Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(рівень вищої освіти)

на тему: **«Розроблення програмного забезпечення для роботи менеджера з прокату авто засобами Phython»**
(тема роботи)

Виконав: студент 2 курсу групи КНс-21
Спеціальності 122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Кондрин Софія Василівна
(прізвище та ініціали)

Керівник Паславський М.М.
(прізвище та ініціали)

Рецензент Часковський О. Г.
(прізвище та ініціали)

Львів-2025

Національний лісотехнічний університет України
(назва навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра: комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 "Комп'ютерні науки"

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
_____ Борецька І.Б.

"10" червня 2025р.

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Кондрин Софія Василівна
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення програмного забезпечення для роботи менеджера з прокату авто засобами Phython / Development of software for the work of a car rental manager by means of Phython
керівник роботи *Паславський М.М, к.т.н., асистент кафедри КН*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "15" листопада 2024 року №С-887
2. Термін подання студентом роботи 10 червня 2025 р.
3. Вихідні дані до роботи розробити програмне забезпечення для автоматизації роботи менеджера з прокату автомобілів. Система повинна забезпечувати ефективне управління клієнтами, автопарком, орендними угодами та фінансовими транзакціями.
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
ВСТУП
РОЗДІЛ 1. Стан проблемної області
РОЗДІЛ 2. Інформаційне та математичне забезпечення
РОЗДІЛ 3. Програмне та технічне забезпечення
ВИСНОВКИ
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Підготовка матеріалу до доповіді .
6. Дата видачі завдання 18 листопада 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Збір та опрацювання матеріалу за темою дипломної роботи	18.11.2024-26.11.2024	Виконано
2	Проєктування програми	26.11.2024-14.12.2024	Виконано
3	Розробка програми	14.12.2024-24.12.2024	Виконано
4	Тестування програми	24.12.2024-20.01.2025	Виконано
5	Внесення правок у програму	20.01.2025-30.01.2025	Виконано
6	Розробка першого розділу пояснювальної записки	30.01.2025-20.02.2025	Виконано
7	Розробка другого розділу пояснювальної записки	20.02.2025-15.02.2025	Виконано
8	Розробка третього розділу пояснювальної записки	15.02.2025-20.03.2025	Виконано
9	Оформлення пояснювальної записки	20.03.2025-10.04.2025	Виконано

Студент _____ **Кондрин С.В.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Паславський М.М.**
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

У дипломній роботі розглянуто процес розроблення програмного забезпечення для автоматизації роботи менеджера з прокату автомобілів. Основна увага приділена аналізу предметної області, вибору сучасних інструментів розробки, таких як Python, Django та PostgreSQL, і реалізації функціоналу для ефективного управління клієнтами, автопарком, орендними угодами та фінансовими транзакціями. Запропонована система дозволяє автоматизувати рутинні завдання, забезпечуючи простоту використання, надійність даних і можливість інтеграції з іншими сервісами. Реалізовано інструменти експорту даних, планування завдань та інтерфейс зручний для користувачів. Оцінено ефективність системи через тестування та економічне обґрунтування впровадження.

Ключові слова: *прокат автомобілів, автоматизація, Python, Django, PostgreSQL, управління клієнтами, управління автопарком, програмне забезпечення.*

ABSTRACT

This diploma thesis examines the process of developing software to automate the work of a car rental manager. The focus is on analysing the subject area, selecting modern development tools such as Python, Django, and PostgreSQL, and implementing functionality for efficient management of clients, vehicle fleets, rental agreements, and financial transactions. The proposed system automates routine tasks, ensuring ease of use, data reliability, and integration capability with other services. The solution includes tools for data export, task scheduling, and a user-friendly interface. The system’s effectiveness has been evaluated through testing and economic justification for its implementation.

Keywords: *Car rental, Automation, Django, PostgreSQL, Client management, Fleet management, Software.*

ТЕХНІЧНЕ ЗАВДАННЯ

У відповідності з вимогами технічного завдання необхідно розробити програмне забезпечення для автоматизації роботи менеджера з прокату автомобілів. Система повинна забезпечувати ефективне управління клієнтами, автопарком, орендними угодами та фінансовими транзакціями.

Користувачі програми повинні мати можливість швидко реєструвати нових клієнтів, зберігаючи їх основні дані, включаючи контактну інформацію, паспортні дані та номер посвідчення водія. Необхідно передбачити функціонал для додавання, перегляду та редагування інформації про автомобілі, таких як марка, модель, номерний знак, технічні характеристики та страховка. Система повинна дозволяти створювати, переглядати та управляти договорами оренди, включаючи автоматичний розрахунок вартості оренди з урахуванням додаткових послуг, таких як страхування, оренда додаткового обладнання або послуги водія. Важливо забезпечити можливість фіксації пробігу автомобілів і оновлення відповідних показників у базі даних. Окрім цього, система повинна дозволяти реєструвати та контролювати фінансові транзакції, включаючи доходи та витрати.

Для забезпечення функціональності необхідно створити базу даних, яка зберігатиме інформацію про клієнтів, автомобілі, оренди та транзакції. Веб-інтерфейс програми має бути зручним для введення, перегляду та редагування даних. Важливо впровадити автоматизацію рутинних завдань, таких як щоденне виконання запланованих завдань (нагадування, оновлення даних) за допомогою планувальника. Окрему увагу слід приділити функціоналу експорту даних про оренди у форматі Excel.

Система має підтримувати мультимовність, початково реалізовану українською мовою, з можливістю адаптації до інших мов. Особлива увага повинна бути приділена захисту даних, зокрема впровадженню авторизації та автентифікації користувачів. Дані повинні перевірятися на унікальність, включаючи паспорти, посвідчення водія та контактну інформацію клієнтів. Завершальним етапом розробки є проведення юніт- і інтеграційного тестування для перевірки роботи основних функцій системи.

Програмне забезпечення має бути зручним у використанні, адаптивним до різних пристроїв та відповідати сучасним стандартам веб-розробки.

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення.

CRM (Customer Relationship Management) – Система управління взаємовідносинами з клієнтами, що забезпечує збереження, аналіз і використання даних для підвищення ефективності обслуговування.

API (Application Programming Interface) – Інтерфейс програмування додатків, що дозволяє взаємодіяти між різними програмами чи компонентами програмного забезпечення.

PDF (Portable Document Format) – Формат електронних документів, який забезпечує збереження вигляду документа незалежно від програмного чи апаратного середовища.

SCDW (Super Collision Damage Waiver) – Повне страхування від пошкоджень транспортного засобу.

ER-діаграма (Entity-Relationship Diagram) – Діаграма, яка описує зв'язки між сутностями в базі даних.

UI (User Interface) – Інтерфейс користувача, який дозволяє взаємодіяти з програмним забезпеченням.

UX (User Experience) – Враження користувача від роботи з програмним забезпеченням, що включає зручність, функціональність та дизайн.

CRUD (Create, Read, Update, Delete) – Основні операції для роботи з базою даних.

HTTP (Hypertext Transfer Protocol) – Протокол передачі гіпертексту, що використовується для обміну даними між веб-браузером та сервером.

HTML (Hypertext Markup Language) – Мова розмітки гіпертексту, яка використовується для створення веб-сторінок.

CSS (Cascading Style Sheets) – Каскадні таблиці стилів, які використовуються для опису вигляду веб-сторінок.

ЗМІСТ

ВСТУП

Сучасний ринок прокату автомобілів характеризується високою конкуренцією та постійним зростанням вимог до швидкості та якості обслуговування клієнтів. Для забезпечення конкурентоспроможності компанії з прокату авто необхідно автоматизувати рутинні бізнес-процеси, що дозволить підвищити ефективність роботи менеджерів, мінімізувати людський фактор та покращити взаємодію з клієнтами. Одним із ключових інструментів для досягнення цих цілей є програмне забезпечення, здатне забезпечити комплексний підхід до управління автопарком, клієнтською базою та фінансовими транзакціями.

Об’єктом дослідження є процес управління взаємодією з клієнтами та автопарком у компаніях з прокату автомобілів.

Метою роботи є розробка програмного забезпечення для автоматизації роботи менеджера з прокату автомобілів, що забезпечує ефективне управління клієнтами, автомобілями, орендними угодами та фінансовими транзакціями.

Предметом дослідження є методи, засоби та інструменти автоматизації процесів управління прокатом автомобілів з використанням Python, Django та PostgreSQL.

Практичне значення роботи полягає у створенні програмного забезпечення, яке дозволяє оптимізувати процеси реєстрації клієнтів, управління автопарком, обліку фінансів та оренди, підвищуючи загальну продуктивність підприємства. Система також може бути інтегрована в інші бізнес-процеси, що сприяє її універсальності.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Огляд проблемної області

Сфера прокату автомобілів є однією з найбільш конкурентоспроможних галузей на сучасному ринку послуг. Швидкість обслуговування клієнтів, точність обліку та управління автопарком є ключовими чинниками, що впливають на успішність компанії. У той же час, галузь стикається з низкою проблем, які ускладнюють діяльність підприємств та потребують інноваційних рішень для їх вирішення.

Однією з основних проблем є складність управління великим обсягом даних, що включає інформацію про клієнтів, транспортні засоби, договори оренди та фінансові транзакції. Ці дані повинні бути збережені, впорядковані та доступні для аналізу в реальному часі. Традиційні підходи, такі як ручне ведення записів або використання базових електронних таблиць, є застарілими і не відповідають сучасним вимогам. Вони схильні до помилок через людський фактор, мають низьку продуктивність та ускладнюють процеси масштабування бізнесу.

Ще однією важливою проблемою є відсутність ефективної системи автоматизації рутинних задач. Наприклад, облік пробігу автомобілів, нагадування про повернення транспортного засобу, розрахунок вартості оренди з урахуванням додаткових послуг, таких як страхування чи оренда додаткового обладнання, часто виконується вручну. Це призводить до затримок у роботі менеджерів, зниження продуктивності та незадоволення клієнтів.

Крім того, виникає проблема недостатньої інтеграції між різними системами, що використовуються підприємством. Наприклад, інформація про клієнтів може зберігатися в одній програмі, дані про автомобілі – в іншій, а фінансові транзакції – у третій. Це ускладнює аналіз і синхронізацію даних, а також збільшує витрати на підтримку та обслуговування різних програмних продуктів.

Ще одним викликом є забезпечення безпеки даних. Оскільки інформація про клієнтів і фінансові операції є конфіденційною, її витік може призвести до серйозних репутаційних втрат і штрафів. Багато компаній з прокату автомобілів використовують застарілі системи, які не відповідають сучасним стандартам захисту інформації.

Окрім цього, клієнти часто вимагають зручності у взаємодії з компанією. Відсутність інтерактивного інтерфейсу для швидкого доступу до інформації або замовлення послуг може знизити лояльність клієнтів і зменшити конкурентоспроможність компанії. Зручний веб-інтерфейс, що забезпечує прозорість і доступність, стає важливим елементом для задоволення очікувань клієнтів.

Таким чином, основними проблемами в галузі прокату автомобілів є: складність управління даними, необхідність автоматизації рутинних задач, відсутність інтеграції між системами, недостатній рівень безпеки даних та низький рівень зручності для клієнтів. Ці виклики вимагають розробки комплексного програмного забезпечення, яке забезпечить ефективність і автоматизацію бізнес-процесів.

У даному дослідженні увага приділяється створенню системи, що дозволяє вирішити зазначені проблеми. Використання сучасних технологій, таких як Python, Django і PostgreSQL, дозволяє забезпечити високу продуктивність, надійність та масштабованість рішення. Автоматизація процесів, інтеграція даних і впровадження зручного інтерфейсу користувача є ключовими аспектами для підвищення ефективності роботи підприємства та задоволення потреб клієнтів.

1.2. Можливості й переваги CRM системи

Розроблена CRM-система для управління прокатом автомобілів пропонує широкий функціонал, що охоплює всі ключові аспекти діяльності підприємств у цій сфері. Основні можливості системи включають управління клієнтською базою, автопарком, договорами оренди та фінансовими транзакціями.

Система дозволяє зберігати детальну інформацію про клієнтів, включаючи контактні дані, паспорти, посвідчення водія та історію оренд. Завдяки інтуїтивно зрозумілому веб-інтерфейсу менеджери можуть швидко створювати й редагувати записи, знаходити потрібну інформацію за кілька кліків.

Управління автопарком охоплює реєстрацію транспортних засобів, моніторинг їхнього технічного стану, пробігу та страхових полісів. Система автоматично розраховує вартість оренди з урахуванням терміну, курсу валюти, додаткових послуг, таких як страховка чи додаткове обладнання.

Важливою перевагою є автоматизація рутинних задач. Система підтримує планування завдань, таких як нагадування про повернення автомобіля, оновлення статусів договорів чи повідомлення клієнтів. Експорт даних у форматі Excel спрощує звітність і аналіз фінансових показників.

На відміну від багатьох аналогів, наша CRM-система забезпечує високий рівень захисту даних, впроваджує багатокористувацький доступ із різними рівнями привілеїв та дозволяє уникати дублювання інформації.

Таким чином, CRM-система підвищує продуктивність роботи менеджерів, мінімізує ризики помилок, забезпечує точність обліку та створює комфорт для клієнтів, що робить її незамінним інструментом у сфері прокату автомобілів.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

Інформаційне забезпечення є ключовим компонентом будь-якої сучасної інформаційної системи. Воно включає сукупність даних, методів їх обробки, інструментів та технологій, які забезпечують ефективну роботу програмного забезпечення. У даному розділі розглядаються основи проєктування та реалізації інформаційного забезпечення CRM-системи для прокату автомобілів, зокрема структура бази даних, методи обробки даних та забезпечення їхньої цілісності та безпеки.

2.1. Django ORM

Object-Relational Mapping (ORM) є потужним інструментом, що дозволяє розробникам взаємодіяти з базою даних на рівні об'єктів мови програмування Python. Django ORM, як частина фреймворку Django, забезпечує зручність роботи з даними, зводячи складні SQL-запити до простих викликів методів об'єктів [1]. Це значно підвищує продуктивність розробки та зменшує ризик помилок.

Основні можливості Django ORM:

1. **CRUD-операції:** Django ORM дозволяє створювати, читати, оновлювати та видаляти записи в базі даних за допомогою простих методів. Наприклад, створення нового клієнта реалізується викликом `Customer.objects.create()`, а пошук за конкретним критерієм — методом `filter()` або `get()`.

```python
# Приклад створення нового клієнта
customer = Customer.objects.create(
    first_name="Іван",
    last_name="Іваненко",
    email="ivan@example.com"
)

# Пошук клієнта за email
customer = Customer.objects.get(email="ivan@example.com")
```

Рисунок 2.1 Приклад створення та пошуку клієнта за допомогою Django ORM

2. **Автоматичне створення та виконання SQL-запитів:** Django ORM генерує SQL-запити на основі моделей і методів Python, що значно зменшує складність взаємодії з базою даних.
3. **Валідація та обмеження даних:** Django ORM забезпечує цілісність даних завдяки вбудованим механізмам валідації, що зменшує ризик некоректних записів у базу. Наприклад, у моделі `Customer` можна визначити унікальність електронної пошти або обов'язковість заповнення певних полів.
4. **Зв’язки між сутностями:** ORM підтримує зв’язки "один до одного", "один до багатьох" та "багато до багатьох". Це дозволяє легко працювати з пов’язаними об’єктами. Наприклад, отримання всіх оренд, пов’язаних з конкретним клієнтом:

```
rentals = customer.rental_set.all()
```

Рисунок 2.2 Отримання всіх оренд, пов’язаних із клієнтом, за допомогою Django ORM

5. **Міграції бази даних:** Django ORM дозволяє автоматично створювати міграції для внесення змін до структури бази даних. Це особливо зручно під час масштабування системи або додавання нового функціоналу.

Переваги Django ORM:

- Спрощення взаємодії з базою даних, що зменшує залежність від конкретного SQL-синтаксису.
- Забезпечення читабельності та підтримуваності коду.
- Зниження ризику помилок при написанні SQL-запитів.
- Можливість легкої зміни структури бази даних через міграції.

2.2. Планувальник завдань

Для автоматизації рутинних задач у системі використовується Django APScheduler, планувальник завдань, який забезпечує виконання операцій за розкладом. Його основна функція — звільнення менеджерів від необхідності вручну виконувати повторювані завдання, такі як оновлення статусів договорів чи надсилання нагадувань клієнтам.

Основні функції Django APScheduler:

1. **Реалізація періодичних задач:** Планувальник дозволяє виконувати завдання з певною періодичністю. Наприклад, щоденне нагадування про необхідність повернення автомобілів:

```python
from apscheduler.schedulers.background import BackgroundScheduler

def send_reminders():
    today = timezone.now().date()
    rentals = Rental.objects.filter(return_date__date=today)
    for rental in rentals:
        # Надсилання нагадування клієнту
        print(f"Нагадування: {rental.customer.email}")
```

Рисунок 2.3 Приклад реалізації функції для автоматичного надсилання нагадувань за допомогою планувальника Django APScheduler

```python
scheduler = BackgroundScheduler()
scheduler.add_job(send_reminders, 'interval', days=1)
scheduler.start()
```

Рисунок 2.4 Налаштування періодичного виконання задачі за допомогою планувальника Django APScheduler

2. **Автоматизація оновлення даних:** Система може автоматично оновлювати статуси договорів, змінювати пробіг автомобілів або скидати параметри, які мають оновлюватися раз на рік (наприклад, позначення клієнтів, які нещодавно святкували день народження).
3. **Інтеграція з Django:** APScheduler легко інтегрується із системою Django, використовуючи механізми модулів, забезпечуючи гнучкість налаштувань.
4. **Журнал виконання задач:** Планувальник веде лог виконання задач, що дозволяє відстежувати успішність або помилки в роботі автоматичних завдань.

Переваги використання планувальника:

- Скорочення часу, витраченого на рутинні задачі.
- Забезпечення своєчасного виконання завдань.
- Гнучкість налаштування періодичності виконання.
- Висока інтеграція з іншими компонентами Django.

Приклади завдань у системі:

- Надсилання нагадувань клієнтам про завершення оренди.
- Скидання статусів клієнтів, які нещодавно святкували день народження (виконується раз на рік).
- Оновлення даних про пробіг автомобілів після завершення оренди.
- Автоматична перевірка актуальності страхових полісів.

Використання Django ORM забезпечує ефективну взаємодію з базою даних через простий і зручний API, що значно спрощує реалізацію CRUD-операцій, валідації та роботу із зв’язками між сутностями. Планувальник завдань Django APScheduler автоматизує повторювані операції, покращуючи продуктивність і зменшуючи ризик пропуску важливих завдань. Ці інструменти забезпечують надійність і гнучкість у роботі з даними, що є критично важливим для CRM-системи прокату автомобілів.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

У цьому розділі буде розглянуто повний цикл розробки CRM-системи для автоматизації процесів управління прокатом автомобілів — від формулювання задач і технічних вимог до проектування та реалізації окремих функціональних модулів.

Основну увагу приділено структурі та логіці бази даних, розробці функціоналу для керування клієнтами, автомобілями, орендами та транзакціями, а також реалізації експорту даних та плануванню задач.

Окремий підрозділ присвячено інтерфейсу користувача — його зручності, структурі та відповідності вимогам до сучасних вебсервісів. Під час реалізації системи були дотримані принципи модульності, розширюваності та безпеки, що дозволяє забезпечити надійну та масштабовану роботу програмного забезпечення.

3.1. Основні функціональні та нефункціональні вимоги

Уявіть систему, яка не просто фіксує дані, а живе життям компанії. Вона зустрічає клієнта раніше за менеджера, знає напам’ять його документи, підказує ціну на оренду залежно від дати й тривалості, автоматично готує договір і ще й попереджає про закінчення страховки.

Саме такою має бути сучасна CRM-система — не пасивним сховищем інформації, а розумним інструментом, який працює поруч з людьми, допомагає приймати рішення, скорочує рутину й підвищує якість сервісу.

Цей підрозділ окреслює, що саме повинна вміти система, щоб стати незамінною частиною щоденної роботи компанії з прокату авто — від управління клієнтами й автопарком до фінансів, звітності та автоматичних нагадувань.

- **Управління клієнтами**
- **Управління автопарком**
- **Облік оренд і розрахунок вартості**
- **Фінансовий облік (транзакції)**
- **Експорт і генерація звітів**
- **Автоматичні нагадування і планувальник**

У той час як функціональні вимоги визначають, що саме має робити система, нефункціональні відповідають на не менш важливе запитання: як вона має це робити.

Саме тут вирішується доля користувацького досвіду, надійності, масштабування та безпеки. Бо можна створити систему, яка вміє все — але якщо вона повільна, небезпечна або складна у використанні, її переваги знецінюються.

Нефункціональні вимоги — це те, що робить хороший продукт блискучим, а зручний сервіс — незамінним.

У цьому підрозділі ми визначимо ключові характеристики, які повинні забезпечити стабільність, швидкодію, захист даних і комфортну взаємодію з користувачем.

Це ті невидимі компоненти, які не відображаються в кнопках, але відчуваються в кожній секунді роботи системи.

- **Зручність інтерфейсу**
- **Безпека доступу**
- **Масштабованість системи**
- **Продуктивність при зростанні даних**

А головне — система розроблена на основі Django ORM, яка дозволяє писати оптимізовані, читабельні та ефективні запити, але у разі потреби — завжди є можливість перейти на «ручне управління» SQL-запитами для точкового прискорення критичних ділянок.

Кожна дія — від відкриття списку клієнтів до експорту в Excel — має бути блискавичною. І байдуже, скільки в системі записів: сто чи сто тисяч. Продуктивність — це відчуття швидкості, яке не зникає навіть тоді, коли зростають дані, документи, навантаження.

Бо продуктивна система — це та, яка працює так, ніби нічого не змінилось. Навіть коли насправді змінилося все.

3.2. Реалізація основних функцій.

Справжня сила CRM-системи розкривається не в теорії, а в дії — у тому, як саме вона полегшує щоденні завдання менеджера, автоматизує рутинні процеси та створює порядок у великій кількості даних [2]. У цьому підрозділі розглянуто реалізацію

ключових функціональних модулів системи: від керування клієнтами й автомобілями до обліку оренд, транзакцій і генерації договорів у Excel. Кожна з функцій була спроєктована з урахуванням зручності, надійності та швидкодії, що робить систему не просто інструментом — а справжнім цифровим партнером в управлінні автопарком.

3.2.1. Реєстрація і управління клієнтами

Клієнт — це серце будь-якого бізнесу з прокату авто. Тому модуль реєстрації та управління клієнтами був розроблений як фундаментальна складова всієї системи, що поєднує зручність, безпеку та точність. У цьому підрозділі розглядається, як реалізовано зберігання особистих і контактних даних, обробку кількох номерів телефону, завантаження документів (паспорт, водійські права) та інтелектуальну валідацію, яка допомагає виявляти дублікати та уникати помилок ще до збереження.

Цей функціонал не просто зберігає інформацію — він створює структурований і надійний клієнтський профіль, готовий до використання в будь-якому бізнес-процесі.

- **Створення, редагування та перегляд клієнтів**

Із першої хвилини взаємодії з новим клієнтом система повинна бути на висоті — швидкою, інтуїтивною та надійною. У розробленій CRM кожна дія — від створення профілю до перегляду його деталей — відбувається у кілька кліків, без затримок, плутанини чи зайвих полів.

Менеджер просто відкриває форму, вводить ім’я, прізвище, email — і клієнт уже зареєстрований у системі. Усе виглядає максимально природно завдяки логічному порядку полів, автоматичним підказкам і миттєвому збереженню. Якщо потрібні зміни — редагування здійснюється у тому ж вікні, без перезавантажень, із візуальним підтвердженням усіх дій.

А перегляд? Це не просто список. Це живий, динамічний профіль клієнта з повною історією взаємодії: оренди, транзакції, завантажені документи. Дизайн інтерфейсу орієнтований на швидке сприйняття — менеджер одразу бачить ключову інформацію, не витрачаючи час на пошук у вкладках чи прихованих блоках.

Усе це реалізовано за допомогою Django Views та шаблонів, у поєднанні з формами, які автоматично підтягують пов'язані дані. Завдяки ORM — жодного дублювання коду, жодної зайвої роботи. Дані зберігаються, обробляються й виводяться на екран так, як цього потребує реальний бізнес-процес [3].

У результаті, менеджер отримує не просто інструмент — а повноцінний робочий простір для ефективної взаємодії з клієнтською базою, який дарує відчуття контролю, швидкості та впевненості в кожному натисканні кнопки.

- **Збереження паспортних даних і водійських посвідчень**

У сфері прокату автомобілів ідентифікація клієнта — не просто формальність, а юридична необхідність. Саме тому в системі реалізовано повноцінний механізм збереження критично важливих документів: паспорта та водійського посвідчення, із фіксацією дати видачі, терміну дії, номеру та сканів файлів.

  - **Модельні поля**

У моделі ***Customer*** створено відповідні поля для кожного з параметрів паспорта та водійського посвідчення:

```
class Customer(models.Model):
    passport_number = models.CharField(max_length=20, blank=False, null=True)
    passport_issue_date = models.DateField(blank=False, null=False)
    passport_expiry_date = models.DateField(blank=True, null=True)
    passport_has_expiry = models.BooleanField(default=False)

    driver_license_number = models.CharField(max_length=20, blank=False, null=True)
    driver_license_issue_date = models.DateField(blank=False, null=False)
    driver_license_expiry_date = models.DateField(blank=True, null=True)
    driver_license_has_expiry = models.BooleanField(default=False)

    first_driver_license_issue_date = models.DateField(blank=False, null=False)
```

Рисунок 3.1 Фрагмент моделі Customer для збереження паспортних та водійських даних клієнта у Django ORM

Кожне з полів було обране з урахуванням юридичної точності та бізнес-вимог: термін дії, окремо для кожного документа, можливість зберігати дату першої видачі водійського посвідчення — особливо актуально у випадку із міжнародним водінням.

  - **Форма з підтримкою календаря**

У формі ***CustomerForm*** для полів, пов'язаних із датами, використано віджет ***DateInput*** із типом ***date***, що дозволяє користувачам вводити дату через зручний календар:

```
class CustomerForm(forms.ModelForm):
    class Meta:
        model = Customer
        fields = ['passport_number', 'passport_issue_date', 'passport_expiry_date',
                  'passport_has_expiry', 'driver_license_number', 'driver_license_issue_date',
                  'driver_license_expiry_date', 'driver_license_has_expiry',
                  'first_driver_license_issue_date']
        widgets = {
            'passport_issue_date': DateInput(format='%Y-%m-%d'),
            'passport_expiry_date': DateInput(format='%Y-%m-%d'),
            'driver_license_issue_date': DateInput(format='%Y-%m-%d'),
            'driver_license_expiry_date': DateInput(format='%Y-%m-%d'),
            'first_driver_license_issue_date': DateInput(format='%Y-%m-%d'),
        }
```

Рисунок 3.2 Форма CustomerForm для введення паспортних і водійських даних користувача

- **Перевірка заповнення — залежно від прапорця**

Система враховує, що деякі документи можуть не мати терміну дії. Тому при збереженні логіка форми враховує стан полів ***passport_has_expiry*** і ***driver_license_has_expiry*** — якщо вони не встановлені, то дата закінчення не є обов'язковою.

```
def clean(self):
    cleaned_data = super().clean()
    if cleaned_data.get('passport_has_expiry') and not cleaned_data.get('passport_expiry_date'):
        self.add_error('passport_expiry_date', "Необхідно вказати дату закінчення паспорта.")

    if cleaned_data.get('driver_license_has_expiry') and not cleaned_data.get('driver_license_exp
        self.add_error('driver_license_expiry_date', "Необхідно вказати дату закінчення водійськ
```

Рисунок 3.3 Перевірка узгодженості дат закінчення дії документів у формі CustomerForm

- **Відображення даних у профілі клієнта**

На сторінці перегляду профілю клієнта всі паспортні та водійські дані відображаються в зрозумілому, структурованому вигляді, з перетворенням дат у

формат *дд.мм.рррр*. Це дозволяє миттєво оцінити статус документів і строк їх дії.

Таким чином, система дозволяє менеджеру швидко і безпомилково занести всі необхідні дані клієнта, зберегти їх у надійній структурі та мати повний контроль над юридичною валідністю документів, що є ключем до легального та безпечного функціонування прокатного бізнесу.

- **Завантаження та зберігання файлів**

У реальному бізнесі електронна копія паспорта, водійського посвідчення чи ІПН має не менше значення, ніж фізичний документ. Саме тому в CRM-системі реалізовано повноцінну підтримку завантаження, зберігання та перегляду важливих клієнтських файлів. Це не просто зручність — це гарантія, що жоден файл не загубиться і завжди буде доступний у кілька кліків.

- **Зберігання файлів у моделі**

У моделі ***Customer*** передбачено окремі поля для збереження кожного документа. Файли завантажуються в унікальні директорії з випадковими іменами, щоб уникнути перезапису:

```python
def get_file_path(instance, filename):
    ext = filename.split('.')[-1]
    filename = f"{uuid.uuid4()}.{ext}"
    return os.path.join('customer_files', filename)

class Customer(models.Model):
    passport_file = models.FileField(upload_to=get_file_path, blank=True, null=True)
    driver_license_file = models.FileField(upload_to=get_file_path, blank=True, null=True)
    rno_kpp_file = models.FileField(upload_to=get_file_path, blank=True, null=True)
```

Рисунок 3.4 Збереження файлів клієнтів у Django-моделі Customer з використанням кастомного шляху

- **Завантаження кількох додаткових файлів**

Окрім основних документів, система підтримує масове завантаження інших файлів — наприклад, довіреностей, контрактів або сканів додаткових прав. Для цього використовується пов’язана модель ***CustomerFile***:

```python
class CustomerFile(models.Model):
    customer = models.ForeignKey(Customer, related_name='files', on_delete=models.CASCADE)
    file = models.FileField(upload_to=get_file_path)
    description = models.CharField(max_length=255, blank=True, null=True)
```

Рисунок 3.5 Модель CustomerFile для збереження додаткових файлів, пов'язаних із клієнтом

- **Форма завантаження у Django**

У шаблоні редагування клієнта файли можна завантажити за допомогою поля ***type="file"***, яке підтримує декілька об'єктів:

```html
<label for="files">Додаткові файли:</label>
<input type="file" name="files" multiple>
```

Рисунок 3.6 HTML-форма для вибору та завантаження кількох додаткових файлів

Вигрузка файлів в ***views.py***:

```python
files = request.FILES.getlist('files')
for file in files:
    CustomerFile.objects.create(customer=customer, file=file)
```

Рисунок 3.7 Збереження кількох файлів клієнта на сервері через request.FILES.getlist()

- **Автоматичне перейменування**

У формі ***CustomerForm*** перед збереженням кожного з основних документів відбувається перейменування файлу, щоб уникнути конфліктів і полегшити менеджмент на сервері:

```python
def get_unique_file_name(self, original_file_name):
    ext = original_file_name.split('.')[-1]
    return f"{uuid.uuid4()}.{ext}"

def save(self, commit=True):
    customer = super().save(commit=False)
    if self.cleaned_data['passport_file']:
        customer.passport_file.name = self.get_unique_file_name(self.cleaned_data['passport_file'].name)
    if commit:
        customer.save()
    return customer
```

Рисунок 3.8 Присвоєння унікального імені файлу паспорта під час збереження об'єкта Customer

- **Перегляд файлів у профілі клієнта**

Файли відображаються в окремому блоці на сторінці перегляду клієнта з можливістю перегляду, завантаження або видалення:

```
{% for f in customer.files.all %}
  <li><a href="{{ f.file.url }}" target="_blank">{{ f.file.name }}</a></li>
{% endfor %}
```

Рисунок 3.9 Виведення списку прикріплених файлів клієнта у шаблоні Django

Таким чином, система дозволяє надійно зберігати документи, уникати повторного завантаження та забезпечує швидкий доступ до всього, що може знадобитися менеджеру чи бухгалтерії. У світі, де швидкий пошук документів = швидке обслуговування, ця функція стає не просто зручною — вона створює професійний стандарт роботи з клієнтом [4].

- **Робота з кількома номерами телефону**

У реальному житті один клієнт — це не один телефон. Це мобільний, робочий, іноді резервний або міжнародний номер. CRM-система має не просто зберігати контакти — вона повинна давати змогу керувати кількома телефонами гнучко, швидко та інтуїтивно. Саме так реалізовано цей функціонал у розробленому застосунку.

- **Модель PhoneNumber: кожен номер — окрема сутність**

Замість додавання кількох полів у таблицю клієнтів, було створено окрему модель ***PhoneNumber***, яка дозволяє зберігати будь-яку кількість номерів для кожного клієнта:

```
class PhoneNumber(models.Model):
    customer = models.ForeignKey(Customer, related_name='phone_numbers', on_delete=models.CASCADE)
    phone = models.CharField(max_length=15, blank=False, null=False)

    def __str__(self):
        return self.phone
```

Рисунок 3.10 Модель PhoneNumber для збереження кількох телефонних номерів клієнта

Цей підхід забезпечує масштабованість, читаємість та повну гнучкість: у майбутньому до кожного номеру можна буде додати тип (наприклад, "основний", "Viber", "WhatsApp") або статус (активний / неактивний).

- **InlineFormSet для зручного керування номерами**

У Django реалізовано керування кількома телефонами через ***inlineformset_factory***. У формі редагування клієнта менеджер бачить кілька полів телефону одночасно й може додавати чи видаляти їх динамічно.

```
PhoneNumberFormSet = inlineformset_factory(
    Customer, PhoneNumber, fields=('phone',), extra=1, can_delete=True
)
```

Рисунок 3.11 Формування inlineformset для редагування телефонних номерів клієнта

У view:

```
formset = PhoneNumberFormSet(request.POST, instance=customer)
if form.is_valid() and formset.is_valid():
    form.save()
    formset.save()
```

Рисунок 3.12 Обробка збереження клієнта та його телефонних номерів через основну форму і formset

У шаблоні:

```
<div id="phone-formset">
  {{ formset.management_form }}
  {% for form in formset %}
    <div class="phone-entry">
      {{ form.phone.label_tag }} {{ form.phone }}
      {% if form.instance.pk %} {{ form.DELETE.label }} {{ form.DELETE }} {% endif %}
    </div>
  {% endfor %}
</div>
```

Рисунок 3.13 Відображення inline formset телефонних номерів у шаблоні Django

- **Виведення телефонів у профілі — у зручному форматі**

Для швидкого перегляду контактів усі номери клієнта виводяться у вигляді списку:

```
def get_phones(self, obj):
    return ", ".join([phone.phone for phone in obj.phone_numbers.all()])
get_phones.short_description = 'Телефони'
```

Рисунок 3.14 Виведення всіх телефонних номерів клієнта в одному полі адміністративної панелі Django

Завдяки такому підходу CRM не просто "знає номер клієнта" — вона адаптується до реального стилю комунікації, підтримує повну історію контактів і дозволяє менеджеру завжди бути на зв’язку. Це не просто технічна зручність — це новий рівень обслуговування та довіри.

- **Валідація унікальності та попередження про дублікати**

У CRM-системі один з найнеприємніших сценаріїв — це коли один клієнт з'являється в базі кілька разів. Такі дублікати порушують статистику, ускладнюють пошук, призводять до помилок у звітах і навіть до подвійного бронювання. Щоб цього уникнути, в системі реалізовано інтелектуальну валідацію унікальності, яка перевіряє не лише email, а й паспортні дані, водійські посвідчення, ІПН та телефонні номери.

- **Валідація полів у формі**

У модулі ***forms.py***, в класі ***CustomerForm***, реалізовано метод ***clean***, який перевіряє унікальність ключових полів, виключаючи поточного клієнта (при редагуванні):

```
def clean(self):
    cleaned_data = super().clean()
    passport_number = cleaned_data.get('passport_number')
    driver_license_number = cleaned_data.get('driver_license_number')
    rno_kpp = cleaned_data.get('rno_kpp')

    instance = self.instance if hasattr(self, 'instance') else None
```

Рисунок 3.15 Отримання даних форми та поточного екземпляра об'єкта в методі clean

- **Пошук на дублікати за паспортом**

```
if passport_number:
    existing = Customer.objects.filter(passport_number=passport_number).exclude(pk=instance.pk if instance else None).first()
    if existing:
        url = f'<a href="/customers/{existing.pk}/">{existing}</a>'
        self.add_error('passport_number', mark_safe(f'<div class="alert alert-danger">Клієнт з таким номером паспорта вже існує: {url}</div>'))
```

Рисунок 3.16 Перевірка унікальності номера паспорта та відображення посилання на існуючого клієнта

- **Аналогічна перевірка для водійського посвідчення та РНОКПП**

```
if driver_license_number:
    existing = Customer.objects.filter(driver_license_number=driver_license_number).exclude(pk=instance.pk if instance else None).first()
    if existing:
        url = f'<a href="/customers/{existing.pk}/">{existing}</a>'
        self.add_error('driver_license_number', mark_safe(f'<div class="alert alert-danger">Клієнт з таким водійським посвідченням вже існує: {url}</div>'))

if rno_kpp:
    existing = Customer.objects.filter(rno_kpp=rno_kpp).exclude(pk=instance.pk if instance else None).first()
    if existing:
        url = f'<a href="/customers/{existing.pk}/">{existing}</a>'
        self.add_error('rno_kpp', mark_safe(f'<div class="alert alert-danger">Клієнт з таким РНОКПП вже існує: {url}</div>'))
```

Рисунок 3.17 Перевірка унікальності водійського посвідчення та РНОКПП з виведенням посилань на існуючих клієнтів

- **Валідація номерів телефону у formset'і**

Для номерів телефону перевірка винесена окремо — на кожен номер у formset'і:

```
for phone_form in self.cleaned_data.get('phone_numbers', []):
    phone = phone_form.cleaned_data.get('phone')
    if phone:
        existing = PhoneNumber.objects.filter(phone=phone).exclude(customer=instance).first()
        if existing:
            url = f'<a href="/customers/{existing.customer.pk}/">{existing.customer}</a>'
            self.add_error(None, mark_safe(f'<div class="alert alert-danger">Клієнт з таким телефоном вже існує: {url}</div>'))
```

Рисунок 3.18 Перевірка унікальності телефонного номера у formset і виведення повідомлення про дублікати

- **Результат — попередження ще до збереження**

Замість того, щоб просто «відхилити» форму, система виводить детальне попередження з посиланням на вже існуючий запис. Це дозволяє менеджеру не лише уникнути дубліката, а й швидко перейти до потрібного клієнта.

Завдяки такій реалізації CRM не просто захищає базу даних від дублювання — вона активно допомагає менеджеру працювати акуратно, свідомо і впевнено. Це не технічна деталь, це — запорука чистої, живої та достовірної інформації, яка служить бізнесу.

3.2.2. Управління автомобілями

Автомобіль у CRM для прокату — це не просто транспортний засіб. Це об'єкт обліку, джерело доходу, носій договорів, пробігу, страхування і навіть фінансових витрат [5]. Саме тому модуль керування автопарком спроєктований так, щоб бути максимально повним, але при цьому зручним для щоденного використання. У цьому підрозділі буде детально описано, як реалізовано реєстрацію нового авто, ведення технічної інформації, завантаження страхових документів, облік пробігу й

формування зручного інтерфейсу для перегляду всіх пов’язаних даних — від контрактів до транзакцій.

- **Реєстрація нового автомобіля**

Уявіть мить: до автопарку щойно прибув новенький автомобіль — його ще пахне салоном, але він уже має потрапити в систему. Менеджер відкриває форму додавання і за лічені хвилини нова одиниця автопарку зареєстрована в CRM, готова до оренди. Саме так працює реалізований модуль реєстрації авто — швидко, зручно та без зайвих кроків.

Модель *Car*: все, що потрібно — і нічого зайвого.

У моделі ***Car*** зберігаються ключові атрибути автомобіля, включно з VIN-кодом, номерним знаком, роком випуску, власником, масами, об’ємами та цінами:

```
class Car(models.Model):
    make = models.CharField(max_length=100)
    model = models.CharField(max_length=100)
    license_plate = models.CharField(max_length=8, validators=[
        RegexValidator(regex=r'^[A-Za-z0-9]{8}$', message="License plate must be exactly 8 characters")
    ])
    year = models.CharField(max_length=4, validators=[
        RegexValidator(regex=r'^[0-9]{4}$', message="Year must be exactly 4 digits")
    ])
    vin = models.CharField(max_length=17, validators=[
        RegexValidator(regex=r'^[A-Za-z0-9]{17}$', message="VIN must be exactly 17 characters")
    ])
    color = models.CharField(max_length=30)
    first_registration_date = models.DateField(null=True, blank=True)
```

Рисунок 3.19 Django-модель Car для збереження даних про автомобіль з валідацією полів

- **Форма створення авто у Django**

Зручна форма створення реалізована через ***CarForm***, де поля виводяться з календарними віджетами:

```
class CarForm(forms.ModelForm):
    class Meta:
        model = Car
        fields = ['make', 'model', 'license_plate', 'year', 'vin', 'color', 'first_registration_date']
        widgets = {
            'first_registration_date': forms.DateInput(attrs={'type': 'date'})
        }
```

Рисунок 3.20 Форма CarForm для введення даних автомобіля з календарним віджетом для дати

- **Обробка запиту у View**

View-функція додає автомобіль у базу і перенаправляє на список:

```
def car_create(request):
    if request.method == 'POST':
        form = CarForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('car_list')
    else:
        form = CarForm()
    return render(request, 'cars/car_form.html', {'form': form})
```

Рисунок 3.21 Обробка створення нового автомобіля через функцію car_create

- **Валідація унікального VIN та номерного знаку**

Щоб уникнути дублювання, у форму додається перевірка унікальності:

```
def clean_vin(self):
    vin = self.cleaned_data['vin']
    if Car.objects.filter(vin=vin).exists():
        raise ValidationError("Автомобіль з таким VIN вже існує.")
    return vin
```

Рисунок 3.22 Валідація унікальності VIN-коду автомобіля у формі

- **Інтерфейс користувача — все під рукою**

Форма створення авто структурована за групами: основна інформація, реєстраційні дані, ціни, страховка. Це забезпечує інтуїтивне заповнення без потреби гортати сторінку чи шукати поля. Після збереження користувач бачить підтвердження, а новий автомобіль одразу доступний для призначення на оренду.

Таким чином, процес реєстрації нового авто у CRM — це не громіздка паперова рутина, а 2 хвилини приємної роботи, після яких бізнес уже готовий почати заробляти з новим автомобілем. Саме такий підхід створює справжню ефективність у сучасному автопрокаті.

- **Збереження технічних характеристик**

У сучасному автопрокатному бізнесі автомобіль — це більше, ніж просто засіб пересування. Це набір характеристик, які визначають його ціну, категорію, доступність для певних умов оренди, а іноді навіть — дозволену вагу багажу або право на виїзд за кордон. Саме тому технічні характеристики в системі реалізовано як

ключову частину моделі автомобіля, доступну для перегляду, аналізу та обліку в кожній операції.

- **Модель Car — розширена технічна карта авто**

До моделі ***Car*** були додані такі технічні параметри:

```python
class Car(models.Model):
    # ... базові поля (make, model, vin, тощо)
    engine_volume = models.PositiveIntegerField(blank=True, null=True, default=None)  # у см³
    gross_weight = models.DecimalField(max_digits=6, decimal_places=2, blank=True, null=True)
    curb_weight = models.DecimalField(max_digits=6, decimal_places=2, blank=True, null=True)
    mileage = models.PositiveIntegerField(null=True, blank=True, validators=[MaxValueValidator(999999)])
```

Рисунок 3.23 Додаткові технічні характеристики автомобіля в моделі Car

Ці поля дозволяють обліковувати такі деталі як:

- Об’єм двигуна — важливо для клієнтів, які бажають потужніший авто.
- Власна та повна маса — для врахування обмежень, наприклад при бронюванні на перевезення вантажів.
- Поточний пробіг — використовується для планування ТО, розрахунку залишкової вартості.

- **Форма введення в CarForm**

Форма редагування автомобіля дозволяє вводити ці параметри в інтерактивний спосіб:

```python
class CarForm(forms.ModelForm):
    class Meta:
        model = Car
        fields = ['engine_volume', 'gross_weight', 'curb_weight', 'mileage']
        widgets = {
            'engine_volume': forms.NumberInput(attrs={'min': 500, 'max': 6000}),
            'gross_weight': forms.NumberInput(attrs={'step': '0.01'}),
            'curb_weight': forms.NumberInput(attrs={'step': '0.01'}),
            'mileage': forms.NumberInput(attrs={'max': 999999}),
        }
```

Рисунок 3.24 Форма CarForm з числовими віджетами для технічних характеристик авто

- **Відображення в адмінці та детальному перегляді**

Для швидкого доступу до технічних параметрів у ***admin.py*** виводяться ключові поля:

```
class CarAdmin(SimpleHistoryAdmin):
    list_display = ['make', 'model', 'year', 'engine_volume', 'gross_weight', 'mileage']
```

Рисунок 3.25 Налаштування виводу полів у списку Django Admin для моделі Car

У шаблоні відображення автомобіля (наприклад, ***car_detail.html***):

```
<ul>
  <li><strong>Об’єм двигуна:</strong> {{ car.engine_volume }} см³</li>
  <li><strong>Повна маса:</strong> {{ car.gross_weight }} кг</li>
  <li><strong>Споряджена маса:</strong> {{ car.curb_weight }} кг</li>
  <li><strong>Пробіг:</strong> {{ car.mileage }} км</li>
</ul>
```

Рисунок 3.26 Виведення технічних характеристик автомобіля у шаблоні Django

- **Бонус: використання технічних характеристик для тарифікації**

Хоча самі характеристики не беруть безпосередньої участі у формуванні вартості, система дозволяє менеджеру швидко порівняти параметри авто при підборі варіанту для VIP-клієнта, вантажного перевезення чи тривалої подорожі.

Таким чином, модуль збереження технічних характеристик — це не просто розширення таблиці. Це база знань про кожен автомобіль, яка дозволяє приймати швидкі, точні й обґрунтовані рішення. А ще — це потужний інструмент для побудови довіри з клієнтами: "Так, ми точно знаємо, що вам потрібно".

- **Завантаження страхових полісів**

У світі прокату авто страхування — це не просто формальність. Це бронежилет для бізнесу. Наявність актуального страхового поліса — обов’язкова умова передача автомобіля клієнту [6]. І саме тому наша CRM-система надає інструмент для завантаження, зберігання і контролю страхових документів з максимальною простотою та надійністю.

- **Поле для зберігання файлів у моделі Car**

Для кожного типу страхування — окреме поле:

```python
class Car(models.Model):
    # ... інші поля
    kasko_insurance_company = models.CharField(max_length=255, blank=True, null=True)
    kasko_policy_number = models.CharField(max_length=255, blank=True, null=True)
    kasko_expiry_date = models.DateField(blank=True, null=True)
    kasko_policy_file = models.FileField(upload_to='policies/kasko/', blank=True, null=True)

    osago_insurance_company = models.CharField(max_length=255, blank=True, null=True)
    osago_policy_number = models.CharField(max_length=255, blank=True, null=True)
    osago_expiry_date = models.DateField(blank=True, null=True)
    osago_policy_file = models.FileField(upload_to='policies/osago/', blank=True, null=True)
```

Рисунок 3.27 Поля для збереження інформації про поліси КАСКО та ОСЦПВ в моделі Car

Файли завантажуються в різні директорії (***/policies/kasko/*** та ***/policies/osago/***), що забезпечує чистоту структури файлів.

- **Відображення у формі Django**

У формі CarForm передбачено відповідні поля з календарем і можливістю вибрати файл:

```python
class CarForm(forms.ModelForm):
    class Meta:
        model = Car
        fields = [
            # ...інші поля
            'kasko_insurance_company', 'kasko_policy_number', 'kasko_expiry_date', 'kasko_policy_file',
            'osago_insurance_company', 'osago_policy_number', 'osago_expiry_date', 'osago_policy_file',
        ]
        widgets = {
            'kasko_expiry_date': forms.DateInput(attrs={'type': 'date'}),
            'osago_expiry_date': forms.DateInput(attrs={'type': 'date'}),
        }
```

Рисунок 3.28 Форма CarForm для введення страхових даних із календарними віджетами для дат

- **Завантаження файлів через інтерфейс**

У шаблоні форми можна завантажити PDF, скан або будь-який файл:

```html
<label for="id_kasko_policy_file">КАСКО поліс:</label>
<input type="file" name="kasko_policy_file" id="id_kasko_policy_file">

<label for="id_osago_policy_file">ОСАГО поліс:</label>
<input type="file" name="osago_policy_file" id="id_osago_policy_file">
```

Рисунок 3.29 HTML-форма для завантаження файлів полісів КАСКО та ОСЦПВ

- **Інформативне відображення в деталях авто**

На сторінці перегляду автомобіля у вигляді списку з'являється:

```
<p><strong>КАСКО ({{ car.kasko_insurance_company }}) до {{ car.kasko_expiry_date }}:</strong>
    {% if car.kasko_policy_file %}
        <a href="{{ car.kasko_policy_file.url }}" target="_blank">📄 Переглянути</a>
    {% else %}
        <span class="text-danger">Не завантажено</span>
    {% endif %}
</p>
```

Рисунок 3.30 Відображення інформації про поліс КАСКО з перевіркою наявності файлу

- **Переваги: нагадування, безпека і прозорість**
  - Менеджер бачить, до якої дати діє поліс.
  - Система дозволяє зберігати PDF-файли на сервері — без потреби в сторонніх сервісах.
  - Доступ до полісу можна отримати у будь-який момент через один клік.

Таким чином, функціональність завантаження страхових полісів — це не просто зручність. Це інформаційна броня для бізнесу, яка забезпечує легітимність кожної операції, скорочує ризики та допомагає миттєво реагувати на закінчення терміну дії страхування [7].

- **Облік пробігу та історії його змін**

Пробіг — це пульс кожного автомобіля. Він свідчить про ступінь зношення, впливає на графік техобслуговування, визначає залишкову вартість і навіть... довіру клієнта. Саме тому в системі реалізований автоматичний і ручний облік пробігу, доповнений механізмом збереження історії змін — усе для прозорості та контролю.

- **Поле пробігу в моделі Car**

Актуальний пробіг зберігається безпосередньо в моделі автомобіля:

```
class Car(models.Model):
    # ...
    mileage = models.PositiveIntegerField(null=True, blank=True, validators=[MaxValueValidator(999999)])
```

Рисунок 3.31 Поле mileage моделі Car з валідацією максимально допустимого пробігу

- **Історія змін — окрема модель MileageHistory**

Щоразу, коли пробіг змінюється, нове значення записується в спеціальну таблицю історії:

```
class MileageHistory(models.Model):
    car = models.ForeignKey(Car, related_name='mileage_history', on_delete=models.CASCADE)
    mileage = models.PositiveIntegerField()
    date_recorded = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"{self.car} – {self.mileage} км на {self.date_recorded:%d.%m.%Y %H:%M}"
```

Рисунок 3.32 Модель MileageHistory для фіксації історії пробігу автомобіля

- **Автоматичне збереження історії при зміні пробігу**

У методі ***save()*** моделі Car реалізована перевірка:

```
def save(self, *args, **kwargs):
    if self.pk:
        original = Car.objects.get(pk=self.pk)
        if original.mileage != self.mileage:
            MileageHistory.objects.create(car=self, mileage=self.mileage)
    super(Car, self).save(*args, **kwargs)
```

Рисунок 3.33 Перевизначення методу save для автоматичного створення запису в історії пробігу

Завдяки цьому механізму історія пробігу формується автоматично, без додаткових дій від менеджера.

- **Зміна пробігу під час оренди**

При створенні або оновленні оренди ***(Rental)***, пробіг перед і після повернення зчитується, і оновлює дані:

```
def rental_update(request, pk):
    rental = get_object_or_404(Rental, pk=pk)
    # ...
    if form.is_valid():
        rental = form.save(commit=False)
        car = rental.car
        car.update_mileage(rental.mileage_before, rental.mileage_after)
```

Рисунок 3.34 Оновлення даних про оренду з викликом методу оновлення пробігу автомобіля

Метод ***update_mileage*** обробляє зміни:

```
def update_mileage(self, mileage_before, mileage_after):
    if mileage_after:
        self.mileage = mileage_after
    elif mileage_before and (self.mileage is None or self.mileage < mileage_before):
        self.mileage = mileage_before
    self.save()
```

Рисунок 3.35 Метод update_mileage для оновлення пробігу автомобіля за даними оренди

- **Вивід історії пробігу на сторінці авто**

У шаблоні ***car_detail.html*** менеджер може побачити всі зміни пробігу:

```
<h4>Історія змін пробігу:</h4>
<ul>
  {% for entry in car.mileage_history.all %}
    <li>{{ entry.date_recorded|date:"d.m.Y H:i" }} – {{ entry.mileage }} км</li>
  {% empty %}
    <li>Даних немає</li>
  {% endfor %}
</ul>
```

Рисунок 3.36 Виведення історії змін пробігу автомобіля у шаблоні Django

Переваги:

- Всі зміни — збережені та прозорі.
- Система автоматично записує пробіг при кожній зміні.
- Жоден маніпулятивний «відкат» пробігу не пройде непоміченим.

Завдяки цьому підходу CRM-система не просто зберігає дані — вона захищає репутацію компанії, довіру клієнтів і технічну справність автопарку.

- **Виведення пов’язаної інформації у зручному вигляді**

Автомобіль — це не ізольована сутність. Він пов’язаний з орендами, клієнтами, транзакціями, страховими полісами, пробігом та навіть фінансовими звітами. Щоб не перетворити управління автопарком на безкінечне "копання" в таблицях, наша система виводить усю пов’язану інформацію в одному зручному інтерфейсі, на сторінці кожного автомобіля.

- **Зв'язки в моделі Car**

Модель ***Car*** має численні зв'язки з іншими моделями:

```python
class Car(models.Model):
    # ...
    mileage_history = models.ManyToOneRel(MileageHistory, field_name='car')
    transactions = models.ManyToOneRel('Transaction', field_name='car')
    # Rental пов'язується через поле в моделі Rental
```

Рисунок 3.37 Явно визначені зворотні зв'язки ManyToOneRel у моделі Car

- **View-функція — збір пов'язаної інформації**

У ***view car_detail*** ми отримуємо все потрібне за один запит:

```python
def car_detail(request, pk):
    car = get_object_or_404(Car, pk=pk)
    rentals = Rental.objects.filter(car=car)
    transactions = car.transactions.all()
    mileage_history = car.mileage_history.all()
    return render(request, 'cars/car_detail.html', {
        'car': car,
        'rentals': rentals,
        'transactions': transactions,
        'mileage_history': mileage_history,
    })
```

Рисунок 3.38 Отримання детальної інформації про автомобіль у функції car_detail

- **Вивід у шаблоні — все під рукою**

У шаблоні ***car_detail.html*** кожен розділ оформлено окремим блоком:

```
<h3>Оренди</h3>
<ul>
  {% for rental in rentals %}
    <li>{{ rental.rental_date|date:"d.m.Y" }} – {{ rental.customer }}</li>
  {% empty %}
    <li>Цей автомобіль ще не використовувався в оренді</li>
  {% endfor %}
</ul>

<h3>Фінансові транзакції</h3>
<ul>
  {% for transaction in transactions %}
    <li>{{ transaction.date|date:"d.m.Y" }}: {{ transaction.amount }} грн – {{ transaction.category }}</li>
  {% empty %}
    <li>Транзакції не знайдено</li>
  {% endfor %}
</ul>
```

Рисунок 3.39 Виведення оренд та фінансових транзакцій автомобіля у шаблоні Django

- **Візуальна зручність: структура і навігація**
  - Всі пов'язані блоки логічно згруповані — спочатку оренди, потім фінанси, потім технічна історія.

- Створені якірні посилання (***#rentals***, ***#transactions***, ***#mileage***) для швидкого переходу до розділів.
- За потреби інформація експортується одним кліком — наприклад, усі оренди цього авто → Excel.

Переваги:

- Менеджер бачить повну картину по автомобілю — від історії використання до витрат.
- Жодного дублювання даних: усе тягнеться з бази в реальному часі.
- Кожна одиниця автопарку — як на долоні.

Таким чином, ми реалізували інтелектуальний огляд автомобіля, який не потребує "клікати в десять меню", а дозволяє приймати рішення швидко, впевнено і на основі фактів.

3.2.3. Система обліку оренди

Оренда — це серце автопрокатного бізнесу. Саме тут відбувається головна взаємодія між клієнтом, автомобілем і прибутком. Саме тому система обліку оренди в нашому програмному забезпеченні розроблена з особливою увагою до деталей, точності та зручності [8]. Вона дозволяє менеджеру буквально за хвилини сформувати договір, обрати клієнта й авто, врахувати всі додаткові послуги та одразу побачити остаточну вартість оренди.

У цьому підрозділі розглянуто ключові елементи реалізації обліку оренд: від інтуїтивного створення нової угоди до автоматичного розрахунку сум, ведення історії пробігу та збереження всіх супутніх даних, що робить систему не просто обліковою — а по-справжньому розумною.

- **Створення договору оренди з вибором авто та клієнта**

Миттєве бронювання. Мінімум кроків. Максимум контролю. Саме так виглядає створення договору оренди в нашій системі. Завдяки зручній формі, менеджер за лічені хвилини може обрати потрібного клієнта, призначити автомобіль і одразу побачити всі деталі майбутньої угоди — без зайвого хаосу і переключень між вкладками.

- **Модель Rental — центр логіки оренди**

```python
class Rental(models.Model):
    enterprise = models.CharField(max_length=20, choices=ENTERPRISE_CHOICES, default='TOMKIV')
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
    car = models.ForeignKey(Car, on_delete=models.CASCADE)
    rental_date = models.DateTimeField(default=timezone.now)
    return_date = models.DateTimeField(null=True, blank=True)
    pickup_location = models.CharField(max_length=255, null=True, blank=True)
    dropoff_location = models.CharField(max_length=255, null=True, blank=True)
    rate = models.DecimalField(max_digits=10, decimal_places=2)
    # ... інші поля
```

Рисунок 3.40 Django-модель Rental для збереження інформації про оренду автомобіля

- **Форма RentalForm — інтуїтивний інтерфейс для створення оренди**

```python
class RentalForm(forms.ModelForm):
    customer = forms.ModelChoiceField(queryset=Customer.objects.all(), label='Клієнт')
    car = forms.ModelChoiceField(queryset=Car.objects.all(), label='Автомобіль')
    rental_date = forms.DateTimeField(widget=DateTimeInput(), label='Дата видачі')
    return_date = forms.DateTimeField(widget=DateTimeInput(), label='Дата повернення')

    class Meta:
        model = Rental
        fields = ['enterprise', 'customer', 'car', 'rental_date', 'return_date', 'pickup_location', 'dropoff_location', 'rate']
```

Рисунок 3.41 Форма RentalForm для введення інформації про оренду автомобіля

- **View-функція rental_create — логіка вибору та збереження**

```python
@login_required
def rental_create(request):
    if request.method == 'POST':
        form = RentalForm(request.POST)
        if form.is_valid():
            rental = form.save()
            # Додаткові обчислення, збереження пробігу, інші дані
            return redirect('rental_list')
    else:
        form = RentalForm()
    return render(request, 'rentals/rental_form.html', {'form': form})
```

Рисунок 3.42 Обробка створення нового запису про оренду у функції rental_create

- **Інтерфейс: форма вибору без перевантаження**

У шаблоні ***rental_form.html*** передбачено просту і водночас інформативну структуру:

```
<form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit" class="btn btn-primary">Створити оренду</button>
</form>
```

Рисунок 3.43 HTML-шаблон для створення оренди з використанням форми Django

Поля "Клієнт" та "Автомобіль" представлені випадаючими списками з пошуком, що значно прискорює вибір.

Автоматизація і гнучкість:

- Система підтягує всі доступні авто, з можливістю відфільтрувати зайняті.
- Після вибору клієнта — автоматично показується його історія, якщо вона є.
- Дата повернення може бути не вказана одразу, що дозволяє створювати відкриті угоди.

Таким чином, процес створення договору в нашій CRM — це не просто заповнення форми. Це інтелектуальна взаємодія між людьми та даними, яка перетворює рутинну операцію в швидкий і приємний процес.

- **Автоматичний розрахунок кількості днів і загальної вартості**

Час — це гроші. І система, що орієнтована на реальний бізнес, має це розуміти. Саме тому при створенні чи редагуванні оренди наша CRM автоматично підраховує тривалість оренди, вартість за курсом та враховує додаткові опції — миттєво, без калькулятора і сторонніх Excel.

- **Модель Rental: вся логіка в одному місці**

Методи в моделі Rental відповідають за дні, євро та гривні:

```
class Rental(models.Model):
    rental_date = models.DateTimeField(default=timezone.now)
    return_date = models.DateTimeField(null=True, blank=True)
    rate = models.DecimalField(max_digits=10, decimal_places=2)
    currency_rate = models.DecimalField(max_digits=11, decimal_places=2, null=True, blank=True)
    days = models.IntegerField(null=True, blank=True)
    total_rent_eur = models.DecimalField(max_digits=10, decimal_places=2, null=True, blank=True)
    total_rent_uah = models.DecimalField(max_digits=10, decimal_places=2, null=True, blank=True)

    def calculate_days(self):
        if self.return_date and self.rental_date:
            return (self.return_date - self.rental_date).days + 1
        return 0

    def calculate_total_rent_eur(self):
        total = self.calculate_days() * self.rate
        return total

    def calculate_total_rent_uah(self):
        return round(self.calculate_total_rent_eur() * self.currency_rate, 2) if self.currency_rate else 0
```

Рисунок 3.44 Розрахунок кількості днів та суми оренди у моделі Rental

- **Автоматичне збереження при save()**

```
def save(self, *args, **kwargs):
    self.days = self.calculate_days()
    self.total_rent_eur = self.calculate_total_rent_eur()
    self.total_rent_uah = self.calculate_total_rent_uah()
    super().save(*args, **kwargs)
```

Рисунок 3.45 Перевизначення методу save у моделі Rental для автоматичних розрахунків

Усі обчислення оновлюються щоразу, коли змінюються дати або тариф.

- **У формі: поля вартості лише для читання**

```
class RentalForm(forms.ModelForm):
    days = forms.IntegerField(widget=forms.TextInput(attrs={'readonly': 'readonly'}), required=False)
    total_rent_eur = forms.DecimalField(widget=forms.TextInput(attrs={'readonly': 'readonly'}), required=False)
    total_rent_uah = forms.DecimalField(widget=forms.TextInput(attrs={'readonly': 'readonly'}), required=False)
```

Рисунок 3.46 Поля лише для читання у формі RentalForm для виведення розрахованих значень

- **Відображення у шаблоні: зрозуміло та миттєво**

```
<p>Дні оренди: <strong>{{ form.days.value }}</strong></p>
<p>Сума (€): <strong>{{ form.total_rent_eur.value }}</strong></p>
<p>Сума (₴): <strong>{{ form.total_rent_uah.value }}</strong></p>
```

Рисунок 3.47 Виведення розрахованих полів тривалості та вартості оренди у шаблоні

Результат:

- Система підраховує кількість днів автоматично, навіть якщо менеджер забув це зробити.
- Курс гривні до євро — враховується в розрахунках для точного фінансового обліку.
- Менеджер бачить результат до збереження — і одразу знає, що погоджує з клієнтом.

Завдяки цьому функціоналу CRM-система стає не просто «записником угод», а розумним помічником, що не допускає помилок у розрахунках і знімає з менеджера рутину.

- **Облік додаткових послуг (GPS, дитяче крісло, виїзд за кордон тощо)**

Додаткові послуги — не просто спосіб покращити комфорт клієнта, а й джерело додаткового прибутку для компанії. Тому система повинна вміти гнучко враховувати кожну опцію, автоматично розраховувати її вартість та включати у загальний підсумок оренди.

- **Поля в моделі Rental**

```python
class Rental(models.Model):
    add_child_seat = models.BooleanField(default=False)
    add_wifi_router = models.BooleanField(default=False)
    add_gps_navigator = models.BooleanField(default=False)
    add_electric_scooter = models.BooleanField(default=False)
    add_cross_border = models.BooleanField(default=False)
    add_additional_insurance = models.BooleanField(default=False)
```

Рисунок 3.48 Додаткові опції в моделі Rental для обліку додаткових послуг

- **Розрахунок у calculate_total_rent_eur()**

```python
def calculate_total_rent_eur(self):
    total = self.calculate_days() * self.rate

    if self.add_child_seat:
        total += min(5 * self.days, 40)
    if self.add_wifi_router:
        total += min(3 * self.days, 40)
    if self.add_gps_navigator:
        total += min(5 * self.days, 40)
    if self.add_electric_scooter:
        total += 10 * self.days
    if self.add_cross_border:
        total += 100

    return total
```

Рисунок 3.49 Розширений метод calculate_total_rent_eur з урахуванням додаткових послуг

Вартість кожної опції розраховується залежно від тривалості, з обмеженням на максимум.

- **Автоматичне оновлення при save()**

```python
def save(self, *args, **kwargs):
    self.total_rent_eur = self.calculate_total_rent_eur()
    self.total_rent_uah = self.calculate_total_rent_uah()
    super().save(*args, **kwargs)
```

Рисунок 3.50 Збереження обчислених сум оренди у методі save моделі Rental

- **Виведення на сторінці угоди**

```
{% if rental.add_gps_navigator %}<li>GPS-навігатор</li>{% endif %}
{% if rental.add_child_seat %}<li>Дитяче крісло</li>{% endif %}
{% if rental.add_cross_border %}<li>Виїзд за кордон</li>{% endif %}
```

Рисунок 3.51 Виведення обраних додаткових послуг у шаблоні оренди

Результат:

- Кожна додаткова опція автоматично враховується у фінальній сумі.
- Вартість завжди прозора для менеджера та клієнта.
- Логіка легко масштабована — нові опції додаються у кілька рядків.

- **Збереження пробігу до та після**

Пробіг — це не просто цифра. Це гарантія, що авто повернули в належному стані. Це основа для розрахунку амортизації, контролю зносу і навіть вирішення конфліктних ситуацій. У нашій системі облік пробігу реалізований так, щоб його неможливо було забути чи проігнорувати.

- **Поля в моделі Rental**

```
class Rental(models.Model):
    mileage_before = models.PositiveIntegerField(null=True, blank=True)
    mileage_after = models.PositiveIntegerField(null=True, blank=True)
```

Рисунок 3.52 Поля для фіксації пробігу до та після оренди в моделі Rental

- **Оновлення пробігу автомобіля**

У методі save() оренди, ми оновлюємо фактичний пробіг авто:

```
def save(self, *args, **kwargs):
    super().save(*args, **kwargs)
    self.car.update_mileage(self.mileage_before, self.mileage_after)
```

Рисунок 3.53 Оновлення пробігу автомобіля після збереження оренди

А в моделі Car:

```
def update_mileage(self, mileage_before, mileage_after):
    if mileage_after:
        self.mileage = mileage_after
    elif mileage_before and (self.mileage is None or self.mileage < mileage_before):
        self.mileage = mileage_before
    self.save()
```

Рисунок 3.54 Метод update_mileage для оновлення загального пробігу автомобіля

- **Історія змін пробігу**

Для збереження історії:

```
class MileageHistory(models.Model):
    car = models.ForeignKey(Car, related_name='mileage_history', on_delete=models.CASCADE)
    mileage = models.PositiveIntegerField()
    date_recorded = models.DateTimeField(auto_now_add=True)
```

Рисунок 3.55 Модель MileageHistory для зберігання історії пробігу автомобіля

І при зміні:

```
if rental.mileage_before != previous_mileage_before:
    MileageHistory.objects.create(car=car, mileage=rental.mileage_before)
if rental.mileage_after != previous_mileage_after:
    MileageHistory.objects.create(car=car, mileage=rental.mileage_after)
```

Рисунок 3.56 Створення записів історії пробігу при зміні значень

Результат:

- Наявність пробігу — обов'язкова і контролюється формою.
- Історія змін зберігається, що дає змогу відслідковувати зношення авто.
- Менеджер бачить фактичний пробіг без ручних обчислень — система робить це за нього.

- **Відображення історії оренд**

Кожен автомобіль має свою історію — з ким, коли, на скільки днів і за яку суму він їздив. У нашій CRM-системі ця історія не губиться в хаосі таблиць, а відображається наочно, структуровано й миттєво доступна менеджеру. Такий підхід не лише підвищує ефективність, а й створює новий рівень прозорості й довіри в роботі з автопарком.

- **Оренди, пов'язані з авто або клієнтом**

Зв'язок реалізовано через поле ForeignKey у моделі Rental:

```
class Rental(models.Model):
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
    car = models.ForeignKey(Car, on_delete=models.CASCADE)
```

Рисунок 3.57 Зв'язок між моделями Rental, Customer та Car

- **Отримання історії оренд в views.py**

Наприклад, для сторінки клієнта або автомобіля:

```python
def customer_detail(request, pk):
    customer = get_object_or_404(Customer, pk=pk)
    rentals = Rental.objects.filter(customer=customer).order_by('-rental_date')
    return render(request, 'customers/customer_detail.html', {
        'customer': customer,
        'rentals': rentals
    })
```

Рисунок 3.58 Відображення інформації про клієнта та його оренди

- **Фільтрація, пагінація і швидкий пошук**

Система підтримує:

- Сортування за датою.
- Пошук за іменем клієнта або номером авто.
- Можливість експортувати історію в Excel (див. 3.3.4).

Результат:

- Історія оренд виводиться миттєво, без необхідності вручну шукати по БД.
- Менеджер бачить усі деталі — дати, авто, сума, пробіг, статус.
- Візуалізація дозволяє швидко оцінити лояльність клієнта чи ефективність авто.

Цей функціонал завершує комплексну реалізацію обліку оренди.

3.2.4. Експорт даних у форматі Excel

Сучасний автопрокат — це не лише цифровий облік, але й документи, які потрібно передати клієнту в паперовому або електронному вигляді [9]. Щоб зекономити час менеджера і зменшити кількість помилок, наша CRM-система дозволяє одним кліком сформувати повноцінний договір оренди у форматі Excel.

Цей підрозділ описує, як реалізовано експорт даних: від заповнення шаблону до підрахунку вартості, курсу, додаткових послуг та підтримки декількох типів договорів — залежно від обраного підприємства.

- **Генерація Excel-договору з шаблону**

Інтелектуальний договір в один клік — саме так виглядає процес генерації Excel-документа в нашій CRM. Менеджер натискає кнопку — система бере шаблон, заповнює всі дані автоматично і формує договір, готовий для друку або надсилання клієнту.

- **Шаблони договорів: основа гнучкості**

У директорії ***templates/*** зберігаються файли ***.xlsx*** для кожного з підприємств:

```
templates/
├── FOP SALBAY.xlsx
├── TOV DYPLOM.xlsx
└── FOP KLEBAN.xlsx
```

Рисунок 3.59 Приклад структури директорії з шаблонами Excel-файлів

- **Вибір шаблону залежно від підприємства**

У в'юсі ***export_rental_to_excel***:

```
template_mapping = {
    'FOP SALABAY': 'SALABAY.xlsx',
    'TOV DYPLOM': 'DYPLOM.xlsx',
    'FOP KLEBAN': 'KLEBAN.xlsx'
}
template_path = os.path.join(settings.BASE_DIR, 'templates', template_mapping[rental.enterprise])
wb = openpyxl.load_workbook(template_path)
ws = wb.active
```

Рисунок 3.60 Вибір та відкриття відповідного шаблону Excel залежно від підприємства

Один рядок — і система сама знає, який файл відкривати.

- **Виклик експорту через URL**

У ***urls.py***:

```
path('rental/<int:pk>/export/', views.export_rental_to_excel, name='export_rental_to_excel'),
```

Рисунок 3.61 Шлях для експорту даних оренди у форматі Excel

Менеджер бачить кнопку "Завантажити договір" на сторінці угоди. Достатньо натиснути — і Excel-договір згенерований.

- **Готовий файл завантажується одразу**

```
response = HttpResponse(content_type='application/vnd.openxmlformats-officedocument.spreadsheetml.sheet')
response['Content-Disposition'] = f'attachment; filename=rental_{pk}.xlsx'
wb.save(response)
return response
```

Рисунок 3.62 Формування відповіді для завантаження Excel-файлу з орендою

Без збереження на сервері. Без тимчасових файлів. Миттєвий результат.

Результат:

- Договір формується за лічені секунди, без ручного введення.
- Вибір шаблону залежить від обраного підприємства, що критично для юридичної точності.

- Менеджер отримує професійно оформлений документ, який відповідає бренду компанії.

- **Автоматичне заповнення клієнтських та автомобільних даних**

Кожен договір оренди починається з імені клієнта та закінчується технічними деталями авто. Помилка в одному полі — і документ може стати недійсним. Саме тому наша CRM не просто експортує дані, а розумно й безпомилково переносить всю важливу інформацію з бази в договір, використовуючи OpenPyXL.

- **Заповнення даних клієнта у шаблоні**

```
ws['B9'] = f'{customer.last_name} {customer.first_name} {customer.middle_name or ""}'.upper()
ws['B16'] = customer.phone_numbers.first().phone if customer.phone_numbers.exists() else ''
ws['B17'] = customer.email
ws['B18'] = customer.birth_date.strftime('%d.%m.%Y')
ws['B19'] = customer.passport_number
ws['B20'] = customer.passport_issue_date.strftime('%d.%m.%Y')
ws['B21'] = customer.driver_license_number
ws['B22'] = customer.driver_license_issue_date.strftime('%d.%m.%Y')
ws['B23'] = customer.rno_kpp or ''
```

Рисунок 3.63 Заповнення Excel-шаблону персональними даними клієнта

Система перевіряє наявність кожного поля, уникаючи помилок при відсутності даних.

- **Заповнення даних автомобіля**

```
ws['D17'] = f'{car.make} {car.model}'.upper()
ws['D18'] = car.license_plate
ws['D20'] = rental.rental_date.strftime('%d.%m.%Y')
ws['D21'] = rental.rental_date.strftime('%H:%M')
```

Рисунок 3.64 Заповнення Excel-шаблону інформацією про автомобіль та дату оренди

- **Додатковий водій (якщо є)**

```
if additional_driver:
    ws['B24'] = f'{additional_driver.last_name} {additional_driver.first_name} {additional_driver.middle_name or ""}'.upper()
    ws['B25'] = additional_driver.birth_date.strftime('%d.%m.%Y')
    ws['B26'] = additional_driver.passport_number
    ws['B27'] = additional_driver.passport_issue_date.strftime('%d.%m.%Y')
    ws['B28'] = additional_driver.driver_license_number
    ws['B29'] = additional_driver.driver_license_issue_date.strftime('%d.%m.%Y')
    ws['B30'] = additional_driver.rno_kpp
```

Рисунок 3.65 Виведення даних додаткового водія у шаблон Excel

Результат:

- Усі поля договору автоматично заповнюються актуальними даними з бази.

- Зводиться до нуля ризик помилок або пропусків.
- Навіть за складних умов — наприклад, відсутність номера РНОКПП або другого водія — система адаптується без збоїв.

- **Формування сум, курсів, дат та обраних опцій**

Що перетворює звичайний договір у професійний фінансовий документ? Правильно: точні суми, актуальні курси валют, коректні дати та список додаткових послуг. Наше рішення автоматично виводить усю цю інформацію, виключаючи ручні розрахунки та людський фактор.

- **Фінансові поля: євро, курс, гривня**

```
ws['J27'] = rental.total_rent_eur          # Сума оренди в євро
ws['J28'] = rental.currency_rate           # Курс євро до гривні
ws['J29'] = rental.total_rent_uah          # Сума оренди в гривнях
```

Рисунок 3.66 Запис вартості оренди та валютного курсу в Excel-шаблон

Значення беруться з моделі ***Rental***, яка автоматично розраховує суму оренди на основі кількості днів, тарифу та обраних послуг.

- **Дати початку та завершення оренди**

```
ws['D20'] = rental.rental_date.strftime('%d.%m.%Y')
ws['D21'] = rental.rental_date.strftime('%H:%M')
ws['D13'] = rental.return_date.strftime('%d.%m.%Y %H:%M') if rental.return_date else ''
```

Рисунок 3.67 Запис дати та часу видачі й повернення автомобіля в Excel-шаблон

Часова зона враховується: всі дати переводяться в київський час через ***pytz***.

- **Обрані опції — дитяче крісло, GPS, Wi-Fi тощо**

```
if rental.add_child_seat:
    ws['J18'] = min(5 * rental.days, 40)
if rental.add_wifi_router:
    ws['J15'] = min(3 * rental.days, 40)
if rental.add_gps_navigator:
    ws['J19'] = min(5 * rental.days, 40)
if rental.add_electric_scooter:
    ws['J16'] = 10 * rental.days
if rental.add_cross_border:
    ws['J22'] = 100
if rental.add_driver_fee:
    ws['J23'] = 10
```

Рисунок 3.68 Заповнення додаткових витрат в Excel-шаблоні оренди залежно від обраних опцій

- **Особливий випадок — повне страхування**

```
if rental.add_additional_insurance:
    ws['J31'] = car.fuel_deposit
    ws['D28'] = 0  # депозит не береться
    ws['D29'] = 0
    ws['J17'] = rental.calculate_insurance_cost()
```

Рисунок 3.69 Заповнення полів Excel-шаблону при виборі додаткового страхування

Результат:

- Сума оренди розраховується точно й прозоро, враховуючи всі параметри.
- Дата і час — у локальному форматі, без збоїв.
- Опції — позначені в документі й включені у фінальний підсумок.

- **Підтримка різних шаблонів для різних підприємств**

У сфері автопрокату один формат договору — це занадто мало. Різні юридичні особи, різні логотипи, реквізити, шаблони. Наше програмне забезпечення враховує цю реальність і дозволяє формувати Excel-документи на основі шаблону, що відповідає обраному підприємству.

- **Поле в моделі Rental**

```
class Rental(models.Model):
    ENTERPRISE_CHOICES = [
        ('SALABAY', 'ФОП Салабай'),
        ('DYPLOM', 'ТОВ Диплом'),
        ('KLEBAN', 'ФОП Клебан')
    ]
    enterprise = models.CharField(max_length=20, choices=ENTERPRISE_CHOICES, default='SALABAY')
```

Рисунок 3.70 Перелік підприємств для вибору у моделі оренди

Це поле заповнюється при створенні договору.

Кожен файл — це унікальний шаблон договору, адаптований під візуальні й юридичні вимоги підприємства.

- **Автоматичний вибір шаблону**

У в’юсі ***export_rental_to_excel***:

```
template_mapping = {
    'SALABAY': 'SALABAY.xlsx',
    'DYPLOM': 'DYPLOM.xlsx',
    'KLEBAN': 'KLEBAN.xlsx'
}
template_path = os.path.join(settings.BASE_DIR, 'templates', template_mapping[rental.enterprise])
wb = openpyxl.load_workbook(template_path)
ws = wb.active
```

Рисунок 3.71 Вибір Excel-шаблону на основі підприємства

Все, що потрібно — це вибрати підприємство під час створення оренди. Далі система зробить усе сама.

Результат:

- Один інструмент — три шаблони, кожен із власними реквізитами.
- Жодної плутанини чи необхідності вручну підставляти інформацію.
- Автоматизація, яка адаптується до структури вашої компанії.

3.2.5. Управління транзакціями

Жодна CRM-система не буде повноцінною без чіткого обліку фінансів. Саме транзакції — це пульс бізнесу, який відображає прибутковість, витрати, рентабельність кожного автомобіля чи договору. У нашій системі реалізовано зручний модуль управління фінансовими потоками, який дозволяє не просто фіксувати платежі, а аналізувати структуру доходів і витрат, фільтрувати їх за ключовими параметрами та пов’язувати з конкретними клієнтами, авто або угодами.

У цьому підрозділі розглянемо, як реалізоване:

- внесення фінансових записів (як доходів, так і витрат),
- гнучка прив’язка до сутностей системи,
- інтелектуальна фільтрація транзакцій,
- формування статистичних зведень.

- **Внесення доходів і витрат за орендою або авто**

Управління фінансами — це не просто бухгалтерія, а спосіб контролювати бізнес у реальному часі. У нашій системі транзакції — це окремі об’єкти, які можуть бути пов’язані з клієнтом, автомобілем чи конкретною орендою. Такий підхід дозволяє точно бачити, звідки приходять гроші і куди вони витрачаються — від оплати послуг до технічного обслуговування авто.

- **Модель Transaction**

```python
class Transaction(models.Model):
    TRANSACTION_TYPES = [
        ('income', 'Доходи'),
        ('expense', 'Витрати'),
    ]

    car = models.ForeignKey(Car, related_name='transactions', on_delete=models.CASCADE, blank=True, null=True)
    rental = models.ForeignKey(Rental, related_name='transactions', on_delete=models.CASCADE, blank=True, null=True)
    client = models.ForeignKey(Customer, related_name='transactions', on_delete=models.CASCADE, blank=True, null=True)

    date = models.DateField()
    amount = models.DecimalField(max_digits=10, decimal_places=2)
    type = models.CharField(max_length=7, choices=TRANSACTION_TYPES)
    category = models.ForeignKey(TransactionCategory, on_delete=models.CASCADE)
    description = models.TextField(blank=True, null=True)
```

Рисунок 3.72 Модель Transaction для збереження доходів і витрат

- **Приклад створення транзакції в коді**

```python
Transaction.objects.create(
    car=car,
    rental=rental,
    client=customer,
    date=timezone.now().date(),
    amount=500.00,
    type='income',
    category=some_category,
    description='Оплата за оренду авто Audi A4'
)
```

Рисунок 3.73 Створення запису про дохідну транзакцію

Один рядок — і транзакція одразу з'являється у фінансовій аналітиці, прив'язана до всіх ключових об'єктів.

- **Адмін-інтерфейс для ручного додавання**

У ***admin.py***:

```python
@admin.register(Transaction)
class TransactionAdmin(admin.ModelAdmin):
    list_display = ('date', 'amount', 'type', 'category', 'car', 'rental', 'client')
    list_filter = ('type', 'category', 'date')
    search_fields = ('description',)
```

Рисунок 3.74 Реєстрація моделі Transaction в Django Admin

Результат:

- Витрати та доходи прив'язані до відповідних об'єктів.
- Можливість створення вручну чи автоматично при створенні оренди.
- Бухгалтерський контроль без Excel — усе в системі.

- **Прив'язка до клієнтів, автомобілів, категорій**

Жодна транзакція не повинна жити у вакуумі. Вона завжди має контекст: хто оплатив, за яке авто, з якою метою. Саме тому наша система дозволяє детально прив’язувати кожен запис до відповідного клієнта, автомобіля, оренди та категорії витрат чи доходів.

Ця гнучка структура дозволяє не просто зберігати записи, а будувати глибоку фінансову аналітику, що базується на реальних бізнес-діях.

- **Зв’язки у моделі Transaction**

```
car = models.ForeignKey(Car, related_name='transactions', on_delete=models.CASCADE, blank=True, null=True)
rental = models.ForeignKey(Rental, related_name='transactions', on_delete=models.CASCADE, blank=True, null=True)
client = models.ForeignKey(Customer, related_name='transactions', on_delete=models.CASCADE, blank=True, null=True)
category = models.ForeignKey(TransactionCategory, on_delete=models.CASCADE)
```

Рисунок 3.75 Поля-зв’язки у моделі Transaction

Ці поля дозволяють відстежувати, наприклад, скільки витратили на обслуговування конкретного авто або який клієнт забезпечив найбільший прибуток.

- **Приклад транзакції з повною прив’язкою**

```
Transaction.objects.create(
    client=customer,
    car=car,
    rental=rental,
    date=timezone.now().date(),
    amount=1200.00,
    type='income',
    category=income_category,
    description='Додатковий платіж за подовження оренди BMW X5'
)
```

Рисунок 3.76 Створення транзакції типу "дохід" (income) для подовження оренди

- **Категорії як інструмент аналітики**

Модель ***TransactionCategory***:

```
class TransactionCategory(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField(blank=True, null=True)
```

Рисунок 3.77 Модель TransactionCategory

Категорії можуть бути як «Оплата за оренду», «Паливо», «Техобслуговування», так і будь-які інші, які зручно аналізувати за типом витрат.

Результат:

- Прозора фінансова структура, де кожен долар має походження.

- Звіти можуть формуватися за клієнтом, авто чи категорією — максимальна деталізація.
- Система не просто зберігає числа — вона розповідає історію кожної гривні.

- **Фільтрація за типом, категорією, датою**

Уявіть фінансовий звіт, де з кількох тисяч транзакцій вам потрібно знайти тільки витрати на страхування за останній місяць. Ручний перегляд — це минуле. Наша система надає інтелектуальну фільтрацію, яка дозволяє за секунди отримати точні зрізи по типу, категорії та діапазону дат.

- **Інтерфейс фільтрації в адмінці**

Завдяки Django Admin фільтрація реалізована через панель зліва:

```
@admin.register(Transaction)
class TransactionAdmin(admin.ModelAdmin):
    list_display = ('date', 'amount', 'type', 'category', 'car', 'rental', 'client')
    list_filter = ('type', 'category', 'date')
    search_fields = ('description',)
```

Рисунок 3.78 Клас TransactionAdmin у Django Admin

Достатньо кількох кліків — і ви бачите всі «доходи» за «травень» у категорії «Оренда».

- **Фільтрація за датою**

Використання вбудованого поля ***date*** дозволяє швидко фільтрувати записи:

```
Transaction.objects.filter(date__range=['2024-05-01', '2024-05-31'])
```

Рисунок 3.79 Фільтрація транзакцій за період

- **Комбіновані фільтри**

```
Transaction.objects.filter(
    type='expense',
    category__name='Страхування',
    date__month=5,
    date__year=2024
)
```

Рисунок 3.80 Фільтрація витрат по категорії та місяцю

Такий запит одразу дає список усіх витрат на страхування за травень 2024 року.

- **Фільтрація у власному веб-інтерфейсі**

У ***views.py*** для сторінки авто реалізовано кастомний фільтр:

```
transaction_type = request.GET.get('transaction_type')
category_id = request.GET.get('category')
date_start = request.GET.get('date_start')
date_end = request.GET.get('date_end')

if transaction_type:
    transactions = transactions.filter(type=transaction_type)
if category_id:
    transactions = transactions.filter(category__id=category_id)
if date_start:
    transactions = transactions.filter(date__gte=date_start)
if date_end:
    transactions = transactions.filter(date__lte=date_end)
```

Рисунок 3.81 Динамічна фільтрація транзакцій на основі параметрів

Результат:

- Швидкий доступ до потрібних транзакцій — без зайвих зусиль.
- Можливість створювати власні фінансові звіти на льоту.
- Гнучкий фільтр — як у Django Admin, так і у користувацькому інтерфейсі.

- **Статистичний аналіз доходів і витрат**

Фінансова звітність — це не просто таблиці, а інструмент прийняття рішень. Наш додаток перетворює сухі транзакції у зрозумілу картину прибутковості, витрат за напрямками та динаміки доходу по клієнтах і автомобілях.

- **Агрегація даних — все, що треба для звіту**

Завдяки Django ORM ми можемо за кілька рядків коду отримати повну аналітичну вибірку:

```
from django.db.models import Sum, Count

# Загальна сума доходів
total_income = Transaction.objects.filter(type='income').aggregate(Sum('amount'))

# Загальна сума витрат
total_expense = Transaction.objects.filter(type='expense').aggregate(Sum('amount'))

# Доходи по категоріях
income_by_category = Transaction.objects.filter(type='income').values('category__name').annotate(total=Sum('amount'))
```

Рисунок 3.82 Підрахунок доходів і витрат, агрегація по категоріях

- **Динаміка по місяцях**

```python
from django.db.models.functions import TruncMonth

monthly_stats = (
    Transaction.objects
    .annotate(month=TruncMonth('date'))
    .values('month', 'type')
    .annotate(total=Sum('amount'))
    .order_by('month')
)
```

Рисунок 3.83 Щомісячна статистика транзакцій

Це дозволяє будувати графіки фінансового росту та оцінювати ефективність за періодами.

- **Аналіз прибутковості автопарку**

```python
income_per_car = (
    Transaction.objects.filter(type='income', car__isnull=False)
    .values('car__license_plate')
    .annotate(total=Sum('amount'))
)
```

Рисунок 3.84 Підрахунок доходу по кожному автомобілю

Таким чином легко побачити, які авто приносять найбільше доходу — і приймати рішення про заміну чи інвестування.

Результат:

- Фінансові рішення більше не приймаються навмання — система дає конкретні цифри.
- Бізнес бачить свою ефективність у розрізі місяців, категорій і клієнтів.
- Агрегати, аналітика, динаміка — усе у зручному й інтерактивному форматі.

3.3. Інтерфейс користувача: дизайн та зручність використання

У сучасному програмному забезпеченні перемогу визначає не лише функціональність, а й досвід взаємодії користувача з системою. Навіть найпотужніша CRM не стане ефективною, якщо нею незручно користуватись [10].

Саме тому під час розробки даного застосунку ми приділили особливу увагу UI/UX-дизайну, простоті навігації, швидкому доступу до основних функцій і логічній структурі всіх сторінок. Від форми реєстрації клієнта до експорту Excel — усе побудовано так, щоб користувач не думав про інтерфейс, а просто працював.

У цьому підрозділі ми розглянемо ключові дизайнерські рішення, реалізацію адаптивності, логіку переходів між розділами, та на прикладах покажемо, як система стає інструментом, а не перешкодою.

Основна мета — створити інтерфейс, у якому можна розібратися без інструкцій, і який працює швидко, передбачувано і зручно.

- **Мінімалізм та інтуїтивність**

Інтерфейс побудований за принципом «нічого зайвого». Кожна сторінка містить лише найнеобхідніші елементи — поля, кнопки, фільтри, інформаційні блоки. Завдяки цьому система не перевантажує користувача і дозволяє зосередитися на ключових діях: пошуку, редагуванні, перегляді.

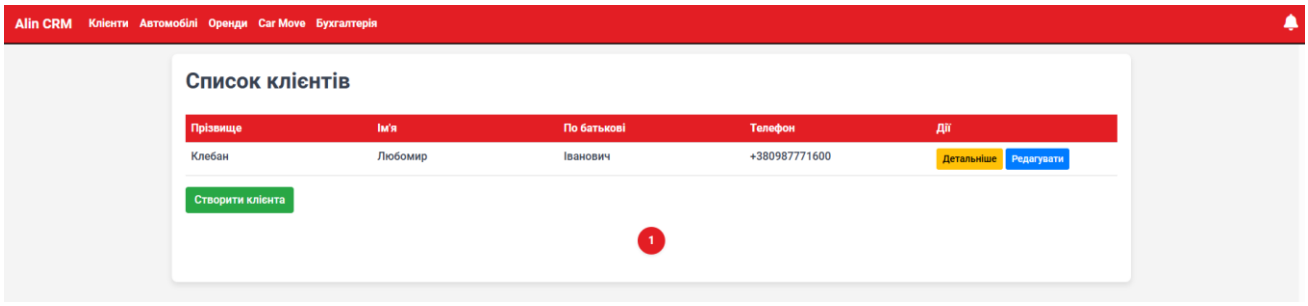


Рисунок 3.85 Простий інтерфейс переліку клієнтів без зайвих елементів

- **Єдина структура сторінок**

Усі форми створення/редагування побудовані за однаковим шаблоном: заголовок, поля введення, кнопки підтвердження/повернення. Це створює відчуття передбачуваності — користувач не губиться при переході до нових розділів.

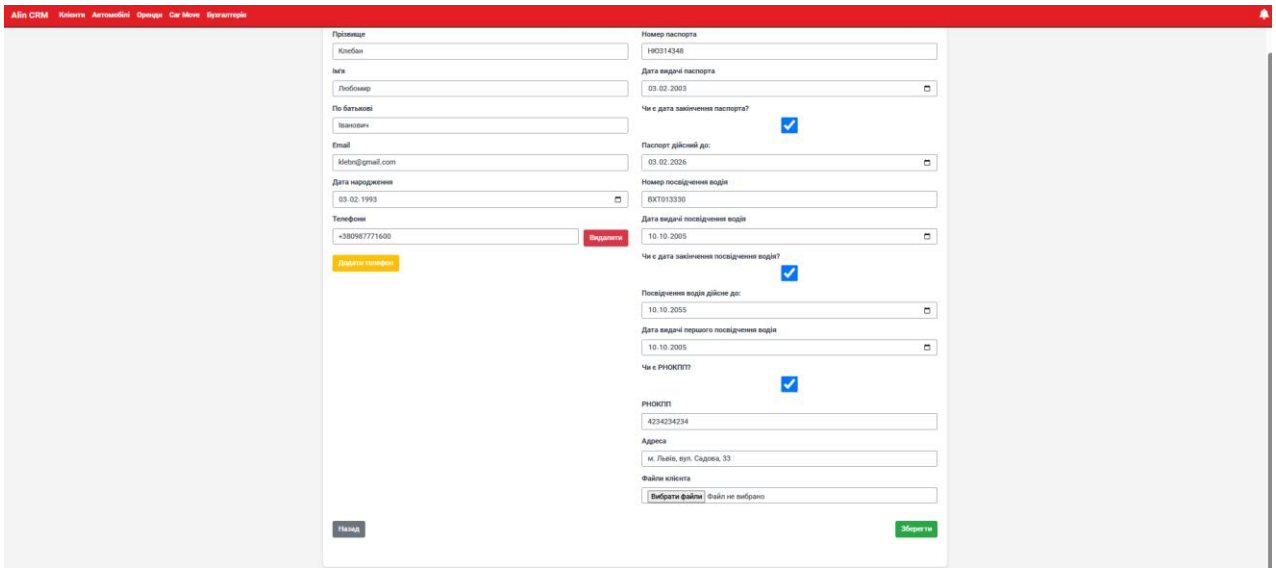


Рисунок 3.86 Уніфікована структура форми редагування даних

- **Візуальні підказки та зворотний зв’язок**

Система повідомляє користувача про успішні або помилкові дії через повідомлення зверху або під формами. Помилки валідації відображаються безпосередньо біля проблемних полів. Це дозволяє швидко виправити введені дані без втрати контексту.

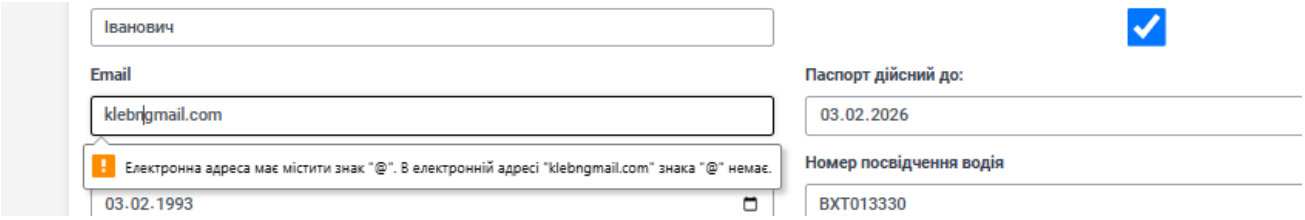


Рисунок 3.87 Повідомлення про помилку під час введення даних

Інтерфейс без зрозумілої навігації — як місто без вказівників. Навіть найфункціональніша CRM втрачає свою цінність, якщо користувач не може швидко знайти потрібний розділ або не розуміє, куди рухатись далі [11]. Тому структура навігації в нашій системі розроблена з фокусом на простоту, логіку та швидкий доступ до всіх ключових функцій.

- **Головне меню та логіка переходів**

На кожній сторінці присутній верхній рядок з головними пунктами: Клієнти, Автомобілі, Оренди, Транзакції, Календар. Це дозволяє з будь-якого місця системи одним кліком повернутись до основного розділу.

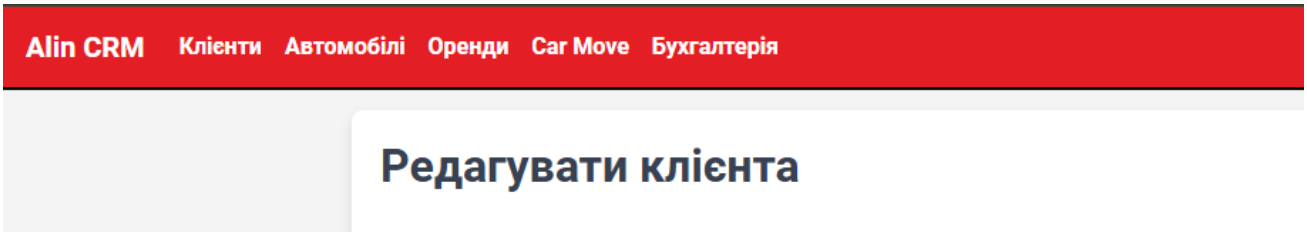


Рисунок 3.88 Головне меню навігації по системі

- **Поділ інтерфейсу на розділи: клієнти, авто, оренди, фінанси**

Кожен блок інформації винесено у власний функціональний розділ:

- Клієнти — пошук, редагування, перегляд документів і номерів.
- Автомобілі — реєстрація, технічні дані, страховки, історія пробігу.
- Оренди — створення договорів, розрахунок, історія.
- Транзакції — облік доходів/витрат, фільтрація, статистика.

Такий поділ дозволяє швидко знайти потрібну функцію, не занурюючись у вкладені меню.

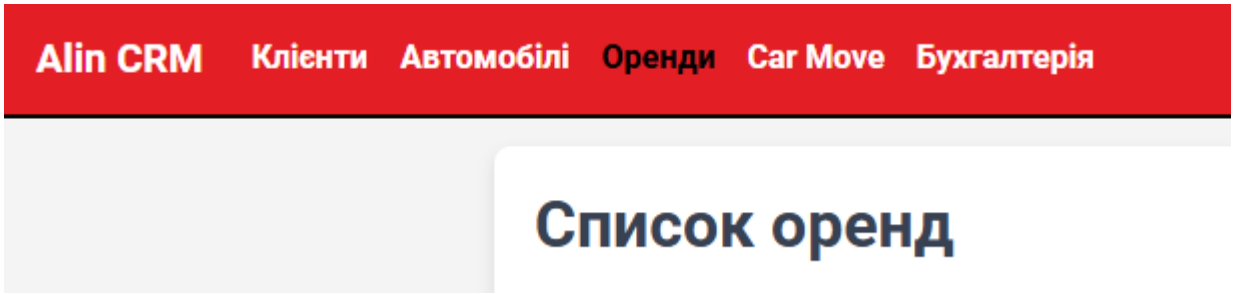


Рисунок 3.89 Структура розділів системи CRM

- **Реалізація пошуку та фільтрації**

У більшості розділів реалізовано рядок пошуку або фільтри: за іменем клієнта, маркою авто, датами оренди, категорією транзакцій тощо. Це дозволяє опрацьовувати великі масиви даних без втрати ефективності.

Підсумок:

Структура навігації в системі — це архітектурний скелет усього інтерфейсу. Вона створена так, щоб навіть новий користувач з першого дня міг ефективно працювати, не витрачаючи час на навчання.

3.3.1. Форми взаємодії з даними: створення, редагування, перегляд

У сучасній CRM-системі кожна дія з даними повинна відбуватись максимально швидко та без помилок. Саме тому ми побудували взаємодію користувача з системою на основі зручних, зрозумілих та уніфікованих форм. Від додавання нового клієнта до редагування транзакції — усі сценарії роботи з даними об’єднані однією філософією: “менше кліків — більше результату”.

- **Зручність введення даних**

Всі форми побудовані з урахуванням логічного порядку полів: спочатку — основна інформація, далі — додаткові деталі.

Поля згруповані в блоки з заголовками, що дозволяє швидко зорієнтуватися навіть у великій формі.

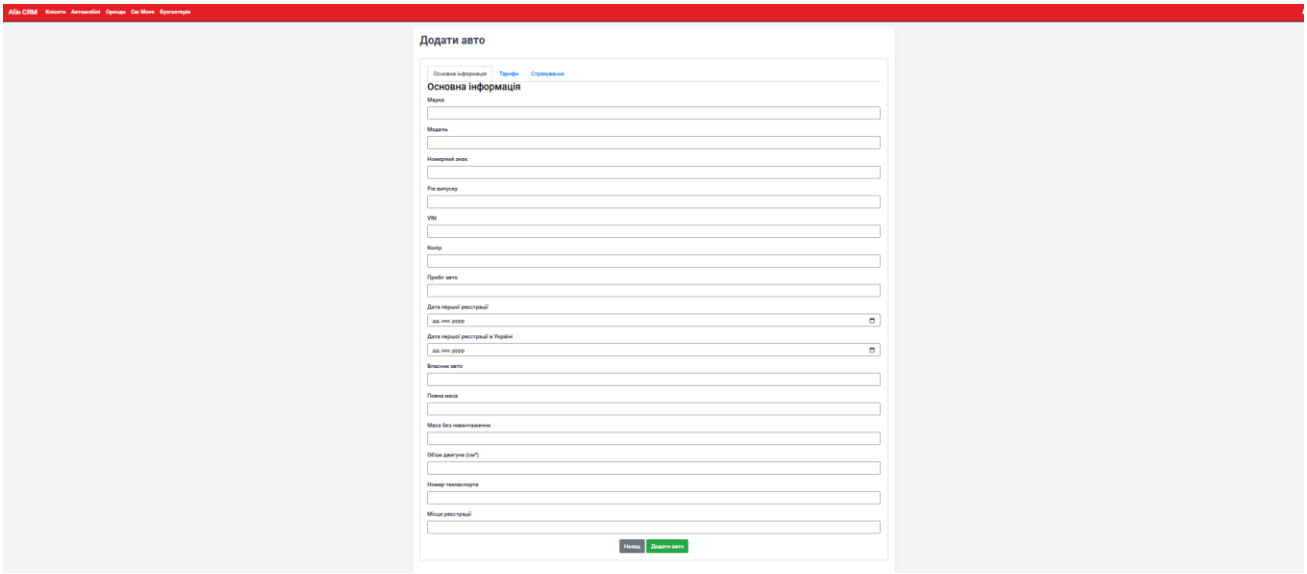


Рисунок 3.90 Структурована форма введення даних про авто

- **Валідація форм у реальному часі**

Система перевіряє коректність введених даних одразу після заповнення:

- email має бути валідним,
- обов'язкові поля не можна залишати порожніми,
- дата видачі паспорта не може бути пізнішою за сьогоднішню.

Це допомагає уникати помилок ще до відправлення форми.

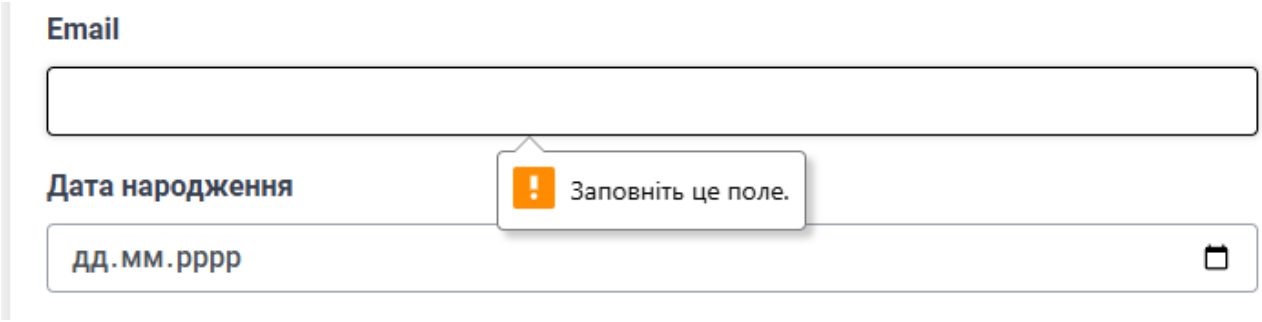


Рисунок 3.91 Повідомлення про помилку при некоректному заповненні форми

- **Повідомлення про успішні дії**

Після створення чи редагування об'єкта користувач отримує візуальне підтвердження:

- «Клієнт успішно створений»,
- «Оренда оновлена»,
- «Файл завантажено».

Ці повідомлення розташовані у верхній частині екрана і зникають автоматично через кілька секунд, що не відволікає користувача.

Підсумок:

Усі форми системи створено з акцентом на швидкість, безпомилковість і комфорт. Це дозволяє зменшити час рутинних операцій, підвищити точність даних і зменшити кількість помилок користувача.

3.3.2. Приклади сторінок інтерфейсу

Жоден опис не скаже більше, ніж живе зображення. Тому нижче наведено ключові екрани системи, які демонструють її функціональність, логіку побудови інтерфейсу та естетику.

- **Список клієнтів**

Цей екран дозволяє швидко знайти будь-якого клієнта за ім’ям, email або номером телефону. Доступні кнопки перегляду, редагування та видалення.

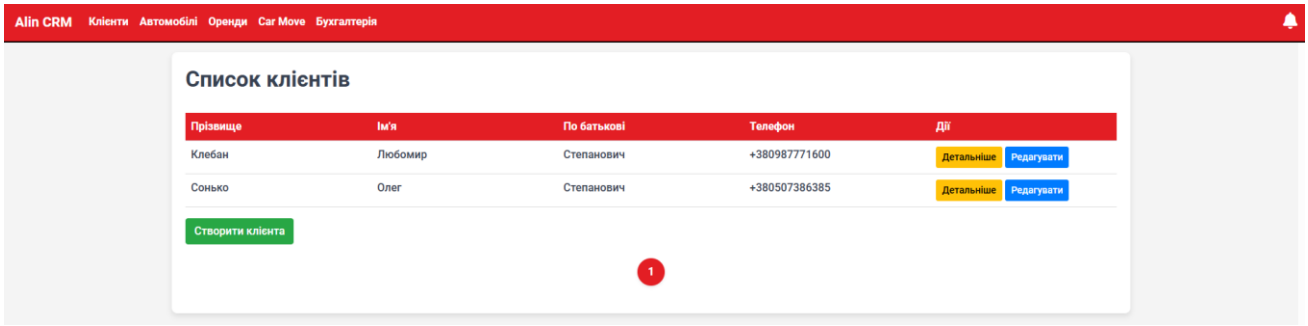


Рисунок 3.92 Список клієнтів із можливістю пошуку та дій

- **Форма додавання автомобіля**

Структурована форма із розбиттям на блоки: основна інформація, технічні дані, страховка, документи.

Всі поля мають підписи та обмеження для зменшення помилок користувача.

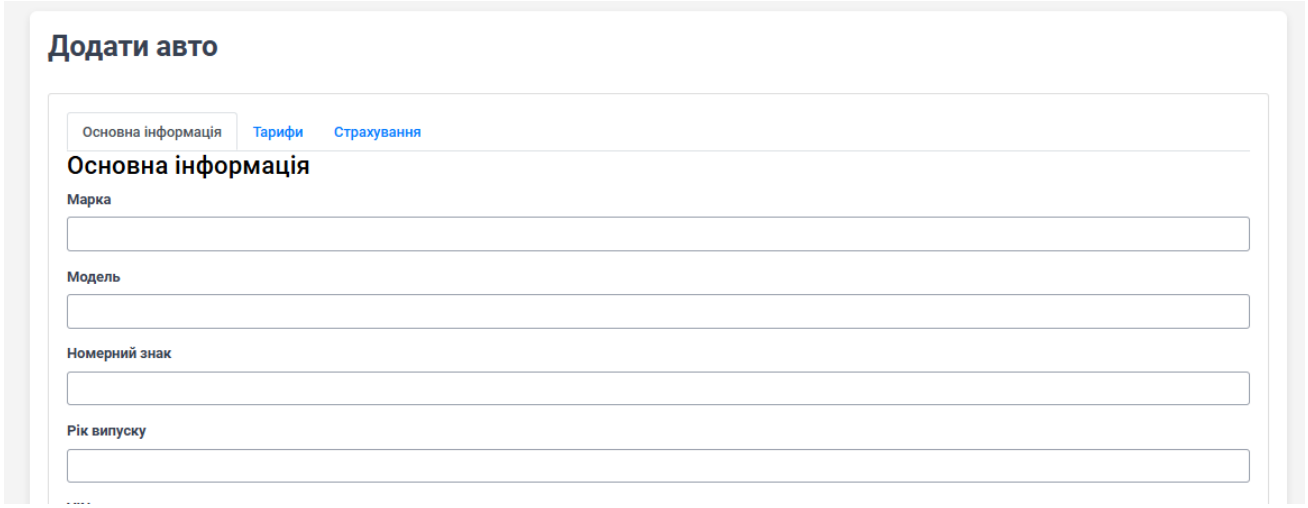


Рисунок 3.93 Форма додавання нового автомобіля

- **Огляд оренди**

На сторінці відображається повна інформація про договір оренди, включаючи клієнта, авто, дати, пробіг, додаткові опції та фінальну суму. Ідеальне місце для швидкого перегляду перед генерацією Excel.

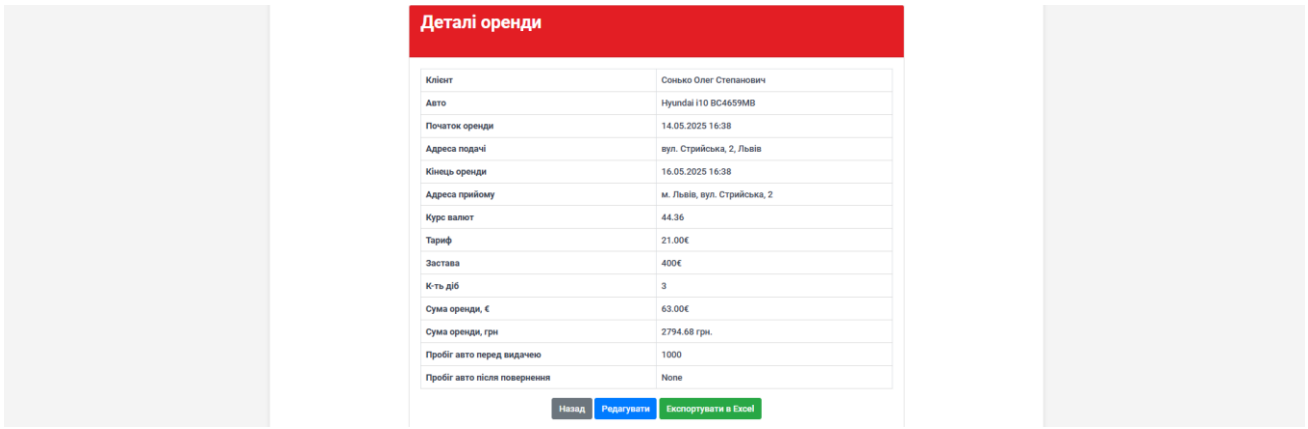


Рисунок 3.94 Детальна сторінка договору оренди

- **Транзакції**

Фінансовий розділ із можливістю фільтрувати за типом (дохід/витрата), датою та категорією.

Це зручно для швидкого перегляду фінансового стану бізнесу.

- **Календар бронювань**

Інтерактивний календар, що дозволяє побачити усі оренди по днях та автомобілях. Особливо корисно для планування автопарку.

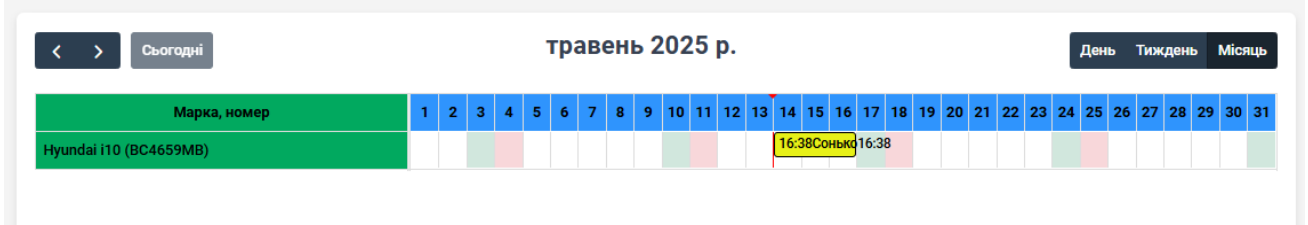


Рисунок 3.95 Календарна сітка з бронюваннями авто

Підсумок

Ці знімки не просто ілюстрації — це візуальне підтвердження якості реалізації інтерфейсу. Вони демонструють, що система є не лише технічно функціональною, а й зручною для щоденного використання.

3.4. Реалізація автоматизації завдань (планувальник).

Уявіть собі менеджера з прокату авто, якому не потрібно щодня вручну перевіряти, які клієнти повинні повернути автомобіль, чиє свято сьогодні, або які договори слід оновити. CRM-система, яку ми створили, не просто зберігає дані — вона працює замість менеджера у фоновому режимі, автоматизуючи повторювані та рутинні завдання.

Автоматизація — це мозок системи, який працює за розкладом і без людського втручання. Завдяки планувальнику завдань ми отримали змогу реалізувати щоденні нагадування, оновлення записів у базі, періодичний контроль за договорами, а також основу для майбутніх масштабованих процесів.

Цей підрозділ покаже, як саме вбудований планувальник виконує свої функції, на яких технологіях він базується та як його можна розширити для потреб бізнесу.

Автоматизація у CRM — це не просто додаткова функція, а стратегічна перевага, яка трансформує повсякденну роботу менеджера з прокату в ефективну і передбачувану систему. Уявіть собі, що найважливіші завдання виконуються точно вчасно, без затримок, людського фактора та помилок. Саме так працює наша CRM-система з вбудованим планувальником.

- **Призначення**

Основна мета автоматизації — розвантажити людину від повторюваних дій. Менеджер більше не повинен щодня вручну перевіряти, хто сьогодні повертає авто, кому надіслати нагадування, або які оренди мають бути оновлені. Замість цього система самостійно:

- перевіряє дати повернення авто;
- фільтрує відповідні оренди;
- генерує повідомлення;
- виконує потрібні дії в базі.

Таким чином, CRM перетворюється з простого реєстру клієнтів у динамічного помічника, який "думає" і діє автоматично.

Що саме автоматизується?

- Нагадування про дату повернення авто.

- Очищення/оновлення записів раз на рік (наприклад, скидання поля ***seen_birthday***).
- Майбутнє розширення: щомісячна генерація фінансових звітів, сповіщення про закінчення страховки, контроль за пробігом тощо.

Переваги:

- Швидкість реакції. Користувач отримує повідомлення вчасно, незалежно від графіка менеджера.
- Зменшення людських помилок. Автоматичний запуск виключає ризик “забути” або “не встигнути”.
- Регулярність. Завдання виконуються щоденно, щомісячно або за будь-яким заданим графіком.
- Масштабованість. При збільшенні кількості оренд або клієнтів — система працює так само стабільно, не потребуючи додаткових ресурсів менеджера.
- Економія часу. Менеджери зосереджуються на спілкуванні з клієнтами, а не на рутинних перевірках.

Автоматизація — це двигун, який дозволяє системі працювати на випередження. І саме завдяки їй CRM-платформа перетворюється з цифрового журналу в інтелектуальну систему управління прокатом.

3.4.1. Приклад реалізації щоденного нагадування про повернення авто

У системі прокату важливо не лише укладати договори, а й контролювати терміни повернення авто. Автоматичне щоденне нагадування — це не просто зручність, це спосіб уникнути затримок, спізнень та неприємних ситуацій. У цій частині покажемо, як реалізований один із таких сценаріїв — нагадування про заплановане повернення автомобіля саме сьогодні.

Логіка роботи

Кожного дня о 9:00 система має:

- Отримати поточну дату;
- Знайти всі оренди, де ***return_date*** збігається з цією датою;
- Згенерувати повідомлення або іншу дію (наприклад, відправити email, Telegram-повідомлення або зберегти лог).

Крок 1 – Функція send_reminders

```
from django.utils import timezone
from .models import Rental

def send_reminders():
    today = timezone.now().date()
    rentals = Rental.objects.filter(return_date__date=today)
    for rental in rentals:
        print(f"Нагадування: {rental.customer.email} має повернути авто сьогодні.")
```

Рисунок 3.96 Логіка функції send_reminders() для щоденних сповіщень

Що тут відбувається:

- Ми отримуємо поточну дату ***(today)***;
- Використовуємо ORM-фільтр, щоб знайти всі оренди з поверненням саме сьогодні;
- Для кожного запису виводимо повідомлення у консоль (в реальному проєкті це може бути email або push-нотифікація).

Крок 2 – Додавання завдання до планувальника

```
from apscheduler.schedulers.background import BackgroundScheduler
from .tasks import send_reminders

scheduler = BackgroundScheduler()
scheduler.add_job(send_reminders, 'cron', hour=9, minute=0)
scheduler.start()
```

Рисунок 3.97 Додавання функції до планувальника через APScheduler

Пояснення:

- ***add_job()*** реєструє завдання з типом ***'cron'***, яке виконуватиметься щодня о 09:00;
- Завдання викликає нашу функцію ***send_reminders***.

Крок 3 – Автоматичний запуск разом із Django

Щоб планувальник запускався автоматично після старту сервера, ми ініціалізували його в методі ***ready()*** додатку:

```python
# customers/apps.py
class CustomersConfig(AppConfig):
    name = 'customers'

    def ready(self):
        from customers import tasks
        from customers.scheduler import scheduler
```

Рисунок 3.98 Ініціалізація планувальника при запуску Django

Це гарантує, що планувальник буде активний після запуску runserver, без ручного втручання.

Таким чином, система щоденно автоматично перевіряє критичні дати і виконує відповідні дії. Менеджеру залишається лише відреагувати. Усе — без зайвих витрат часу, без забутих дзвінків, без людського фактора.

3.4.2. Налаштування планувальника та запуск

Автоматизація не працює без запуску. Щоб щоденні нагадування, перевірки і обчислення не залишилися просто Python-функціями, їх потрібно правильно інтегрувати в цикл життя Django-проєкту[12]. У цьому пункті покажемо, як ми налаштували планувальник ***APScheduler*** так, щоб він працював невидимо для користувача, але безвідмовно щодня.

1. Ініціалізація планувальника

Планувальник створюється один раз на старті проєкту. Ми розмістили це у файлі ***scheduler.py***, а далі підключаємо його у ***apps.py***, щоб гарантувати автоматичне підняття.

```python
# customers/scheduler.py
from apscheduler.schedulers.background import BackgroundScheduler
from customers.tasks import send_reminders

scheduler = BackgroundScheduler()
scheduler.add_job(send_reminders, 'cron', hour=9, minute=0)
scheduler.start()
```

Рисунок 3.99 Реєстрація фонової задачі через APScheduler

2. Автоматичний запуск після старту Django

Ми не хочемо запускати планувальник вручну після кожного runserver. Тому логіку ініціалізації планувальника вставлено у метод ***ready()*** класу конфігурації застосунку:

```python
# customers/apps.py
class CustomersConfig(AppConfig):
    name = 'customers'

    def ready(self):
        from customers.scheduler import scheduler
```

Рисунок 3.100 Запуск планувальника при старті проєкту

3. Відлагодження та тестування

Планувальник працює у фоновому режимі, тому вся діагностика ведеться через лог-файли. У ***settings.py*** прописано:

```python
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': os.path.join(BASE_DIR, 'debug.log'),
        },
    },
    'loggers': {
        'customers': {
            'handlers': ['file'],
            'level': 'DEBUG',
        },
    },
}
```

Рисунок 3.101 Логування подій планувальника у файл debug.log

Це дозволяє записувати всі виклики, помилки та результати виконання задач у файл ***debug.log***.

У результаті ми маємо невидимий, але потужний механізм. Планувальник піднімається автоматично, виконує завдання точно за графіком, і дає змогу системі працювати не лише за запитом користувача, а й на випередження.

Майбутні сценарії використання

- Нагадування про завершення страхових полісів.

Автоматична перевірка дати завершення КАСКО та ОСЦП, щоб заздалегідь повідомити менеджера та клієнта про потребу в оновленні.

- Автоматична генерація щомісячних фінансових звітів.

Один раз на місяць система підсумовує доходи/витрати, створює Excel-файл і надсилає його менеджеру або власнику бізнесу.

- Нагадування про техогляди та заміну масла.

На основі пробігу або дати останнього обслуговування система інформує про необхідність перевірки стану авто.

- Скидання стану фільтрів або тимчасових позначок.

*Як уже реалізовано для **seen_birthday***, можливе щоквартальне очищення тимчасових полів або тригерів.

3.5. Тестування

Після завершення етапів проєктування та реалізації критично важливим є впевнитися, що програмне забезпечення працює надійно, передбачувано та відповідає поставленим вимогам. Саме на цьому етапі — етапі тестування — система проходить перевірку на відповідність реальним сценаріям використання.

3.5.1. Юніт-тести: перевірка окремих компонентів

Юніт-тестування є фундаментальним етапом перевірки коректності логіки на рівні окремих модулів. У нашій системі юніт-тести були зосереджені на перевірці ключових моделей, форм і допоміжних методів, що обробляють бізнес-логіку.

Основні об’єкти тестування:

- Моделі: ***Customer***, ***Car***, ***Rental*** — з перевіркою правильності збереження, пов’язаних даних, обчислювальних методів.
- Моделі: ***Customer***, ***Car***, ***Rental*** — з перевіркою правильності збереження, пов’язаних даних, обчислювальних методів.
- Методи розрахунку: наприклад, ***calculate_total_rent_eur()*** або ***calculate_insurance_cost()***.

Юніт-тест для моделі Rental:

```python
from django.test import TestCase
from customers.models import Customer, Car, Rental
from django.utils import timezone
from datetime import timedelta

class RentalModelTest(TestCase):
    def setUp(self):
        self.customer = Customer.objects.create(
            first_name="Олена", last_name="Ковальчук",
            email="olena@example.com", birth_date="1995-04-10",
            passport_number="KA123456", passport_issue_date="2015-01-01",
            driver_license_number="DL987654", driver_license_issue_date="2016-01-01",
            first_driver_license_issue_date="2016-01-01"
        )
        self.car = Car.objects.create(
            make="Toyota", model="Corolla", license_plate="AA1234BB", year="2020", vin="1HGCM8263
        )

    def test_total_rent_eur_calculation(self):
        rental = Rental.objects.create(
            customer=self.customer,
            car=self.car,
            rental_date=timezone.now(),
            return_date=timezone.now() + timedelta(days=3),
            rate=30,
            currency_rate=40
        )
        self.assertEqual(rental.total_rent_eur, 120)
        self.assertEqual(rental.total_rent_uah, 4800)
```

Рисунок 3.102 Результат юніт-тестування методу обчислення вартості оренди

Результат запуску тесту в терміналі (Ran 1 test in 0.014s OK)

Юніт-тест на валідацію форми:

```python
from customers.forms import CustomerForm

class CustomerFormTest(TestCase):
    def test_invalid_rnokpp_format(self):
        form_data = {
            'first_name': 'Марія',
            'last_name': 'Петренко',
            'email': 'maria@example.com',
            'birth_date': '1990-03-12',
            'passport_number': 'AA123456',
            'passport_issue_date': '2010-01-01',
            'driver_license_number': 'DL123456',
            'driver_license_issue_date': '2011-01-01',
            'first_driver_license_issue_date': '2011-01-01',
            'rno_kpp': '123'  # некоректний формат
        }
        form = CustomerForm(data=form_data)
        self.assertFalse(form.is_valid())
        self.assertIn('rno_kpp', form.errors)
```

Рисунок 3.103 Перевірка валідації форми на прикладі поля РНОКПП

Помилка валідації поля РНОКПП при некоректному форматі

Юніт-тестування дозволило перевірити, що критичні бізнес-функції — від розрахунків до перевірки даних — працюють коректно та стабільно. Всі тести були написані згідно з принципами модульності, що забезпечує гнучкість при подальшому розширенні логіки системи.

3.5.2. Інтеграційне тестування: взаємодія між модулями

Інтеграційне тестування — це етап перевірки цілісної роботи взаємопов'язаних компонентів системи. У нашій CRM-системі, де модулі працюють у тісній взаємодії (наприклад, створення оренди тісно пов'язане з клієнтами, автомобілями та розрахунками), саме інтеграційні тести дозволяють виявити помилки, що не фіксуються в юніт-тестах, але виникають під час реального використання.

Мета інтеграційного тестування:

- Перевірити взаємодію моделей ***Customer***, ***Car*** та ***Rental***.
- Переконатися, що створення договору оренди впливає на пов'язані поля та логіку.
- Перевірити збереження пов'язаної інформації, такої як пробіг авто чи загальна сума.

Приклад тесту створення оренди з інтеграцією:

```python
from django.test import TestCase
from customers.models import Customer, Car, Rental
from django.utils import timezone
from datetime import timedelta

class RentalIntegrationTest(TestCase):
    def test_rental_creation_links_customer_and_car(self):
        customer = Customer.objects.create(
            first_name="Ігор", last_name="Демчук",
            email="ihor@example.com", birth_date="1990-05-05",
            passport_number="AA123456", passport_issue_date="2010-01-01",
            driver_license_number="DL987654", driver_license_issue_date="2011-01-01",
            first_driver_license_issue_date="2011-01-01"
        )
        car = Car.objects.create(
            make="BMW", model="320i", license_plate="AB9876CD", year="2021", vin="1HGCM82633A1234
        )
        rental = Rental.objects.create(
            customer=customer,
            car=car,
            rental_date=timezone.now(),
            return_date=timezone.now() + timedelta(days=5),
            rate=50,
            currency_rate=41.5
        )

        self.assertEqual(rental.customer.last_name, "Демчук")
        self.assertEqual(rental.car.make, "BMW")
        self.assertEqual(rental.days, 6)
        self.assertEqual(rental.total_rent_uah, round(6 * 50 * 41.5, 2))
```

Рисунок 3.104 Перевірка взаємодії між клієнтом, автомобілем та договором оренди

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.224s

OK
Destroying test database for alias 'default'...
```

Рисунок 3.105 Результат тесту у Django (успішне проходження інтеграційного тесту)

Що було перевірено:

- Правильне обчислення кількості днів оренди.

- Зв’язки ***ForeignKey*** між клієнтом, авто та орендою.
- Розрахунок загальної суми оренди з урахуванням курсу валюти.

Інтеграційне тестування дало змогу впевнитися у стабільності логіки на стику кількох сутностей — те, що є серцем CRM-системи для прокату. Завдяки цьому етапу вдалося вчасно виявити й усунути критичні помилки, пов’язані з некоректною обробкою дати повернення чи відсутністю валюти при збереженні.

3.5.3. Виявлені помилки та способи їх усунення

У процесі тестування були виявлені окремі помилки, які, хоч і не блокували основну функціональність, могли призвести до зниження зручності або втрати даних у нестандартних ситуаціях. Завдяки завчасному виявленню на етапі тестування всі недоліки були оперативно усунені.

Помилка 1: Відсутність перевірки на заповнення обмінного курсу

Суть:

Під час створення оренди без заповнення поля ***currency_rate*** сума в гривнях обчислювалась як 0, що спотворювало фінансову звітність.

Рішення:

Було додано перевірку обов’язковості цього поля на рівні форми ***RentalForm***. Тепер, якщо поле порожнє — з'являється валідаційне повідомлення.

```
currency_rate = forms.DecimalField(
    widget=forms.TextInput(),
    required=True,
    error_messages={'required': 'Потрібно вказати курс валюти для розрахунку'}
)
```

Рисунок 3.106 Поле currency_rate у формі Django

Помилка 2: Відсутність дублювання перевірки унікальності email на рівні форми

Суть:

Хоча база даних має обмеження unique=True, у формі не було зручного попередження про дубль email до відправки.

Рішення:

Додано ручну перевірку в ***CustomerForm.clean()*** з виведенням попередження:

```
if Customer.objects.filter(email=email).exclude(pk=self.instance.pk).exists():
    self.add_error('email', "Клієнт з таким email вже існує.")
```

Рисунок 3.107 Ручна перевірка в CustomerForm.clean() з виведенням попередження

Помилка 3: Обчислення пробігу не оновлювало історію, якщо пробіг збігався з попереднім

Суть:

Система не фіксувала історію пробігу, якщо його значення не змінилося — це могло створити прогалини в обліку.

Рішення:

Змінено логіку методу ***Car.save()***, щоб зберігати запис навіть за повторного значення, якщо дата зміни відрізняється:

```
if not original.mileage == self.mileage or not MileageHistory.objects.filter(car=self, mileage=self.mileage).exists():
    MileageHistory.objects.create(car=self, mileage=self.mileage)
```

Рисунок 3.108 Метод Car.save()

Помилка 4: Дублювання телефонів у клієнта

Суть:

Можна було ввести кілька однакових телефонів — особливо при редагуванні.

Рішення:

У форму ***PhoneNumberFormSet*** додано перевірку на повторення:

```
phones = [form.cleaned_data['phone'] for form in self.forms if 'phone' in form.cleaned_data]
if len(phones) != len(set(phones)):
    raise ValidationError("Телефони мають бути унікальними.")
```

Рисунок 3.109 Форма PhoneNumberFormSet з перевіркою

Всі знайдені помилки були локалізовані у ранній фазі, що дозволило уникнути серйозних наслідків у продакшн-середовищі. Завдяки детальним тестам, система стала стійкішою, а користувацький досвід — значно зручнішим.

ВИСНОВКИ

У ході виконання дипломної роботи на тему «Розроблення програмного забезпечення для роботи менеджера з прокату авто засобами Python» було досягнуто усіх поставлених цілей та виконано завдання дослідження.

Результати виконання завдань дипломної роботи:

- Проведено детальний аналіз предметної області автопрокату, виявлено актуальні потреби бізнесу щодо автоматизації процесів.
- Обґрунтовано вибір технологій розробки: Python та Django, які забезпечили високу продуктивність, швидку реалізацію та безпеку.
- Спроєктовано інформаційне середовище, у тому числі структуру бази даних, моделі, зв’язки та логіку обробки даних.
- Реалізовано повнофункціональну CRM-систему з підтримкою управління клієнтами, автопарком, договорами оренди, фінансовими транзакціями, генерацією Excel-документів та автоматичними нагадуваннями.
- Проведено тестування системи, виявлено та усунено недоліки, а також проведено порівняльний аналіз із конкурентними рішеннями.

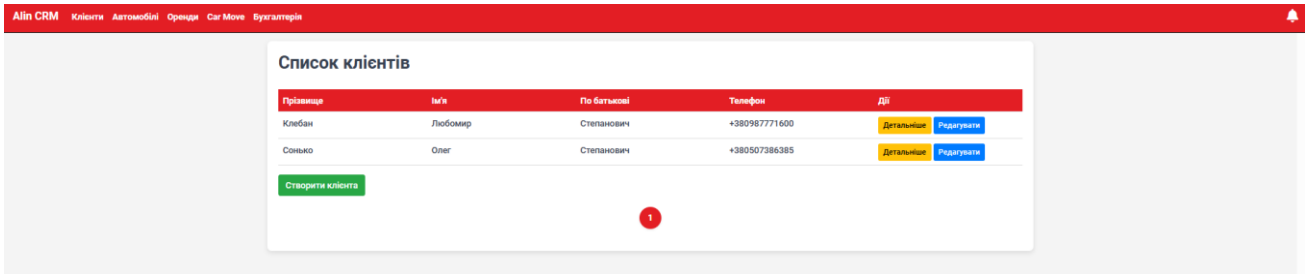
Мета дипломної роботи — створення зручної, масштабованої та функціональної CRM-системи для менеджера з прокату авто — успішно досягнута. Система продемонструвала здатність ефективно вирішувати завдання з реєстрації, управління орендами, фінансового обліку та автоматизації процесів. Вона дозволяє зменшити ручну працю, підвищити точність та контроль, а також значно прискорити щоденні операції.

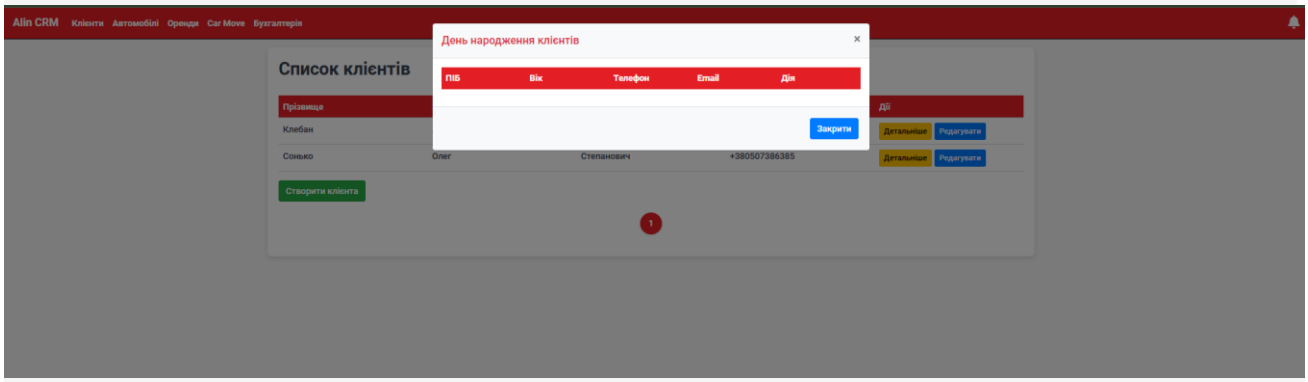
Таким чином, розроблена система є повноцінним рішенням для автоматизації процесів автопрокатної компанії, що може бути впроваджене в реальному середовищі вже сьогодні та масштабовано відповідно до зростання бізнесу.

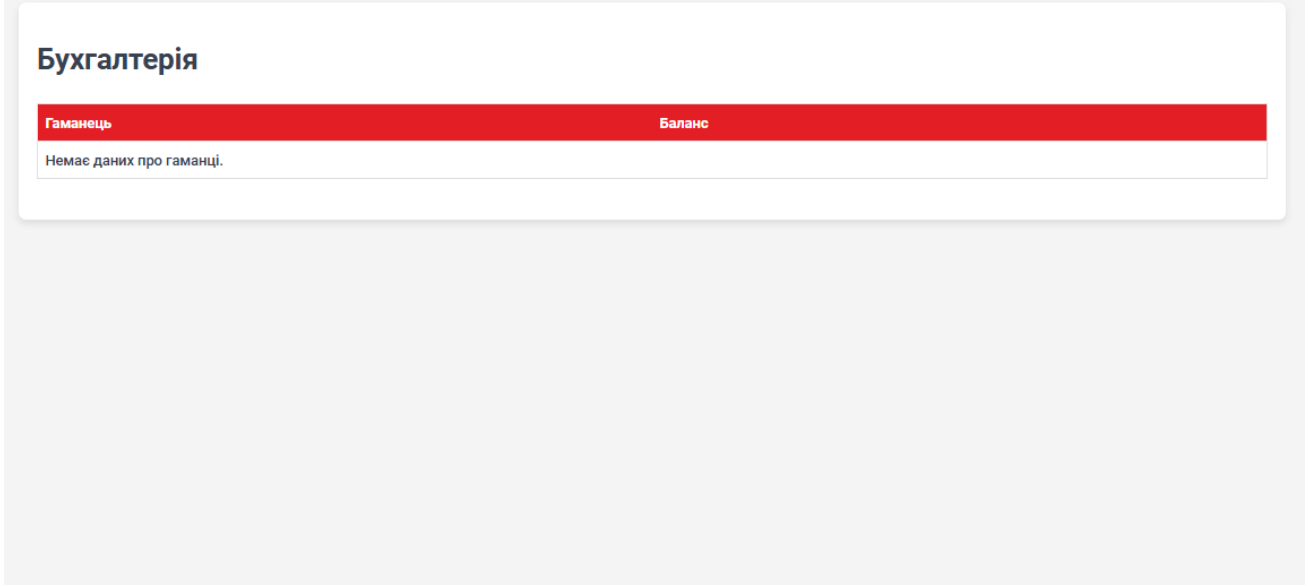
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

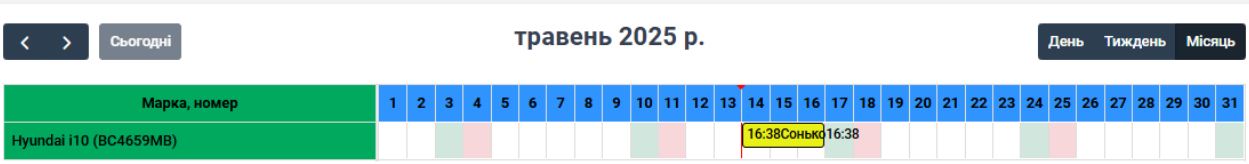
1. "Python Programming: An Introduction to Computer Science" by John Zelle – 2017. 56 с.
2. "Two Scoops of Django 3.x: Best Practices for the Django Web Framework" by Audrey Roy Greenfeld, Daniel Roy Greenfeld – 2021. 84 с.
3. "Django for Professionals" by William S. Vincent – 2020. 31 с.
4. "Fluent Python: Clear, Concise, and Effective Programming" by Luciano Ramalho – 2022
5. "High Performance Django" by Peter Baumgartner, Yann Malet – 2015. 123 с.
6. Django Project Documentation – https://docs.djangoproject.com/
7. PostgreSQL Official Documentation – https://www.postgresql.org/docs/
8. Python Official Documentation – https://docs.python.org/3/
9. Django REST framework – https://www.django-rest-framework.org/
10. Stack Overflow – Questions tagged Django – https://stackoverflow.com/questions/tagged/django
11. Real Python – Tutorials and Guides – https://realpython.com/
12. GitHub – django-apscheduler project – https://github.com/jmcarp/django-apscheduler

ДОДАТКИ


Alin CRM
Клієнти
Автомобілі
Оренди
Car Move
Бухгалтерія
Список клієнтів
Прізвище
Ім'я
По батькові
Телефон
Дії
Клебан
Любомир
Степанович
+380987771600
Детальніше
Редагувати
Сонько
Олег
Степанович
+380507386385
Детальніше
Редагувати
Створити клієнта
1


Alin CRM
Клієнти
Автомобілі
Оренди
Car Move
Бухгалтерія
День народження клієнтів
ПІБ
Вік
Телефон
Email
Дія
Закрити
Список клієнтів
Прізвище
Дії
Клебан
Детальніше
Редагувати
Сонько
Олег
Степанович
+380507386385
Детальніше
Редагувати
Створити клієнта
1


Бухгалтерія
Гаманець
Баланс
Немає даних про гаманці.

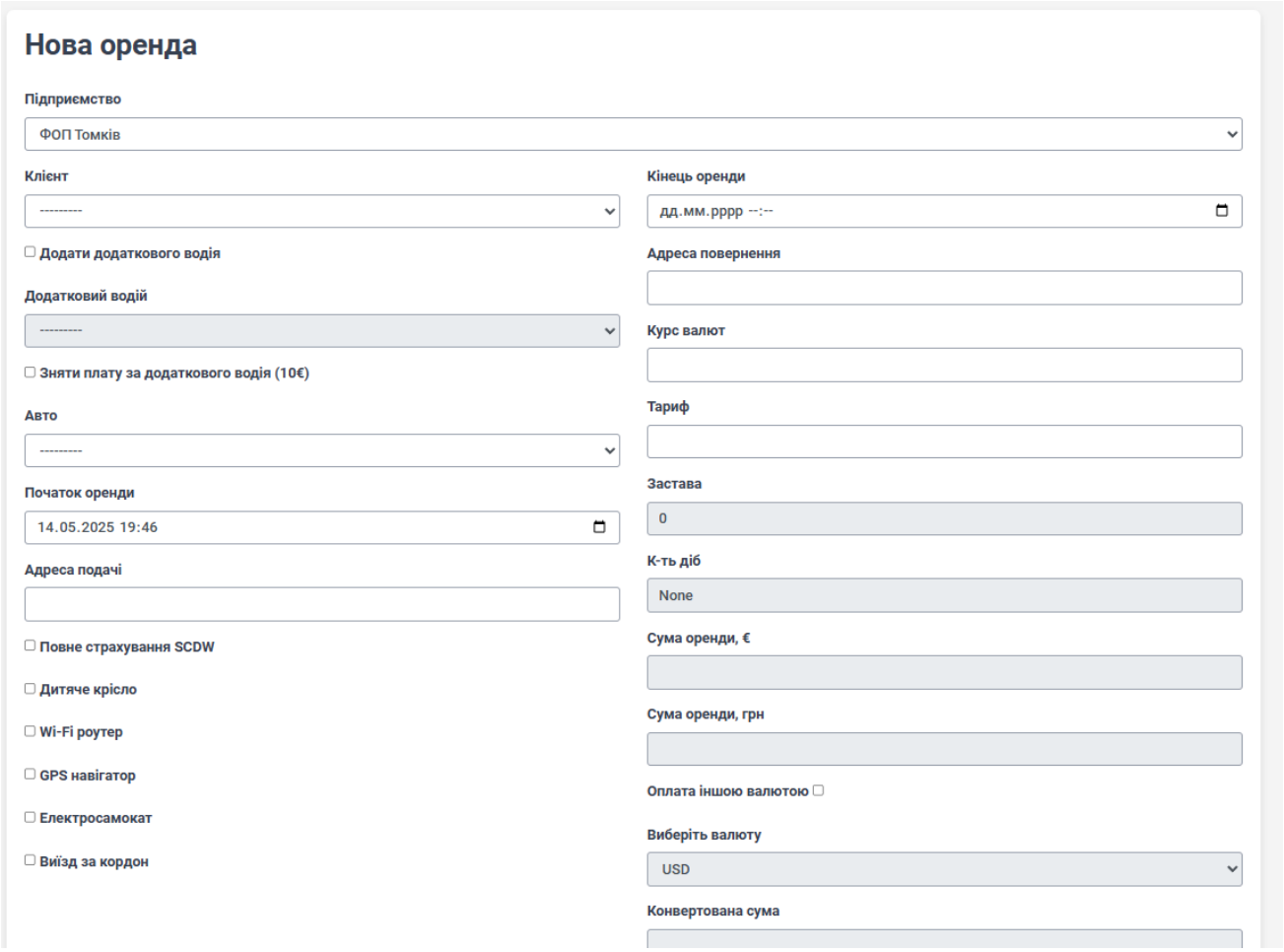


Список оренд

Клієнт	Авто	Початок оренди	Кінець оренди	К-ть діб	Тариф	Дії
Сонько Олег Степанович	Hyundai i10 BC4659MB	14.05.2025 16:38	16.05.2025 16:38	3	21.00	Детальніше Редагувати

Створити оренду

1




```
import logging
from datetime import timedelta
from django.utils import timezone
from customers.models import Car
from customers.telegram_bot import send_telegram_message
logger = logging.getLogger(__name__)
def check_insurance():
    today = timezone.now().date()
    notification_days = [30, 14, 7, 1]
    logger.info("Starting check_insurance command")
    cars = Car.objects.all()
    if not cars.exists():
        print("No cars found in the database.")
        return
    for car in cars:
        logger.info(f"Checking car: {car.license_plate}")
        sent_kasko_notifications = set()
        sent_osago_notifications = set()
        if car.kasko_expiry_date or car.osago_expiry_date:
```

```
        for days in notification_days:
            if car.kasko_expiry_date:
                notification_date_kasko = car.kasko_expiry_date - timedelta(days=days)
                if today == notification_date_kasko and days not in sent_kasko_notifications:
                    if car.owner and hasattr(car.owner, 'profile') and car.owner.profile.telegram_chat_id:
                        logger.info(f"Sending KASKO notification for {car.license_plate}")
                        send_telegram_message(car.owner, f"КАСКО {car.make} {car.model} {car.license_plate} закінчується через {days} днів. Авто застраховане в {car.kasko_insurance_company}")
                        print(f"Sent KASKO notification for {car.license_plate}")
                        sent_kasko_notifications.add(days)
                    else:
                        print(f"Owner of car {car.license_plate} does not have a telegram_chat_id or owner is missing")
                        logger.warning(f"Owner of car {car.license_plate} does not have a telegram_chat_id or owner is missing")
            if car.osago_expiry_date:
                notification_date_osago = car.osago_expiry_date - timedelta(days=days)
                if today == notification_date_osago and days not in sent_osago_notifications:
                    if car.owner and hasattr(car.owner, 'profile') and car.owner.profile.telegram_chat_id:
                        logger.info(f"Sending OSAGO notification for {car.license_plate}")
                        send_telegram_message(car.owner, f"ОСЦПВ {car.make} {car.model} {car.license_plate} закінчується через {days} днів. Авто застраховане в {car.osago_insurance_company}")
                        print(f"Sent OSAGO notification for {car.license_plate}")
                        sent_osago_notifications.add(days)
                    else:
                        print(f"Owner of car {car.license_plate} does not have a telegram_chat_id or owner is missing")
                        logger.warning(f"Owner of car {car.license_plate} does not have a telegram_chat_id or owner is missing")
        else:
            print(f"No KASKO or OSAGO expiry dates for car {car.license_plate}")
# Викличте функцію, щоб перевірити, чи всі повідомлення надсилаються лише один раз.
check_insurance()
# admin.py
# Імпортуємо необхідні модулі та класи
from django.contrib import admin  # Імпортуємо модуль адміністратора Django
from simple_history.admin import SimpleHistoryAdmin  # Імпортуємо клас SimpleHistoryAdmin для роботи з історією змін
from .models import Customer, Car, Rental, CustomerFile, PhoneNumber  # Імпортуємо моделі з поточного додатку

# Якщо модель вже була зареєстрована, спочатку відреєструйте її
try:
    admin.site.unregister(Customer)  # Спроба відреєструвати модель Customer
    admin.site.unregister(Car)  # Спроба відреєструвати модель Car
    admin.site.unregister(Rental)  # Спроба відреєструвати модель Rental
except admin.sites.NotRegistered:  # Якщо модель не зареєстрована, пропускаємо виключення
```

```
    pass

# Реєстрація моделі з використанням SimpleHistoryAdmin
class CustomerAdmin(SimpleHistoryAdmin):  # Створення класу адміністрування для моделі Customer з історією змін
    list_display = ['last_name', 'first_name', 'email', 'get_phones']  # Поля, що будуть відображатись у списку об'єктів
    search_fields = ['last_name', 'first_name', 'email', 'phone_numbers__phone']  # Поля, за якими можна буде шукати об'єкти
    history_list_display = ["status", "history_date", "history_user", "history_type"]  # Поля історії, що будуть відображатись

    def get_phones(self, obj):
        return ", ".join([phone.phone for phone in obj.phone_numbers.all()])
    get_phones.short_description = 'Телефони'
class CarAdmin(SimpleHistoryAdmin):  # Створення класу адміністрування для моделі Car з історією змін
    list_display = ['make', 'model', 'year', 'license_plate']  # Поля, що будуть відображатись у списку об'єктів
    search_fields = ['make', 'model', 'license_plate']  # Поля, за якими можна буде шукати об'єкти
    history_list_display = ["history_object", "history_date", "history_user", "history_changes"]  # Поля історії, що будуть
відображатись
class RentalAdmin(SimpleHistoryAdmin):
    list_display = ['customer', 'car', 'rental_date', 'return_date', 'rate', 'days', 'currency_rate', 'total_rent_eur', 'total_rent_uah']
    readonly_fields = ['total_rent_eur', 'total_rent_uah']
    search_fields = ['customer__first_name', 'customer__last_name', 'car__make', 'car__model']
    history_list_display = ["history_object", "history_date", "history_user", "history_changes"]

    def total_rent_eur(self, obj):
        return obj.total_rent_eur

    def total_rent_uah(self, obj):
        return obj.total_rent_uah

    total_rent_eur.short_description = 'Сума оренди, €'
    total_rent_uah.short_description = 'Сума оренди, грн'

class CustomerFileInline(admin.TabularInline):  # Створення класу для вбудованого адміністрування файлів клієнтів
    model = CustomerFile  # Модель, яка буде використовуватись у вбудованому адмініструванні
    extra = 1  # Кількість порожніх рядків для додавання нових файлів
class PhoneNumberInline(admin.TabularInline):  # Створення класу для вбудованого адміністрування телефонних
номерів
    model = PhoneNumber  # Модель, яка буде використовуватись у вбудованому адмініструванні
    extra = 1  # Кількість порожніх рядків для додавання нових телефонних номерів
class CustomerAdminWithFiles(CustomerAdmin):  # Створення класу адміністрування для моделі Customer з підтримкою
файлів
    inlines = [CustomerFileInline, PhoneNumberInline]  # Додавання вбудованого адміністрування файлів та телефонів до
моделі Customer
```

```
# Реєстрація моделей у адміністративній панелі
admin.site.register(Customer, CustomerAdminWithFiles)  # Реєстрація моделі Customer з підтримкою файлів та телефонів
admin.site.register(Car, CarAdmin)  # Реєстрація моделі Car
admin.site.register(Rental, RentalAdmin)  # Реєстрація моделі Rental з доданими полями
from django.urls import path
from . import views
urlpatterns = [
    path('', views.customer_list, name='customer_list'),
    path('customer/<int:pk>/', views.customer_detail, name='customer_detail'),
    path('customer/new/', views.customer_create, name='customer_create'),
    path('customer/<int:pk>/edit/', views.customer_update, name='customer_update'),
    path('customer/<int:pk>/delete/', views.customer_delete, name='customer_delete'),
    path('signup/', views.signup, name='signup'),
    path('cars/', views.car_list, name='car_list'),
    path('cars/<int:pk>/', views.car_detail, name='car_detail'),
    path('cars/new/', views.car_create, name='car_create'),
    path('cars/<int:pk>/edit/', views.car_update, name='car_update'),
    path('rentals/', views.rental_list, name='rental_list'),
    path('rentals/new/', views.rental_create, name='rental_create'),
    path('rentals/<int:pk>/', views.rental_detail, name='rental_detail'),
    path('rentals/<int:pk>/edit/', views.rental_update, name='rental_update'),
    path('rental/<int:pk>/export/', views.export_rental_to_excel, name='export_rental_to_excel'),
    path('calendar/', views.calendar_view, name='calendar_view'),
    path('api/events/', views.events, name='events'),
    path('api/resources/', views.resources, name='resources'),
    path('api/birthday_customers/', views.birthday_customers, name='birthday_customers'),
    path('api/mark_birthday_seen/', views.mark_birthday_seen, name='mark_birthday_seen'),
    path('update_car_mileage/<int:pk>/', views.update_car_mileage, name='update_car_mileage'),
    path('accounting/', views.accounting_view, name='accounting'), ]
```