A screenshot of a web registration form. At the top, the word "Registration" is displayed. Below it, there are two input fields: the first contains the text "Dimitry" and the second contains a series of dots representing a password. To the right of these fields is a button labeled "Register".

Рисунок 1.2 – Сторінка реєстрації користувача

Після успішної реєстрації користувача опиняється на сторінці авторизації. Користувачу необхідно ввести логін та пароль у відповідні поля та натиснути кнопку “Sign in”. На рисунку 1.3 зображено сторінку авторизації користувача.

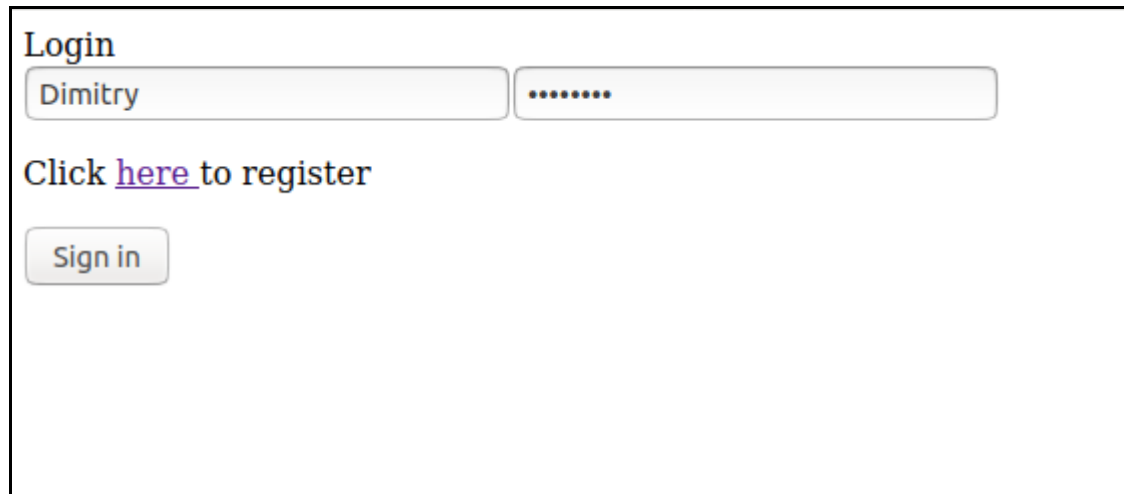
A screenshot of a web login form. At the top, the word "Login" is displayed. Below it, there are two input fields: the first contains the text "Dimitry" and the second contains a series of dots representing a password. Below these fields is a link that says "Click [here](#) to register". At the bottom of the form is a button labeled "Sign in".

Рисунок 1.3 – Сторінка авторизації користувача

Після успішної авторизації користувач опиняється на сторінці зі списком усіх товарів. У списку міститься інформація про назву, ціну, логін власника для кожного товару. На рисунку 1.4 зображено сторінку зі списком усіх товарів.

add Product

Products:

Product Name: Modded
4 PP
Price: 1.09
Owner: User1
[detail](#)

Product Name: Modded
4 PP
Price: 1.11
Owner: User1
[detail](#)

Product Name: New
Product

Рисунок 1.4 – Сторінка зі списком усіх товарів

Для того, щоб додати товар до системи користувачу необхідно натиснути на кнопку “add Product”. Після того, як користувач натиснув на цю кнопку, він опиняється на сторінці створення товару. Користувач має ввести дані про назву, ціну та опис товару у відповідні поля та натиснути кнопку “Add Product”. На рисунку 1.5 зображено сторінку створення товару.



Product Name
Dimitiry`s product N1

Price
12.05

Description
Product description

Add Product

Рисунок 1.5 – Сторінка створення товару

Після того, як користувач натисне на кнопку “Add Product”, він опиняється на сторінці зі списком усіх товарів, де може побачити, що його товар було додано у кінець списку. На рисунку 1.6 зображено сторінку зі списком усіх товарів.

Price: 100.00 Owner: user1 detail	
Product Name: Andriy`s first Product Price: 13.11 Owner: Andriy detail	
Product Name: Dimitiry`s product N1 Price: 12.05 Owner: Dimitry detail Delete	

Рисунок 1.6 – Сторінка зі списком усіх товарів

Для того, щоб видалити товар зі списку, користувач має натиснути кнопку “Delete”. Функція видалення товару є доступною лише для користувача, який додав товар до системи.

Для того, щоб переглянути детальний опис товару, користувач має перейти за посиланням “detail” для певного товару. Після переходу за посиланням користувач опиняється на сторінці з детальним переглядом товару. На сторінці можна побачити інформацію про назву, ціну, опис, логін власника товару. На рисунку 1.7 зображено сторінку детального перегляду товару.

Product Name:

Dimitiry`s product N1

Price:

12.05

Owner:

Dimitry

Description:

Product description

Рисунок 1.7 – Сторінка детального перегляду товару

Для того, щоб відредагувати товар, користувач має натиснути на кнопку “Edit”. Функція редагування товару є доступною лише для користувача, який додав товар до системи. Після того, як користувач натиснув на кнопку, він опиняється на сторінці редагування товару. Користувач має ввести оновлені значення у відповідні поля, які він збирається редагувати. Для збереження змін необхідно натиснути на кнопку “Save changes”. На рисунку 1.8 зображено сторінку редагування товару.

Product Name
Dimitiry`s Updated product N1

Price
133.05

Description
Updated Product description

Save changes

Рисунок 1.8 – Сторінка редагування товару

Після того, як користувач натисне на кнопку “Save changes”, він опиняється на сторінці детального перегляду цього товару, де може побачити оновлені значення товару. На рисунку 1.9 зображено сторінку детального перегляду товару.

Product Name:

Dimitiry`s Updated product N1

Price:

133.05

Owner:

Dimitry

Description:

Updated Product description

Рисунок 1.9 – Сторінка детального перегляду товару

					КПІ.ІП-7205.045490.05.34	Арк.
						10
Змн.	Арк.	№ докум.	Підпис	Дата		

2 КЕРІВНИЦТВО АДМІНІСТРАТОРА

Для початку роботи із системою адміністратора користувач переходить за посиланням на сторінку авторизації адміністратора. Користувач має ввести логін та пароль у відповідні поля та натиснути кнопку “Sign in”. На рисунку 2.1 зображено сторінку авторизації адміністратора.

Рисунок 2.1 – Сторінка авторизації адміністратора

Після успішної авторизації користувач опиняється на сторінці зі списком усіх змін. Для кожної зі змін наявна наступна інформація:

- логін користувача, що вніс зміни;
- тип зміни, що відбулася;
- ідентифікатор запису в таблиці бази даних;
- назва таблиці, де відбулася зміна;
- дата та час, коли відбулася зміна.

Для подій редагування також відображається список зі змінами полів.

Для кожного з відредагованих полів наявна наступна інформація:

- назва поля;
- тип поля;
- старе значення поля;

					КПІ.ІП-7205.045490.05.34	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		11

- нове значення поля.

На рисунку 2.2 зображено список зі змінами.

Filter

Changes:

Actor: user1

Action: insert

Row Id: 13

Table Name: products_product

Action Datetime: 2021-05-16T12:03:41.169000Z

Actor: user1

Action: update

Row Id: 13

Table Name: products_product

Action Datetime: 2021-05-16T12:04:00.526000Z

Field: price

Old value: 344.00

Рисунок 2.2 – Сторінка зі списком змін

Для того, щоб відфільтрувати список зі змінами, користувач має натиснути на кнопку “Filter”. Після того, як користувач натисне на цю кнопку, з’явиться діалогове вікно, де користувач зможе обрати проміжок часу, за який він хоче побачити зміни, та логін користувача зміни якого він хоче побачити. Після цього користувач має натиснути на кнопку “Apply”, щоб застосувати фільтри. На рисунку 2.3 зображено діалогове вікно з параметрами фільтрації.

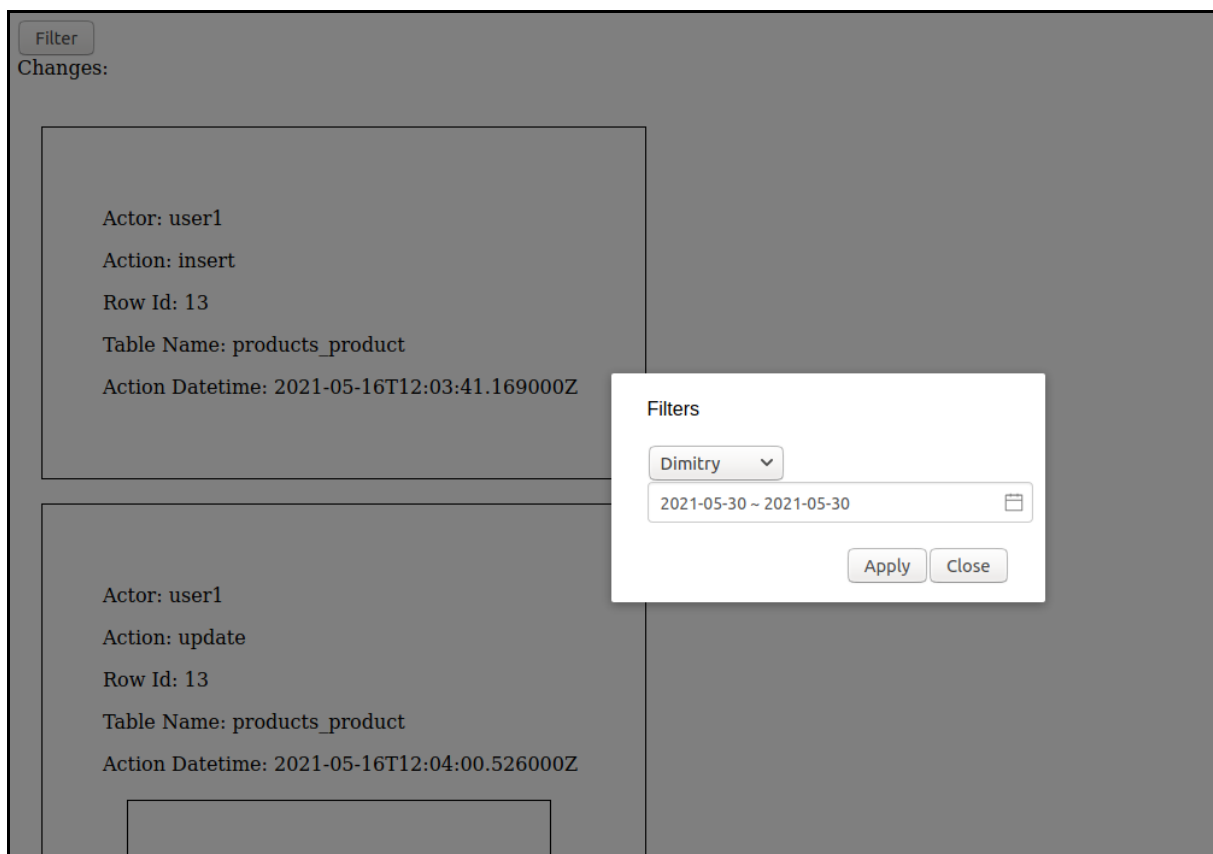


Рисунок 2.3 – Діалогове вікно з параметрами фільтрації

Після того, як користувач натисне на кнопку “Apply”, до списку застосуються фільтри, що були обрані на попередньому кроці. На рисунку 2.4 зображено список зі змінами.

Filter

Changes:

Actor: Dimitry
Action: insert
Row Id: 17
Table Name: products_product
Action Datetime: 2021-05-30T11:41:45.689000Z

Actor: Dimitry
Action: update
Row Id: 17
Table Name: products_product
Action Datetime: 2021-05-30T11:45:54.351000Z

Рисунок 2.4 – Сторінка зі списком змін

Факультет інформатики та обчислювальної техніки
Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ Олександр ПАВЛОВ

“ ____ ” _____ 2021 р.

БІБЛІОТЕКА ДЛЯ ЛОГУВАННЯ ЗМІН В МОДЕЛЯХ DJANGO-

ЗАСТОСУВАНЬ

Керівництво програміста

КПІ.ІП-7205.045490.06.33

“ПОГОДЖЕНО”

Керівник проекту:

_____ К.І. Ліщук

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ К.В. Вакуленко

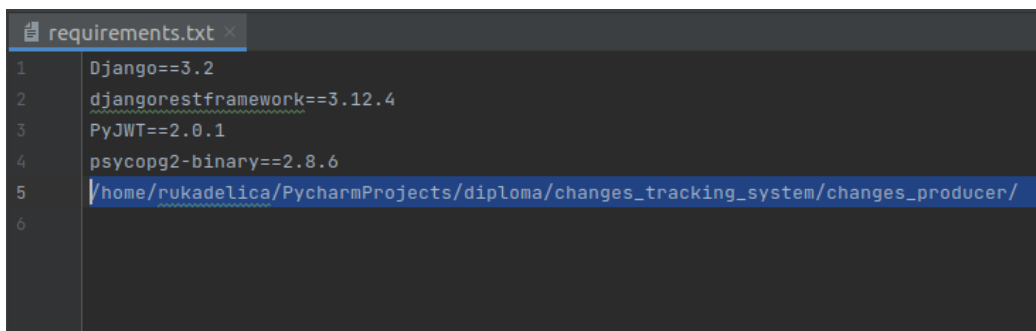
Київ – 2021 року

ЗМІСТ

1	ВИРОБНИК ПОВІДОМЛЕНЬ.....	3
2	СПОЖИВАЧ ПОВІДОМЛЕНЬ	6

1 ВИРОБНИК ПОВІДОМЛЕНЬ

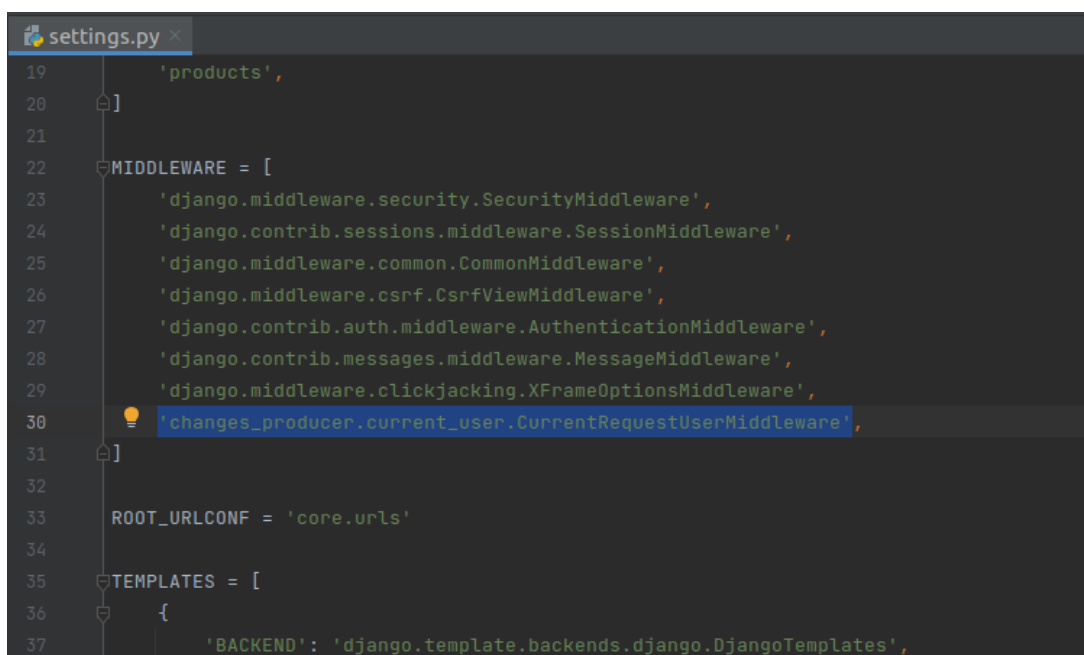
Для підключення сервісу до системи відстеження змін необхідно встановити плагін виробника повідомлень та додати його до списку залежностей, як показано на рисунку 1.1.



```
requirements.txt
1 Django==3.2
2 djangorestframework==3.12.4
3 PyJWT==2.0.1
4 psycopg2-binary==2.8.6
5 /home/rukadelica/PycharmProjects/diploma/changes_tracking_system/changes_producer/
6
```

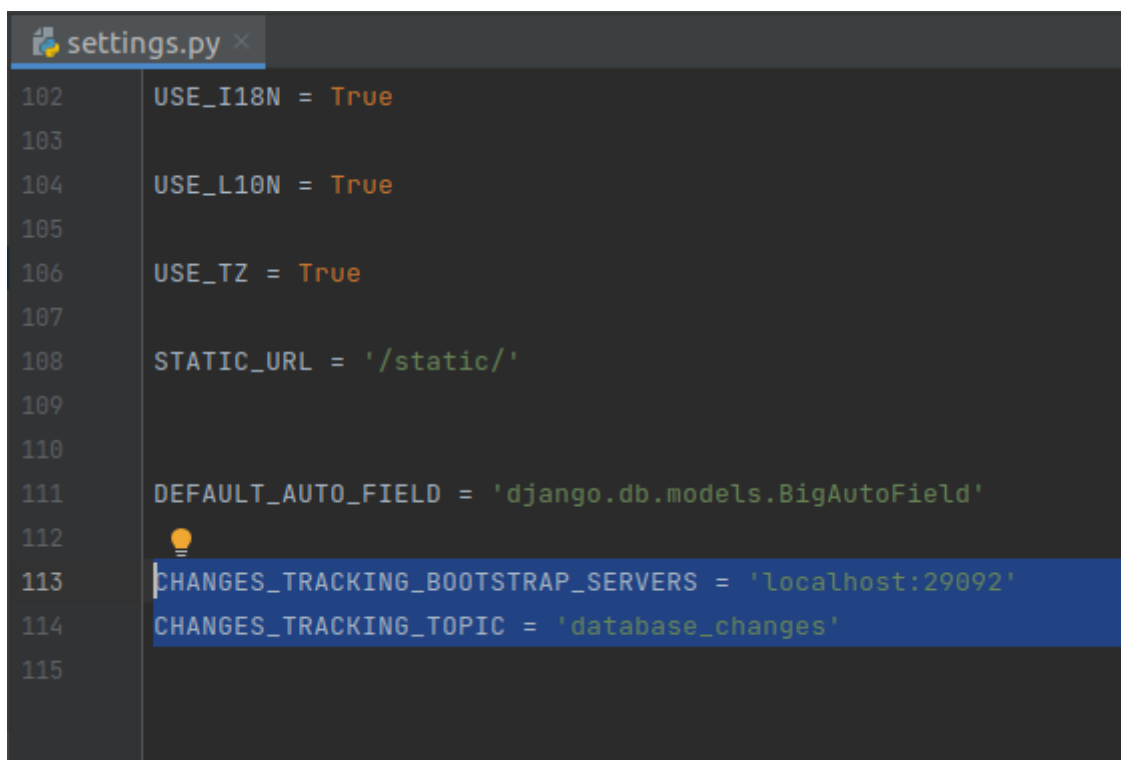
Рисунок 1.1 – Файл із залежностями Django-застосунку

Після встановлення плагіну необхідно додати `changes_producer.current_user.CurrentRequestUserMiddleware` до списку `MIDDLEWARE`, що знаходиться у файлі із налаштуваннями, як показано на рисунку 1.2, а також додати налаштування для `kafka`, як показано на рисунку 1.3.



```
settings.py
19 'products',
20 ]
21
22 MIDDLEWARE = [
23     'django.middleware.security.SecurityMiddleware',
24     'django.contrib.sessions.middleware.SessionMiddleware',
25     'django.middleware.common.CommonMiddleware',
26     'django.middleware.csrf.CsrfViewMiddleware',
27     'django.contrib.auth.middleware.AuthenticationMiddleware',
28     'django.contrib.messages.middleware.MessageMiddleware',
29     'django.middleware.clickjacking.XFrameOptionsMiddleware',
30     'changes_producer.current_user.CurrentRequestUserMiddleware',
31 ]
32
33 ROOT_URLCONF = 'core.urls'
34
35 TEMPLATES = [
36     {
37         'BACKEND': 'django.template.backends.django.DjangoTemplates',
```

Рисунок 1.2 – Файл із налаштуваннями Django-застосунку



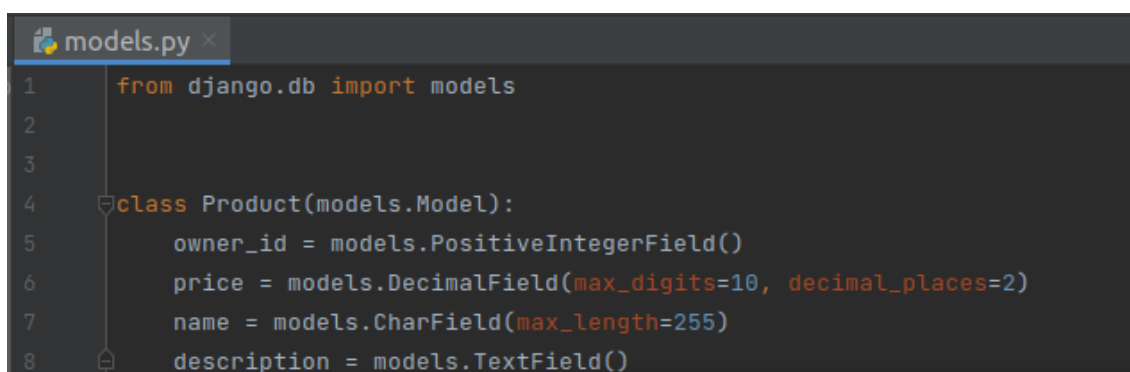
```

102 USE_I18N = True
103
104 USE_L10N = True
105
106 USE_TZ = True
107
108 STATIC_URL = '/static/'
109
110
111 DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
112
113 CHANGES_TRACKING_BOOTSTRAP_SERVERS = 'localhost:29092'
114 CHANGES_TRACKING_TOPIC = 'database_changes'
115

```

Рисунок 1.3 – Файл із налаштуваннями Django-застосунку

Після налаштувань можна переходити до безпосереднього підключення моделей до системи відстеження змін. Як приклад візьмемо модель, що зображена на рисунку 1.4.



```

1 from django.db import models
2
3
4 class Product(models.Model):
5     owner_id = models.PositiveIntegerField()
6     price = models.DecimalField(max_digits=10, decimal_places=2)
7     name = models.CharField(max_length=255)
8     description = models.TextField()

```

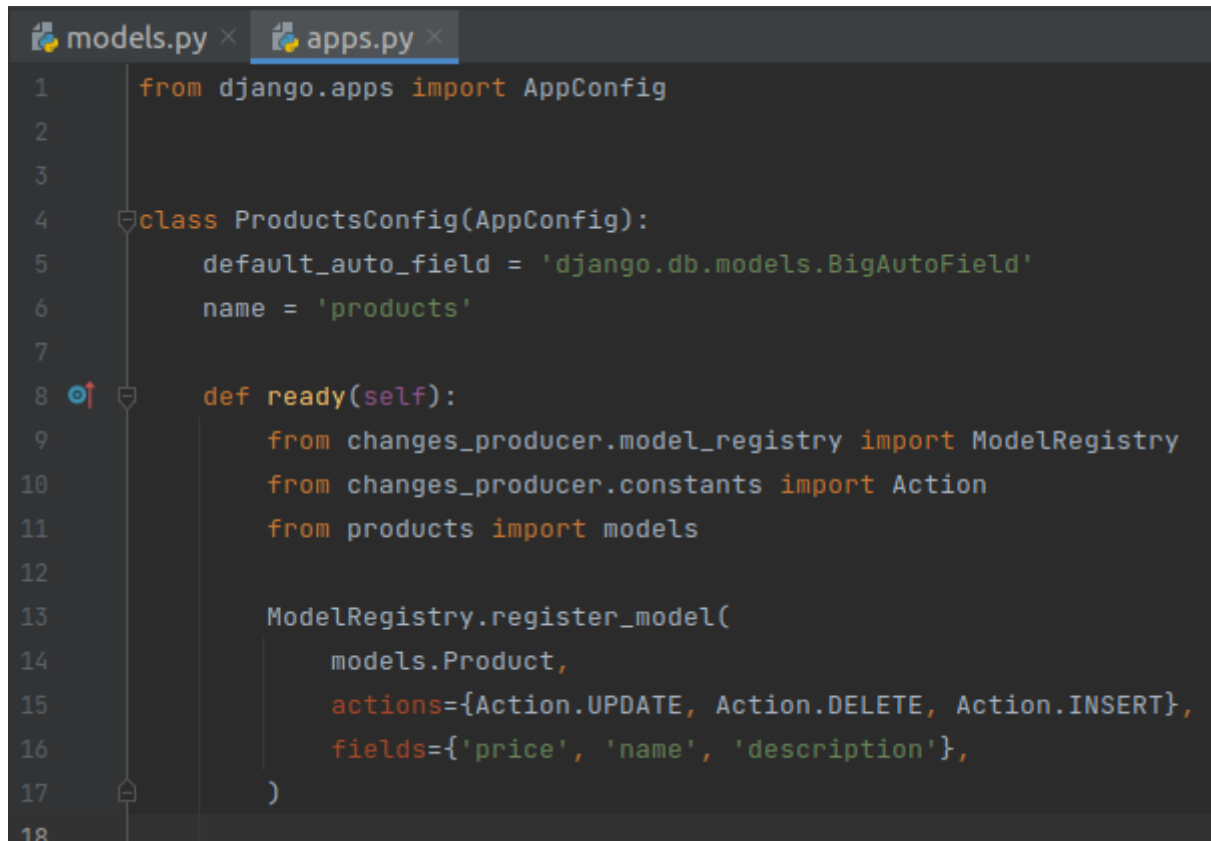
Рисунок 1.4 – Django-модель

Підключення моделей виконується у файлі apps.py, для кожного відповідного Django-app. Необхідно реалізувати метод ready класу, що розширює клас AppConfig. Для підключення використовується метод ModelRegistry.register_model. Метод приймає наступні параметри:

- модель, що необхідно підключити до системи відстеження змін;

- список подій, що мають відстежуватися;
- список полів моделі зміни яких необхідно відстежити.

На рисунку 1.5 зображено підключення моделі до системи відстеження.

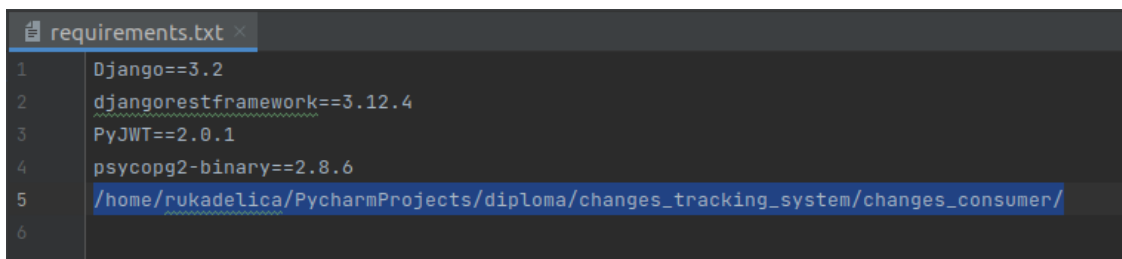


```
1 from django.apps import AppConfig
2
3
4 class ProductsConfig(AppConfig):
5     default_auto_field = 'django.db.models.BigAutoField'
6     name = 'products'
7
8     def ready(self):
9         from changes_producer.model_registry import ModelRegistry
10        from changes_producer.constants import Action
11        from products import models
12
13        ModelRegistry.register_model(
14            models.Product,
15            actions={Action.UPDATE, Action.DELETE, Action.INSERT},
16            fields={'price', 'name', 'description'},
17        )
18
```

Рисунок 1.5 – Підключення моделі до системи відстеження змін

2 СПОЖИВАЧ ПОВІДОМЛЕНЬ

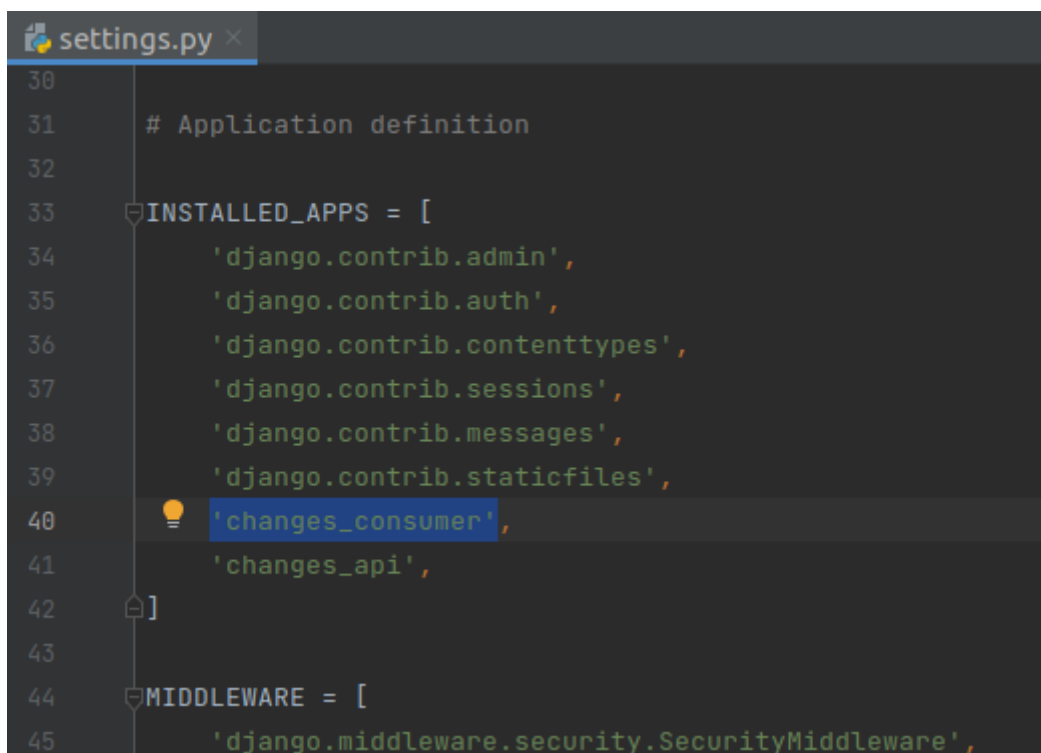
Для застосування функціоналу споживача повідомлень необхідно встановити пагін споживача повідомлень та додати його до списку залежностей відповідного сервісу, як зображено на рисунку 2.1.



```
requirements.txt
1 Django==3.2
2 djangorestframework==3.12.4
3 PyJWT==2.0.1
4 psycopg2-binary==2.8.6
5 /home/rukadelica/PycharmProjects/diploma/changes_tracking_system/changes_consumer/
6
```

Рисунок 2.1 – Файл зі списком залежностей

В файлі з налаштуваннями застосунку необхідно додати назву Django-арр, що надається плагіном, як зображено на рисунку 2.2, а також додати налаштування для kafka та логування, як зображено на рисунку 2.3.



```
settings.py
30
31 # Application definition
32
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'changes_consumer',
41     'changes_api',
42 ]
43
44 MIDDLEWARE = [
45     'django.middleware.security.SecurityMiddleware',
```

Рисунок 2.2 – Файл із налаштуваннями Django-застосунку

```

124
125
126 # Static files (CSS, JavaScript, Images)
127 # https://docs.djangoproject.com/en/3.2/howto/static-files/
128
129 STATIC_URL = '/static/'
130
131 # Default primary key field type
132 # https://docs.djangoproject.com/en/3.2/ref/settings/#default-auto-field
133
134 DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
135
136 CHANGES_TRACKING_TOPIC = 'database_changes'
137 CHANGES_TRACKING_BOOTSTRAP_SERVERS = 'localhost:29092'
138 CHANGES_CONSUMER_LOG_LEVEL = 'DEBUG'
139

```

Рисунок 2.3 - – Файл із налаштуваннями Django-застосунку

Після налаштування плагіну можна використовувати команду `run_consumer`, що надається Django-плагіном споживача повідомлень, як показано на рисунку 2.4.

```

admin_service$ python manage.py run_consumer

```

Рисунок 2.4 – Команда запуску споживача повідомлень

Плагін надає доступ до моделей, які застосовувалися для створення таблиць в базі даних, а отже можна використовувати їх для своїх потреб. На рисунку 2.5 зображено, як можна використовувати модель `ModelChange` для фільтрації збережених змін.

```
views.py
1 from rest_framework.views import APIView
2 from changes_api import serializers
3 from changes_consumer.models import ModelChange
4 from rest_framework.response import Response
5
6
7 class ChangesView(APIView):
8     def get(self, request):
9         if request.query_params:
10             serializer = serializers.ChangesQueryParamsSerializer(data=request.query_params)
11             serializer.is_valid(raise_exception=True)
12             query_params = serializer.validated_data
13         else:
14             query_params = {}
15         qs = ModelChange.objects.prefetch_related('field_changes').filter(**query_params)
16         return Response(serializers.ModelSerializer(qs, many=True).data)
17
```

Рисунок 2.5 – Приклад використання моделі ModelChange

Факультет інформатики та обчислювальної техніки
Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ Олександр ПАВЛОВ

“ ____ ” _____ 2021 р.

**БІБЛІОТЕКА ДЛЯ ЛОГУВАННЯ ЗМІН В МОДЕЛЯХ DJANGO-
ЗАСТОСУВАНЬ**

Опис програми

КПІ.ПІ-7205.045490.07.13

“ПОГОДЖЕНО”

Керівник проєкту:

_____ К.І. Ліщук

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ К. В. Вакуленко

Київ – 2021 року

Тексти програмного коду

Бібліотека для логування змін в моделях Django-застосувань

(Найменування програми (документа))

DVD-R

(Вид носія даних)

19 арк, 96 Кб

(Обсяг програми (документа) , арк.,)

Київ - 2021

					КПІ.ІП-7205.045490.07.13	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		2

run_consumer.py

```
from django.core.management.base import BaseCommand
from changes_consumer.consumer import Consumer
from changes_consumer import constants, loggers
from django.conf import settings

class Command(BaseCommand):

    def get_loggers(self):
        return {
            constants.LOG_LEVEL_ERROR: loggers.ErrorLevelLogger(),
            constants.LOG_LEVEL_DEBUG: loggers.DebugLevelLogger(),
            constants.LEVEL_NO_LOGGING: loggers.NoLoggingLevelLogger(),
        }

    def handle(self, *args, **options):
        consumer = Consumer(
            topic=settings.CHANGES_TRACKING_TOPIC,
            bootstrap_servers=settings.CHANGES_TRACKING_BOOTSTRAP_SERVERS,
            logger=self.get_loggers()[getattr(settings,
constants.CHANGES_CONSUMER_LOG_LEVEL, constants.LEVEL_NO_LOGGING)],
        )
        consumer.run()
```

0001_initial.py

```
from django.db import migrations, models
import django.db.models.deletion
```

```
class Migration(migrations.Migration):
```

```
    initial = True
```

```
    dependencies = [
    ]
```

					КПІ.ІП-7205.045490.07.13	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		3

```

operations = [
    migrations.CreateModel(
        name='ModelChange',
        fields=[
            ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
            ('actor_id', models.CharField(max_length=255)),
            ('action', models.CharField(max_length=255)),
            ('row_id', models.CharField(max_length=255)),
            ('table_name', models.CharField(max_length=255)),
            ('action_datetime', models.DateTimeField()),
        ],
    ),
    migrations.CreateModel(
        name='FieldChange',
        fields=[
            ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
            ('field', models.CharField(max_length=255)),
            ('old_value', models.CharField(max_length=255)),
            ('new_value', models.CharField(max_length=255)),
            ('field_type', models.CharField(max_length=255)),
            ('model_change',
models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, related_name='field_changes',
to='changes_consumer.modelchange')),
        ],
    ),
]

```

apps.py

```
from django.apps import AppConfig
```

```
class ChangesConsumerConfig(AppConfig):
```

```
    default_auto_field = 'django.db.models.BigAutoField'
```

```
    name = 'changes_consumer'
```

constants.py

					КПІ.ІП-7205.045490.07.13	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		4

```
import enum
```

```
class Action(enum.Enum):
```

```
    INSERT = 'insert'
```

```
    UPDATE = 'update'
```

```
    DELETE = 'delete'
```

```
CHANGES_CONSUMER_CUSTOM_SETTINGS
```

=

```
'CHANGES_CONSUMER_CUSTOM_SETTINGS'
```

```
CHANGES_TRACKING_BOOTSTRAP_SERVERS
```

=

```
'CHANGES_TRACKING_BOOTSTRAP_SERVERS'
```

```
CHANGES_TRACKING_TOPIC = 'CHANGES_TRACKING_TOPIC'
```

```
CHANGES_CONSUMER_LOG_LEVEL = 'CHANGES_CONSUMER_LOG_LEVEL'
```

```
LOG_LEVEL_DEBUG = 'DEBUG'
```

```
LEVEL_NO_LOGGING = 'NO_LOGGING'
```

```
LOG_LEVEL_ERROR = 'ERROR'
```

```
consumer.py
```

```
from kafka import KafkaConsumer
```

```
from changes_consumer.schemas import ModelChangeMessageSchema
```

```
from marshmallow import ValidationError
```

```
from changes_consumer.message_saver import MessageSaver
```

```
class Consumer:
```

```
    def __init__(self, topic, bootstrap_servers, logger):
```

```
        self.consumer = KafkaConsumer(
```

```
            topic,
```

```
            bootstrap_servers=bootstrap_servers
```

```
        )
```

```
        self.logger = logger
```

```
    def run(self):
```

```
        for message in self.consumer:
```

					КПІ.ІП-7205.045490.07.13	Арк.
						5
Змн.	Арк.	№ докум.	Підпис	Дата		

```
try:
    message = ModelChangeMessageSchema().loads(message.value)
except ValidationError as err:
    self.logger.log_validation_error(err)
    continue
self.logger.log_message(message)
message_saver = MessageSaver(message)
message_saver.save()
```

loggers.py

```
import logging
```

```
class AbstractConsumerLogger:
```

```
    def log_validation_error(self, error):
        raise NotImplementedError
```

```
    def log_message(self, message):
        raise NotImplementedError
```

```
class DebugLevelLogger(AbstractConsumerLogger):
```

```
    def __init__(self):
        self.logger = logging.getLogger()
```

```
    def log_message(self, message):
        self.logger.warning(f'MESSAGE: {message}')
```

```
    def log_validation_error(self, error):
        self.logger.warning(f'VALIDATION ERROR: {error.messages}')
```

```
class ErrorLevelLogger(AbstractConsumerLogger):
```

```
    def __init__(self):
        self.logger = logging.getLogger()
```

```
    def log_message(self, message):
```

					КПІ.ІП-7205.045490.07.13	Арк.
						6
Змн.	Арк.	№ докум.	Підпис	Дата		

```
return
```

```
def log_validation_error(self, error):  
    self.logger.warning(f'VALIDATION ERROR: {error.messages}')
```

```
class NoLoggingLevelLogger(AbstractConsumerLogger):
```

```
    def log_validation_error(self, error):  
        return
```

```
    def log_message(self, message):  
        return
```

message_saver.py

```
from changes_consumer.schemas import ModelChangeMessageSchema  
from changes_consumer import models, constants
```

```
class MessageSaver:
```

```
    def __init__(self, message: ModelChangeMessageSchema):  
        self.message = message
```

```
    def save(self):  
        model_change = models.ModelChange.objects.create(  
            actor_id=self.message['actor_id'],  
            action=self.message['action'],  
            row_id=self.message['row_id'],  
            table_name=self.message['table_name'],  
            action_datetime=self.message['action_datetime'],  
        )  
        if model_change.action != constants.Action.UPDATE.value:  
            return  
        for change in self.message['changes']:  
            models.FieldChange.objects.create(  
                model_change=model_change,  
                field=change['field'],  
                old_value=change['old_value'],
```

					КПІ.ІП-7205.045490.07.13	Арк.
						7
Змн.	Арк.	№ докум.	Підпис	Дата		

```
        new_value=change['new_value'],  
        field_type=change['field_type'],  
    )
```

models.py

```
from django.db import models
```

```
class ModelChange(models.Model):
```

```
    actor_id = models.CharField(max_length=255)  
    action = models.CharField(max_length=255)  
    row_id = models.CharField(max_length=255)  
    table_name = models.CharField(max_length=255)  
    action_datetime = models.DateTimeField()
```

```
class FieldChange(models.Model):
```

```
    model_change = models.ForeignKey(ModelChange, on_delete=models.CASCADE,  
related_name='field_changes')  
    field = models.CharField(max_length=255)  
    old_value = models.CharField(max_length=255)  
    new_value = models.CharField(max_length=255)  
    field_type = models.CharField(max_length=255)
```

schemas.py

```
from marshmallow import Schema, fields, validate  
from changes_consumer import constants
```

```
class FieldChangeSchema(Schema):
```

```
    field = fields.Str(required=True)  
    old_value = fields.Str(required=True)  
    new_value = fields.Str(required=True)  
    field_type = fields.Str(required=True)
```

					КПІ.ІП-7205.045490.07.13	Арк.
						8
Змн.	Арк.	№ докум.	Підпис	Дата		

```
class ModelChangeMessageSchema(Schema):
    actor_id = fields.Str(required=True)
    action = fields.Str(validate=validate.OneOf([action.value for action in constants.Action]),
required=True)
    row_id = fields.Str(required=True)
    table_name = fields.Str(required=True)
    action_datetime = fields.DateTime(required=True)
    changes = fields.List(fields.Nested(FieldChangeSchema), required=True)
```

setup.py

```
import setuptools
```

```
setuptools.setup(
    name='changes_consumer',
    version='1.0.0',
    packages=setuptools.find_packages(exclude=['core']),
    install_requires=[
        'kafka-python',
        'marshmallow',
    ]
)
```

changes_tracker.py

```
from changes_producer.messages import FieldChange
from changes_producer.message_builders import ChangeBuilder
from django.db import models
from typing import Set, Iterable, Optional, NamedTuple, List
```

```
class ChangesTrackerResult(NamedTuple):
    tracked_changes: List[FieldChange]
    is_instance_updated: bool
```

					КПІ.ІП-7205.045490.07.13	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		9

```
class ChangesTracker:
    change_builder_cls = ChangeBuilder

    def __init__(self,
                  instance: models.Model,
                  original_instance: models.Model,
                  fields_to_track: Set[str],
                  update_fields: Optional[Iterable[str]]):
        self.instance = instance
        self.original_instance = original_instance
        self.fields_to_track = fields_to_track & (set(update_fields) if update_fields else fields_to_track)

    def get_fields_changes(self) -> List[FieldChange]:
        result = []
        for field in self.instance._meta.get_fields():
            field_value = field.value_from_object(self.instance)
            field_name = field.name
            try:
                old_value = getattr(self.original_instance, field_name)
            except AttributeError:
                continue
            if field_value == old_value:
                continue
            change_builder = self.change_builder_cls(self.instance, field, old_value)
            result.append(change_builder.build_change())
        return result

    def get_changes(self) -> ChangesTrackerResult:
        fields_changes = self.get_fields_changes()
        changes = [
            field_change for field_change in fields_changes
            if field_change.field in self.fields_to_track
        ]
        return ChangesTrackerResult(
```

					КПІ.ІП-7205.045490.07.13	Арк.
						10
Змн.	Арк.	№ докум.	Підпис	Дата		

```
        tracked_changes=changes,
        is_instance_updated=bool(fields_changes)
    )
```

constants.py

```
import enum
```

```
class Action(enum.Enum):
```

```
    INSERT = 'insert'
```

```
    UPDATE = 'update'
```

```
    DELETE = 'delete'
```

```
ORIGINAL_INSTANCE_KEY = '_changes_producer_original_instance'
```

```
CHANGES_PRODUCER_CUSTOM_SETTINGS = 'CHANGES_PRODUCER_CUSTOM_SETTINGS'
```

```
CHANGES_TRACKING_BOOTSTRAP_SERVERS =
'CHANGES_TRACKING_BOOTSTRAP_SERVERS'
```

```
CHANGES_TRACKING_TOPIC = 'CHANGES_TRACKING_TOPIC'
```

current_user.py

```
class CurrentRequestUserMiddleware:
```

```
    current_request = None
```

```
    def __init__(self, get_response):
```

```
        self.get_response = get_response
```

```
    def __call__(self, request):
```

```
        self.__class__.current_request = request
```

```
        response = self.get_response(request)
```

```
        self.__class__.current_request = None
```

```
        return response
```

					КПІ.ІП-7205.045490.07.13	Арк.
						11
Змн.	Арк.	№ докум.	Підпис	Дата		

```
def get_current_user():
    try:
        return CurrentRequestUserMiddleware.current_request.user
    except AttributeError:
        return None
```

message_builders.py

```
import datetime
from changes_producer.constants import Action
from changes_producer.current_user import get_current_user
from changes_producer.messages import ModelChangeMessage, FieldChange
from django.db import models
from typing import List, Optional, Any
```

```
class MessageBuilder:
    def __init__(self, instance: models.Model):
        self.instance = instance

    def get_actor_id(self) -> str:
        return str(get_current_user().id)

    def get_row_id(self) -> str:
        return str(self.instance.pk)

    def get_table_name(self) -> str:
        return self.instance._meta.db_table

    def get_action_datetime(self) -> datetime.datetime:
        return datetime.datetime.now()

    def build_message(self, action: Action, changes: Optional[List[FieldChange]] = None) ->
    ModelChangeMessage:
        return ModelChangeMessage(
```

					КПІ.ІП-7205.045490.07.13	Арх.
Змн.	Арх.	№ докум.	Підпис	Дата		12

```

        actor_id=self.get_actor_id(),
        action=action.value,
        row_id=self.get_row_id(),
        table_name=self.get_table_name(),
        changes=changes if changes else [],
        action_datetime=self.get_action_datetime(),
    )

```

```

class ChangeBuilder:

```

```

    def __init__(self, instance: models.Model, field: models.Field, old_value: Any):

```

```

        self.instance = instance

```

```

        self.field = field

```

```

        self.old_value = old_value

```

```

    def get_field(self) -> str:

```

```

        return self.field.name

```

```

    def get_old_value(self) -> str:

```

```

        return str(self.old_value)

```

```

    def get_new_value(self) -> str:

```

```

        return str(self.field.value_from_object(self.instance))

```

```

    def get_field_type(self) -> str:

```

```

        return str(type(self.field))

```

```

    def build_change(self) -> FieldChange:

```

```

        return FieldChange(

```

```

            field=self.get_field(),

```

```

            old_value=self.get_old_value(),

```

```

            new_value=self.get_new_value(),

```

```

            field_type=self.get_field_type(),

```

```

        )

```

message_sender.py

```
from django.conf import settings
```

```
import kafka
```

```
from changes_producer.messages import ModelChangeMessage
```

```
from changes_producer.message_serializer import MessageSerializer
```

```
class MessageSender:
```

```
    message_serializer_cls = MessageSerializer
```

```
    def __init__(self, message: ModelChangeMessage):
```

```
        self.message = message
```

```
    def send_message(self):
```

```
        message_serializer = self.message_serializer_cls(self.message)
```

```
        producer
```

=

```
kafka.KafkaProducer(bootstrap_servers=settings.CHANGES_TRACKING_BOOTSTRAP_SERVERS)
```

```
        producer.send(settings.CHANGES_TRACKING_TOPIC, message_serializer.to_json().encode())
```

```
        producer.close()
```

message_serializer.py

```
from changes_producer.messages import ModelChangeMessage
```

```
from django.core.serializers.json import DjangoJSONEncoder
```

```
import json
```

```
class MessageSerializer:
```

```
    json_encoder_cls = DjangoJSONEncoder
```

```
    def __init__(self, message: ModelChangeMessage):
```

```
        self.message = message
```

```
    def to_json(self) -> str:
```

```
        changes = [change._asdict() for change in self.message.changes]
```

```
        message = self.message._asdict()
```

					КПІ.ІП-7205.045490.07.13	Арк.
						14
Змн.	Арк.	№ докум.	Підпис	Дата		

```
message['changes'] = changes  
return json.dumps(message, cls=self.json_encoder_cls)
```

messages.py

```
import datetime  
from typing import NamedTuple, List
```

```
class FieldChange(NamedTuple):
```

```
    field: str  
    old_value: str  
    new_value: str  
    field_type: str
```

```
class ModelChangeMessage(NamedTuple):
```

```
    actor_id: str  
    action: str  
    row_id: str  
    table_name: str  
    changes: List[FieldChange]  
    action_datetime: datetime.datetime
```

model_registry.py

```
from django.db import models  
from changes_producer.constants import Action  
from typing import Set, Type, Optional, Dict  
from changes_producer import message_sender, receiver_connectors, message_builders, changes_tracker
```

```
class ModelRegistry:
```

```
    message_sender_cls = message_sender.MessageSender  
    message_builder_cls = message_builders.MessageBuilder  
    changes_tracker_cls = changes_tracker.ChangesTracker
```

					КПІ.ІП-7205.045490.07.13	Арк.
						15
Змн.	Арк.	№ докум.	Підпис	Дата		

```

def __init__(self, model: Type[models.Model], actions: Set[Action], fields: Set[str]):
    self.model = model
    self.actions = actions
    self.fields = fields

@classmethod
def register_model(cls,
                    model: Type[models.Model],
                    actions: Optional[Set[Action]] = None,
                    fields: Optional[Set[str]] = None):
    if actions is None:
        actions = {Action.INSERT, Action.UPDATE, Action.DELETE}
    if fields is None:
        fields = {field.name for field in model._meta.get_fields()}
    model_registry = cls(model, actions, fields)
    model_registry.validate_model_registry()
    model_registry.connect_actions()

def get_connectors(self) -> Dict[Action, receiver_connectors.AbstractReceiverConnector]:
    return {
        Action.INSERT: self.create_insert_connector(),
        Action.DELETE: self.create_delete_connector(),
        Action.UPDATE: self.create_update_connector(),
    }

def connect_actions(self):
    connectors = self.get_connectors()
    for action in self.actions:
        connectors[action].connect()

def create_insert_connector(self):
    return receiver_connectors.InsertActionReceiverConnector(
        self.model,
        self.message_sender_cls,
        self.message_builder_cls

```

)

def create_delete_connector(self):

```

return receiver_connectors.DeleteActionReceiverConnector(
    self.model,
    self.message_sender_cls,
    self.message_builder_cls
)

```

def create_update_connector(self):

```

return receiver_connectors.UpdateActionReceiverConnector(
    self.model,
    self.message_sender_cls,
    self.message_builder_cls,
    self.changes_tracker_cls,
    self.fields,
)

```

def validate_model_registry(self):

```

if not issubclass(self.model, models.Model):
    raise ValueError(f'<model> {self.model} is not a subclass of {models.Model}.')

```

```

if not isinstance(self.actions, set):
    raise ValueError(f'<actions> is not of {set} type.')

```

```

for action in self.actions:
    if not isinstance(action, Action):
        raise ValueError(f'all elements of <actions> have to be of {Action} type.')

```

```

if not isinstance(self.fields, set):
    raise ValueError(f'<fields> is not of {set} type.')

```

```

model_fields = {field.name for field in self.model._meta.get_fields()}

```

```

for field in self.fields:

```

```

    if field not in model_fields:
        raise ValueError(f'Invalid <fields> values available options: {model_fields}.')

```

receiver_connectors.py

```

from django.db import models
from typing import Type, Set
from django.db.models import signals
from changes_producer.message_sender import MessageSender
from changes_producer.message_builders import MessageBuilder
from changes_producer import constants
from changes_producer.current_user import get_current_user
from changes_producer.changes_tracker import ChangesTracker

```

```

class AbstractReceiverConnector:

```

```

    def __init__(self,
                  model: Type[models.Model],
                  message_sender_cls: Type[MessageSender],
                  message_builder_cls: Type[MessageBuilder]):
        self.model = model
        self.message_sender_cls = message_sender_cls
        self.message_builder_cls = message_builder_cls

```

```

    def connect(self):
        raise NotImplementedError

```

```

class DeleteActionReceiverConnector(AbstractReceiverConnector):

```

```

    def build_send_message_receiver(self):
        def send_delete_action_message(sender, instance, **kwargs):
            current_user = get_current_user()
            if not current_user:
                return
            message_builder = self.message_builder_cls(instance)
            message = message_builder.build_message(constants.Action.DELETE)
            message_sender = self.message_sender_cls(message)
            message_sender.send_message()
        return send_delete_action_message

```

```
def connect(self):
    signals.post_delete.connect(
        self.build_send_message_receiver(),
        sender=self.model,
        weak=False,
    )
```

```
class InsertActionReceiverConnector(AbstractReceiverConnector):
    def build_send_message_receiver(self):
        def send_insert_action_message(sender, instance, created, **kwargs):
            current_user = get_current_user()
            if not current_user:
                return
            if not created:
                return
            message_builder = self.message_builder_cls(instance)
            message = message_builder.build_message(constants.Action.INSERT)
            message_sender = self.message_sender_cls(message)
            message_sender.send_message()

        return send_insert_action_message

    def connect(self):
        signals.post_save.connect(
            self.build_send_message_receiver(),
            sender=self.model,
            weak=False,
        )
```

```
def set_original_instance(sender, instance, **kwargs):
    current_user = get_current_user()
    if not current_user:
        return
```

```
original_instance_qs = sender.objects.filter(pk=instance.pk)
if not original_instance_qs:
    return
original_instance = original_instance_qs.first()
setattr(instance, constants.ORIGINAL_INSTANCE_KEY, original_instance)
```

```
class UpdateActionReceiverConnector(AbstractReceiverConnector):
    def __init__(self,
        model: Type[models.Model],
        message_sender_cls: Type[MessageSender],
        message_builder_cls: Type[MessageBuilder],
        changes_tracker_cls: Type[ChangesTracker],
        fields: Set[str]):
        super().__init__(model, message_sender_cls, message_builder_cls)
        self.changes_tracker_cls = changes_tracker_cls
        self.fields = fields

    def build_send_message_receiver(self):
        def send_update_action_message(sender, instance, created, raw, using, update_fields, **kwargs):
            current_user = get_current_user()
            if not current_user:
                return
            if created:
                return
            original_instance = getattr(instance, constants.ORIGINAL_INSTANCE_KEY)
            delattr(instance, constants.ORIGINAL_INSTANCE_KEY)
            changes_tracker = self.changes_tracker_cls(instance, original_instance, self.fields, update_fields)
            changes = changes_tracker.get_changes()
            if not changes.is_instance_updated:
                return
            message_builder = self.message_builder_cls(instance)
            message = message_builder.build_message(constants.Action.UPDATE,
changes.tracked_changes)
            message_sender = self.message_sender_cls(message)
            message_sender.send_message()
```

```

        return send_update_action_message
    
```

```

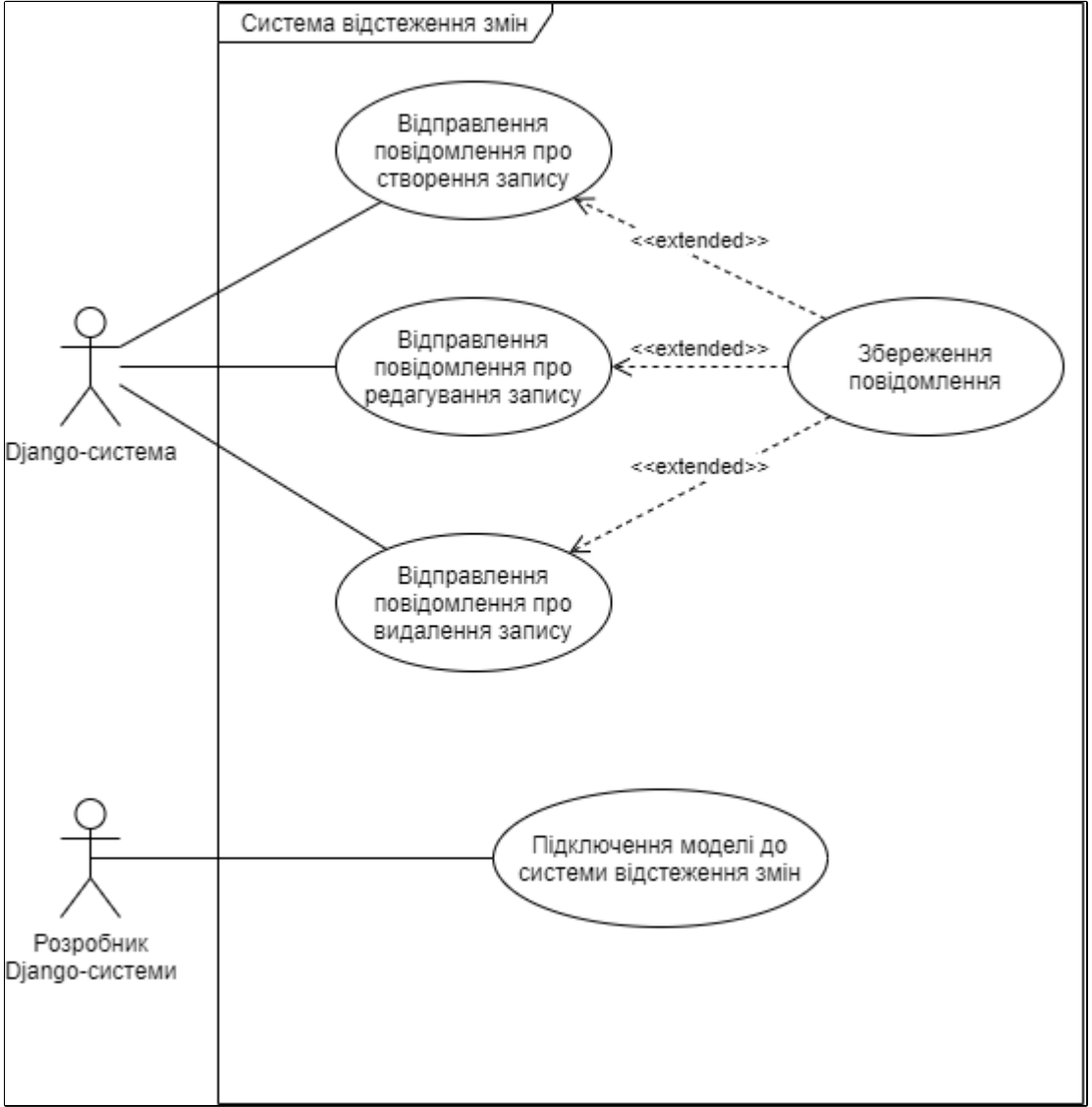
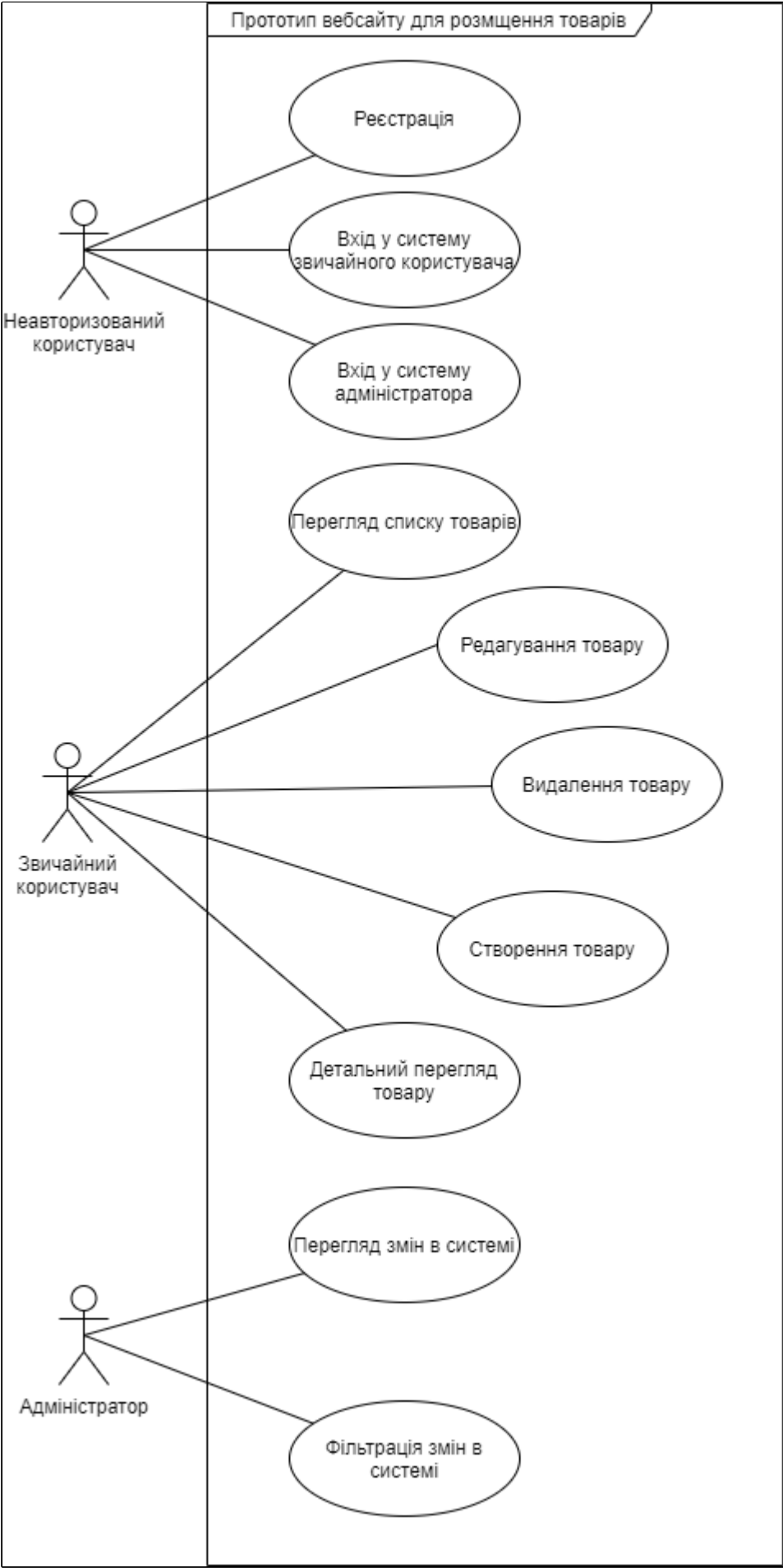
def connect(self):
    signals.post_save.connect(
        self.build_send_message_receiver(),
        sender=self.model,
        weak=False,
    )
    signals.pre_save.connect(
        set_original_instance,
        sender=self.model,
        weak=False,
    )
    
```

setup.py

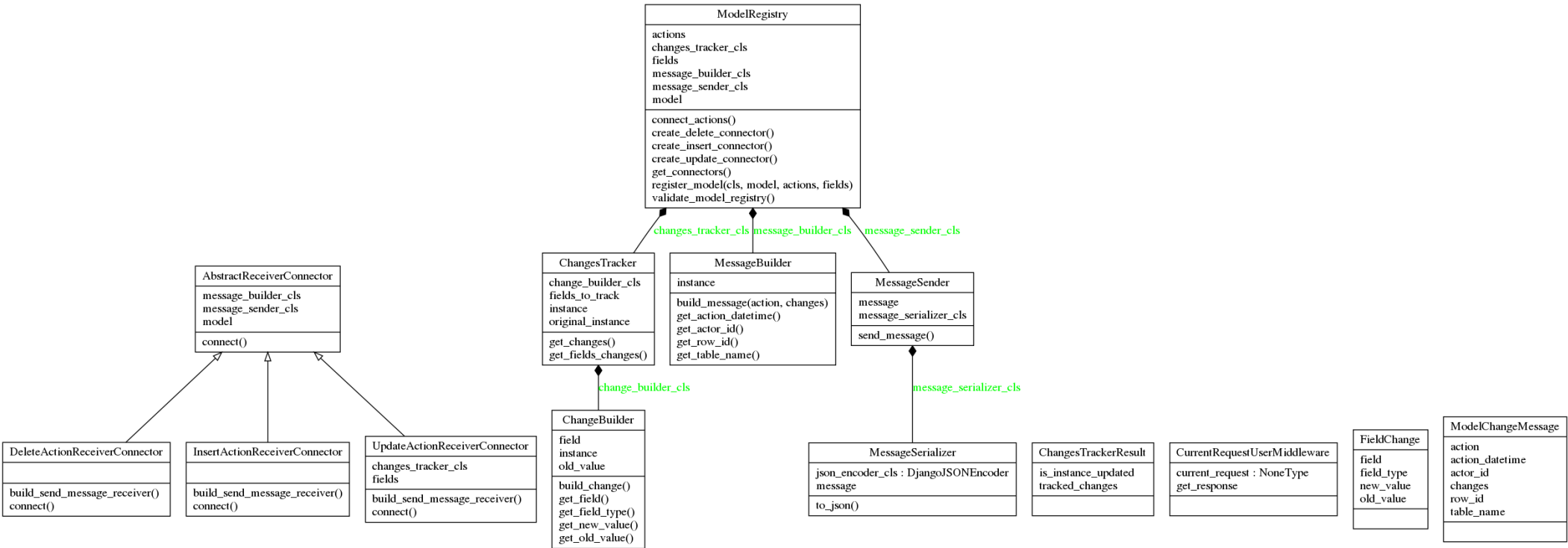
```

import setuptools

setuptools.setup(
    name='changes_producer',
    version='1.0.0',
    packages=setuptools.find_packages(),
    install_requires=[
        'kafka-python',
    ]
)
    
```

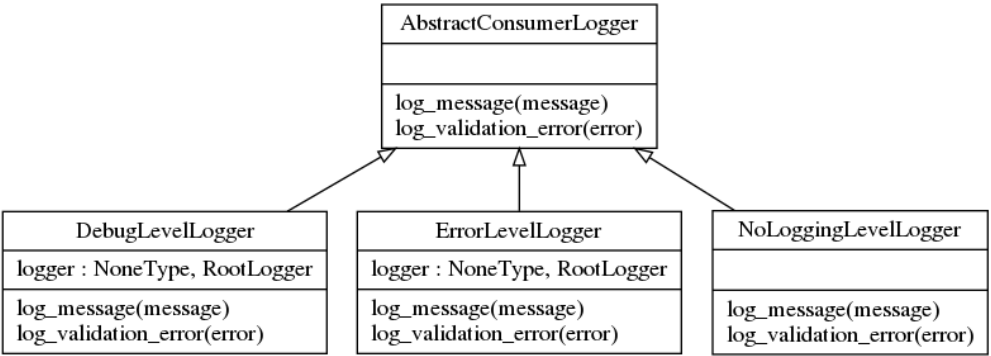


					КПІ.ІП-7205.045490.08.99.СС					
					Схема структурна варіантів використань	Літера			Маса	Масштаб
Зм.	Арк.	№ документа	Підпис	Дата						
Розробив		Вакуленко К.В.								
Перевірив		Ліщук К.І.								
Т. кон.										
						Аркуш			Аркушів	
					Бібліотека для логування змін в моделях Django-застосувань	КПІ ім.Ігоря Сікорського Кафедра АСОІУ гр. ІП-72				
Н. кон.		Ліщук К.І.								
Затвердив		Ліщук К.І.								



FieldChangeSchema
field : String
field_type : String
new_value : String
old_value : String

ModelChangeMessageSchema
action : String
action_datetime : DateTime
actor_id : String
changes : List
row_id : String
table_name : String



FieldChange
field : CharField
field_type : CharField
model_change : ForeignKey
new_value : CharField
old_value : CharField

ModelChange
action : CharField
action_datetime : DateTimeField
actor_id : CharField
row_id : CharField
table_name : CharField

					КПІ.ІП-7205.045490.09.99.СС				
					Схема структурна класів програмного забезпечення				
Зм.	Арк.	№ документа	Підпис	Дата					
Розробив	Вакуленко К.В.				Бібліотека для логування змін в моделях Django-застосувачів				
Перевірів	Ліщук К.І.								
Т. кон.									
					КПІ ім.Ігоря Сікорського Кафедра АСОІУ гр. ІП-72				
Н. кон.	Ліщук К.І.								
Затвердив	Ліщук К.І.								
					Літера				
					Маса				
					Масштаб				
					Аркуш				
					Аркушів				

Login

Username

Password

Click [here](#) to register

Sign in

Admin Login

admin

Sign in

Filter

Changes:

Actor: user1

Action: insert

Row Id: 13

Table Name: products_product

Action Datetime: 2021-05-16T12:03:41.169000Z

Registration

Dimitry

Register

Product Name

Dimitiry's Updated product N1

Price

133.05

Description

Updated Product description

Save changes

Actor: user1

Action: update

Row Id: 13

Table Name: products_product

Action Datetime: 2021-05-16T12:04:00.526000Z

Field: price

Old value: 344.00

Product Name

Dimitiry's product N1

Price

12.05

Description

Product description

Add Product

Product Name:

Dimitiry's product N1

Price:

12.05

Owner:

Dimitry

Description:

Product description

Edit

add Product

Products:

Product Name: Modded 4 PP

Price: 1.09

Owner: User1

[detail](#)

Product Name: Modded 4 PP

Price: 1.11

Owner: User1

[detail](#)

Product Name: New Product

Filter

Changes:

Actor: user1

Action: insert

Row Id: 13

Table Name: products_product

Action Datetime: 2021-05-16T12:03:41.169000Z

Actor: user1

Action: update

Row Id: 13

Table Name: products_product

Action Datetime: 2021-05-16T12:04:00.526000Z

Filters

Dimitry

2021-05-30 ~ 2021-05-30

Apply

Close

					КПІ.ІП-7205.045490.10.99.KE										
					Креслення вигляду екранних форм										
Зм.	Арк.	№ документа	Підпис	Дата	Креслення вигляду екранних форм										
Розробив	Вакуленко К.В.														
Перевірів	Ліщук К.І.														
Т. кон.															
					Бібліотека для логування змін в моделях Django-застосувань										
Н. кон.	Ліщук К.І.														
Затвердив	Ліщук К.І.														
					Літера			Маса			Масштаб				
								Аркуш			Аркушів				
					КПІ ім.Ігоря Сікорського Кафедра АСОІУ гр. ІП-72										