Використання можливостей бібліотек ORM для роботи із БД на прикладі Django ORM

МЕТОДИЧНІ ВКАЗІВКИ ДЛЯ ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ ІЗ ДИСЦИПЛІНИ «БАЗИ ДАНИХ ТА ЗНАНЬ»

Рівень вищої освіти: перший (бакалаврський)

Галузь знань: 12 Інформаційні технології

Спеціальність: 124 Системний аналіз

Освітня-професійна програма: Системний аналіз

ШТЕЛЬМАХ ІГОР МИКОЛАЙОВИЧ, к.т.н., асистент кафедри системного аналізу та інформаційних технологій

Матерілаи роботи доступні за адресою: https://github.com/IgorShtelmakh/django-orm

Мета роботи

Навчитися працювати з Django ORM для створення та управління базою даних інтернет-магазину, використовуючи моделі Django, міграції, адміністративну панель та seeders для наповнення демонстраційними даними.

1. Теоретичні відомості

1.1. Що таке Django?

Django — це високорівневий веб-фреймворк для Python, який дозволяє швидко розробляти безпечні та масштабовані веб-додатки. Django був створений у 2003 році та названий на честь джазового гітариста Джанго Рейнхардта.

Ключові особливості Django:

- **Batteries included** (все включено) - Django надає готові рішення для типових завдань веб-розробки

- **MVT архітектура** (Model-View-Template) - чітке розділення бізнес-логіки, представлення даних та шаблонів
- **Безпека** - вбудований захист від CSRF, SQL-ін'єкцій, XSS атак
- **Масштабованість** - використовується великими компаніями (Instagram, Pinterest, NASA)
- **Адміністративна панель** - автоматично генерований інтерфейс для управління даними
- **ORM** - потужна система роботи з базами даних

Популярні проекти на Django:

- Instagram - соціальна мережа для обміну фото
- Spotify - музичний стрімінговий сервіс
- YouTube - частина інфраструктури
- Dropbox - хмарне сховище

1.2. Що таке Django ORM?

Django ORM (Object-Relational Mapping) - це компонент фреймворку Django, який дозволяє працювати з базою даних використовуючи Python-об'єкти замість SQL-запитів. ORM автоматично перетворює операції з об'єктами в SQL-запити та навпаки.

Як працює ORM:

```
# Замість SQL: SELECT * FROM products WHERE price > 1000
products = Product.objects.filter(price__gt=1000)

# Замість SQL: INSERT INTO categories (name) VALUES ('Electronics')
category = Category.objects.create(name='Electronics')
```

1.3. Основні переваги Django ORM:

- Незалежність від конкретної СУБД
- Захист від SQL-ін'єкцій
- Зручний Python-синтаксис
- Автоматична генерація міграцій
- Вбудована валідація даних

2. Структура БД інтернет-магазину

База даних складається з наступних таблиць:

2.1. categories - Категорії товарів

- **id** (INT, PK) - унікальний ідентифікатор категорії
- **name** (VARCHAR 120) - назва категорії (унікальна)

2.2. suppliers - Постачальники товарів

- **id** (INT, PK) - унікальний ідентифікатор постачальника
- **name** (VARCHAR 120) - назва постачальника (унікальна)
- **phone** (VARCHAR 32, nullable) - контактний телефон
- **email** (VARCHAR 150, nullable) - електронна пошта

2.3. products - Товари

- **id** (INT, PK) - унікальний ідентифікатор товару
- **sku** (VARCHAR 64) - артикул товару (унікальний)
- **name** (VARCHAR 160) - назва товару
- **category_id** (INT, FK) - посилання на категорію
- **default_supplier_id** (INT, FK, nullable) - постачальник за замовчуванням
- **list_price** (DECIMAL 12,2, nullable) - ціна товару

2.4. customers - Клієнти

- **id** (INT, PK) - унікальний ідентифікатор клієнта
- **first_name** (VARCHAR 64) - ім'я клієнта
- **last_name** (VARCHAR 64) - прізвище клієнта
- **email** (VARCHAR 150) - електронна пошта (унікальна)
- **phone** (VARCHAR 32, nullable) - контактний телефон

2.5. customer_addresses - Адреси клієнтів

- **id** (INT, PK) - унікальний ідентифікатор адреси
- **customer_id** (INT, FK) - посилання на клієнта
- **address_line** (VARCHAR 255) - адреса (вулиця, будинок)
- **city** (VARCHAR 64) - місто
- **postal_code** (VARCHAR 16) - поштовий індекс
- **country** (VARCHAR 64) - країна
- **is_default** (BOOLEAN) - чи є адреса за замовчуванням

2.6. orders - Замовлення

- **id** (INT, PK) - унікальний ідентифікатор замовлення
- **customer_id** (INT, FK) - посилання на клієнта

- **order_date** (DATETIME) - дата та час створення замовлення
- **status** (VARCHAR 20) - статус замовлення (pending, processing, shipped, delivered, cancelled)
- **shipping_address_id** (INT, FK) - адреса доставки
- **total_amount** (DECIMAL 12,2) - загальна сума замовлення

2.7. order_items - Позиції замовлення

- **id** (INT, PK) - унікальний ідентифікатор позиції
- **order_id** (INT, FK) - посилання на замовлення
- **product_id** (INT, FK) - посилання на товар
- **quantity** (INT) - кількість товару
- **unit_price** (DECIMAL 12,2) - ціна за одиницю на момент замовлення

2.8. payments - Платежі

- **id** (INT, PK) - унікальний ідентифікатор платежу
- **order_id** (INT, FK) - посилання на замовлення
- **payment_date** (DATETIME) - дата та час платежу
- **amount** (DECIMAL 12,2) - сума платежу
- **payment_method** (VARCHAR 32) - спосіб оплати (card, cash, bank_transfer)
- **status** (VARCHAR 20) - статус платежу (pending, completed, failed, refunded)

2.9. shipments - Відправлення

- **id** (INT, PK) - унікальний ідентифікатор відправлення
- **order_id** (INT, FK) - посилання на замовлення
- **shipment_date** (DATETIME) - дата відправлення
- **carrier** (VARCHAR 64) - перевізник (Нова Пошта, Укрпошта, тощо)
- **tracking_number** (VARCHAR 100, nullable) - трекінг-номер відправлення
- **status** (VARCHAR 20) - статус відправлення (preparing, shipped, in_transit, delivered)

3. Рекомендована структура проекту

Після завершення всіх кроків, ваш проект матиме наступну структуру:

```
shop_project/
├── shop/                          # Головний проект Django
│   ├── __init__.py
│   ├── settings.py                # Налаштування проекту
│   ├── urls.py                    # URL маршрути
│   ├── asgi.py
```

```
│   └── wsgi.py
├── store/                          # Додаток магазину
│   ├── __init__.py
│   ├── apps.py
│   ├── models/                     # Моделі (організовані у окремі файли)
│   │   ├── __init__.py             # Імпорт всіх моделей
│   │   ├── category.py             # Модель категорій
│   │   ├── supplier.py             # Модель постачальників
│   │   ├── product.py              # Модель товарів
│   │   ├── customer.py             # Моделі клієнтів та адрес
│   │   ├── order.py                # Моделі замовлень
│   │   ├── payment.py              # Модель платежів
│   │   └── shipment.py             # Модель відправлень
│   ├── admin/                      # Адмін-панель (організована у окремі
файли)
│   │   ├── __init__.py             # Імпорт всіх admin класів
│   │   ├── category.py             # Admin для категорій
│   │   ├── product.py              # Admin для товарів
│   │   ├── customer.py             # Admin для клієнтів
│   │   └── order.py                # Admin для замовлень
│   ├── management/                 # Кастомні команди
│   │   ├── __init__.py
│   │   └── commands/
│   │       ├── __init__.py
│   │       └── seed.py             # Команда для наповнення БД
│   ├── migrations/                 # Міграції бази даних
│   │   ├── __init__.py
│   │   └── 0001_initial.py         # Створюється автоматично
│   ├── tests.py                    # Тести (опціонально)
│   └── views.py                    # Views (для API або веб-інтерфейсу)
├── manage.py                       # CLI для управління проектом
├── db.sqlite3                      # База даних SQLite (якщо
використовується)
└── venv/                           # Віртуальне середовище Python
```

Переваги такої організації:

- Чітка структура проекту
- Легко знайти потрібний файл
- Зручно працювати в команді
- Масштабованість - легко додавати нові моделі/admin класи
- Менше конфліктів при git merge

4. Хід роботи

Крок 1: Встановлення Python та створення віртуального середовища

```
# Перевірка версії Python (потрібна версія 3.8+)
python3 --version

# Створення директорії проекту
mkdir shop_project
cd shop_project

# Створення віртуального середовища
python3 -m venv venv

# Активація віртуального середовища
# Для macOS/Linux:
source venv/bin/activate
# Для Windows:
venv\Scripts\activate
```

Крок 2: Встановлення Django

```
# Встановлення Django
pip install django

# Перевірка версії
django-admin --version

# Встановлення додаткових пакетів
pip install mysqlclient  # для роботи з MySQL
# або
pip install psycopg2-binary  # для PostgreSQL
```

Крок 3: Створення Django проекту

```
# Створення нового проекту
django-admin startproject shop .
```

```
# Створення додатку для роботи з магазином
python manage.py startapp store
```

Крок 4: Налаштування проекту

Відредагуйте файл `shop/settings.py`:

```
# shop/settings.py

# Додайте додаток до INSTALLED_APPS
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'store',  # наш додаток
]

# Налаштування підключення до бази даних MySQL
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'django_orm',
        'USER': 'root',
        'PASSWORD': 'your_password',
        'HOST': '127.0.0.1',
        'PORT': '3306',
        'OPTIONS': {
            'charset': 'utf8mb4',
        },
    }
}

# Налаштування мови та часового поясу
LANGUAGE_CODE = 'uk'
TIME_ZONE = 'Europe/Kiev'
USE_I18N = True
USE_TZ = True
```

Крок 5: Створення моделей

Теоретичні відомості про моделі та ORM

Що таке моделі в Django?

Модель у Django - це Python-клас, який представляє таблицю в базі даних. Кожен атрибут класу відповідає полю (колонці) таблиці. Django автоматично створює SQL-таблиці на основі визначених моделей.

Основні принципи роботи з моделями:

1. **Один клас = одна таблиця** - кожна модель представляє окрему таблицю в БД
2. **Атрибути = поля таблиці** - кожен атрибут класу стає колонкою в таблиці
3. **Автоматичний ID** - Django автоматично додає поле `id` як первинний ключ
4. **Типізація даних** - Django надає спеціальні типи полів (CharField, IntegerField, DateField тощо)

Django ORM у дії:

ORM (Object-Relational Mapping) дозволяє працювати з базою даних як з Python-об'єктами:

```
# Замість SQL: SELECT * FROM products WHERE price > 1000
products = Product.objects.filter(list_price__gt=1000)

# Замість SQL: INSERT INTO categories (name) VALUES ('Ноутбуки')
category = Category.objects.create(name='Ноутбуки')

# Замість SQL: UPDATE products SET price = 2000 WHERE id = 1
product = Product.objects.get(id=1)
product.list_price = 2000
product.save()
```

Основні типи полів моделі:

- `CharField` - текстове поле обмеженої довжини (потрібен параметр max_length)
- `TextField` - текстове поле необмеженої довжини
- `IntegerField` - ціле число
- `DecimalField` - десяткове число (для грошових сум)
- `DateField` / `DateTimeField` - дата / дата та час
- `BooleanField` - логічне значення (True/False)
- `EmailField` - електронна пошта (з валідацією)
- `ForeignKey` - зв'язок "один до багатьох"
- `ManyToManyField` - зв'язок "багато до багатьох"

Важливі параметри полів:

- `null=True` - дозволяє NULL значення в БД
- `blank=True` - дозволяє порожні значення при валідації форм
- `unique=True` - поле має бути унікальним
- `default` - значення за замовчуванням
- `verbose_name` - зрозуміла назва поля (для адмін-панелі)
- `validators` - список валідаторів для перевірки даних

Типи зв'язків (on_delete):

При визначенні ForeignKey потрібно вказати поведінку при видаленні пов'язаного об'єкта:

- `CASCADE` - видалити всі пов'язані об'єкти (каскадне видалення)
- `PROTECT` - заборонити видалення, якщо є пов'язані об'єкти
- `SET_NULL` - встановити NULL (потрібен параметр null=True)
- `SET_DEFAULT` - встановити значення за замовчуванням
- `RESTRICT` - заборонити видалення з помилкою (схоже на PROTECT)

Клас Meta:

Клас Meta всередині моделі визначає метадані:

```
class Meta:
    db_table = 'categories'            # назва таблиці в БД
    verbose_name = "Категорія"         # назва в однині
    verbose_name_plural = "Категорії" # назва в множині
    ordering = ['name']                 # сортування за замовчуванням
    unique_together = [['field1', 'field2']]  # унікальна комбінація полі
```

Метод str():

Метод `__str__()` визначає текстове представлення об'єкта (використовується в адмін-панелі та при виводі):

```
def __str__(self):
    return self.name  # або будь-яке інше зрозуміле представлення
```

Практичне створення моделей

Створіть моделі в файлі `store/models/[model_name].py`. Див. приклад нижче для моделі Category:

```python
# store/models/category.py
from django.db import models


class Category(models.Model):
    """Модель категорії товарів"""
    name = models.CharField(max_length=120, unique=True, verbose_name="На

    class Meta:
        db_table = 'categories'
        verbose_name = "Категорія"
        verbose_name_plural = "Категорії"
        ordering = ['name']

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

Завдання: Створіть моделі для всіх таблиць:

- Supplier (постачальники)
- Product (товари)
- Customer (клієнти)
- CustomerAddress (адреси клієнтів)
- Order (замовлення)
- OrderItem (позиції замовлення)
- Payment (платежі)
- Shipment (відправлення)

Організація моделей у окремі файли (рекомендовано для великих проектів)

Для кращої організації коду, особливо у великих проектах, краще розділити моделі на окремі файли:

Крок 5.1: Створіть моделі у відповідних файлах:

```python
# store/models/category.py
from django.db import models


class Category(models.Model):
    """Модель категорії товарів"""
    name = models.CharField(max_length=120, unique=True, verbose_name="На
```

```
    class Meta:
        db_table = 'categories'
        verbose_name = "Категорія"
        verbose_name_plural = "Категорії"
        ordering = ['name']

    def __str__(self):
        return self.name


# store/models/supplier.py
from django.db import models


class Supplier(models.Model):
    """Модель постачальника"""
    name = models.CharField(max_length=120, unique=True, verbose_name="На
    phone = models.CharField(max_length=32, null=True, blank=True, verbos
    email = models.EmailField(max_length=150, null=True, blank=True, verb

    class Meta:
        db_table = 'suppliers'
        verbose_name = "Постачальник"
        verbose_name_plural = "Постачальники"
        ordering = ['name']

    def __str__(self):
        return self.name


# store/models/product.py
from django.db import models
from django.core.validators import MinValueValidator
from decimal import Decimal


class Product(models.Model):
    """Модель товару"""
    sku = models.CharField(max_length=64, unique=True, verbose_name="Арти
    name = models.CharField(max_length=160, verbose_name="Назва")
    category = models.ForeignKey(
        'Category',  # Посилання на модель з іншого файлу
        on_delete=models.RESTRICT,
        related_name='products',
        verbose_name="Категорія"
```

```
    )
    default_supplier = models.ForeignKey(
        'Supplier',  # Посилання на модель з іншого файлу
        on_delete=models.SET_NULL,
        null=True,
        blank=True,
        related_name='products',
        verbose_name="Постачальник за замовчуванням"
    )
    list_price = models.DecimalField(
        max_digits=12,
        decimal_places=2,
        null=True,
        blank=True,
        validators=[MinValueValidator(Decimal('0.01'))],
        verbose_name="Ціна"
    )

    class Meta:
        db_table = 'products'
        verbose_name = "Товар"
        verbose_name_plural = "Товари"
        ordering = ['name']

    def __str__(self):
        return f"{self.sku} - {self.name}"
```

Крок 5.3: Імпортуйте всі моделі у `store/models/__init__.py`:

```
# store/models/__init__.py
from .category import Category
from .supplier import Supplier
from .product import Product
from .customer import Customer, CustomerAddress
from .order import Order, OrderItem
from .payment import Payment
from .shipment import Shipment

# Це дозволяє імпортувати моделі як: from store.models import Product
__all__ = [
    'Category',
    'Supplier',
    'Product',
```

```
    'Customer',
    'CustomerAddress',
    'Order',
    'OrderItem',
    'Payment',
    'Shipment',
]
```

Переваги такої структури:

- Кожна модель у своєму файлі - легше знаходити та редагувати
- Менше конфліктів при роботі в команді
- Простіше тестувати окремі моделі
- Краща організація коду для великих проектів
- Можна групувати пов'язані моделі (наприклад, Customer і CustomerAddress в одному файлі)

Примітка: При використанні зовнішніх ключів (ForeignKey) на моделі з інших файлів, використовуйте рядкові посилання: `'Category'` замість прямого імпорту класу. Django автоматично знайде модель під час виконання.

Крок 6: Генерація та застосування міграцій

Що таке міграції?

Міграції - це спосіб Django відстежувати та застосовувати зміни в структурі бази даних. Міграції являють собою Python-файли, які містять інструкції про те, як змінити схему БД (створити таблиці, додати поля, змінити типи даних тощо).

Навіщо потрібні міграції?

1. **Версійний контроль схеми БД** - міграції зберігаються у вигляді файлів, які можна додати до git. Це дозволяє відстежувати історію змін структури БД так само, як і зміни коду.
2. **Синхронізація команди** - коли інший розробник завантажує ваш код, він може просто запустити `migrate` і отримати актуальну структуру БД.
3. **Безпечні зміни** - Django автоматично генерує SQL-код для зміни структури БД, враховуючи збереження існуючих даних.
4. **Відкат змін** - можна повернутися до попередньої версії схеми БД, якщо щось пішло не так.
5. **Незалежність від СУБД** - міграції працюють однаково для MySQL, PostgreSQL, SQLite та інших баз даних.

Як працюють міграції?

1. Ви змінюєте моделі в `models.py`
2. Запускаєте `makemigrations` - Django аналізує зміни і створює файл міграції
3. Запускаєте `migrate` - Django застосовує міграції до БД

Практичне застосування

```
# Створення міграцій на основі моделей
python manage.py makemigrations

# Перегляд SQL-коду міграції (опціонально)
python manage.py sqlmigrate store 0001

# Застосування міграцій до БД
python manage.py migrate

# Перевірка статусу міграцій
python manage.py showmigrations
```

Пояснення команд:

- `makemigrations` - аналізує зміни в моделях і створює файли міграцій у директорії `migrations/`
- `migrate` - застосовує всі непримінені міграції до бази даних
- `sqlmigrate` - показує SQL-код, який буде виконано (корисно для розуміння що відбувається)
- `showmigrations` - показує список всіх міграцій та їх статус ([X] - застосована, [ ] - не застосована)

Крок 7: Створення суперкористувача

```
# Створення адміністратора для доступу до адмін-панелі
python manage.py createsuperuser

# Введіть дані:
# Username: admin
# Email: admin@example.com
# Password: admin
```

Крок 8: Робота з моделями в Django Shell

Django Shell – це інтерактивна консоль Python з повним доступом до моделей та функціоналу Django проекту. Це зручний інструмент для швидкого тестування запитів та експериментів з даними.

Запуск Django Shell:

```
python manage.py shell
```

Основні операції CRUD (Create, Read, Update, Delete)

1. Створення записів (Create):

```
from store.models import Category, Product, Supplier
from decimal import Decimal

# Створення і збереження одразу
category = Category.objects.create(name='Ноутбуки')

# Створення з подальшим збереженням
supplier = Supplier(name='TechSupply Ltd', email='info@techsupply.com')
supplier.save()

# Створення товару з зв'язками
product = Product.objects.create(
    sku='LAP001',
    name='MacBook Pro 14"',
    category=category,
    default_supplier=supplier,
    list_price=Decimal('55999.99')
)
```

2. Читання записів (Read):

```
# Отримати всі записи
categories = Category.objects.all()

# Отримати один запис за ID
category = Category.objects.get(id=1)

# Фільтрація записів
```

```
laptops = Product.objects.filter(category__name='Ноутбуки')
expensive = Product.objects.filter(list_price__gte=30000)

# Підрахунок кількості
count = Product.objects.count()
```

3. Оновлення записів (Update):

```
# Оновлення одного запису
product = Product.objects.get(sku='LAP001')
product.list_price = Decimal('52999.99')
product.save()

# Масове оновлення
Product.objects.filter(category__name='Ноутбуки').update(
    list_price=Decimal('45000')
)
```

4. Видалення записів (Delete):

```
# Видалення одного запису
product = Product.objects.get(id=1)
product.delete()

# Масове видалення
Product.objects.filter(list_price__isnull=True).delete()
```

Крок 9: Написання seeders для наповнення демо-даними

Теоретичні відомості про seeders

Seeders (сідери) — це скрипти або команди, які використовуються для автоматичного наповнення бази даних початковими або тестовими даними. У Django seeders допомагають швидко створити демо-дані для розробки, тестування або демонстрації проекту.

Чому seeders важливі?

1. **Швидкий старт розробки** - Замість ручного введення даних через адмін-панель, можна одною командою наповнити БД
2. **Консистентність даних** - Всі розробники в команді отримують однаковий набір тестових даних

3. **Автоматизація тестування** - Швидке створення реалістичних наборів даних для перевірки функціоналу
4. **Демонстрація проекту** - Презентабельні дані для показу клієнтам або стейкхолдерам
5. **Відновлення після помилок** - Можливість швидко пересоздати базу даних після збоїв

Способи реалізації seeders в Django:

1. **Custom Management Commands (рекомендовано)**
   - Створення власних команд у `management/commands/`
   - Максимальна гнучкість та контроль
   - Можливість використання логіки Python
2. **Fixtures (JSON/YAML файли)**
   - Статичні файли з даними
   - Команди: `dumpdata` для експорту, `loaddata` для імпорту
   - Зручно для незмінних даних (довідники, налаштування)
3. **Міграції з RunPython**
   - Вбудовування даних безпосередньо в міграції
   - Гарантія виконання при розгортанні проекту
4. **Бібліотеки (Factory Boy, Faker)**
   - Генерація реалістичних випадкових даних
   - Корисно для створення великих обсягів тестових даних

Основні принципи написання seeders:

```
# Типова структура seeder-команди
class Command(BaseCommand):
    help = 'Опис команди'

    def handle(self, *args, **options):
        # 1. Очищення старих даних (опціонально)
        Model.objects.all().delete()

        # 2. Створення даних
        obj = Model.objects.create(field='value')

        # 3. Bulk створення для продуктивності
        objects = [Model(field='value') for i in range(100)]
        Model.objects.bulk_create(objects)
```

```
        # 4. Повідомлення про успіх
        self.stdout.write(self.style.SUCCESS('Готово!'))
```

Порівняння методів створення об'єктів:

- `Model.objects.create()` - створює і зберігає один об'єкт за раз
- `Model.objects.bulk_create()` - створює багато об'єктів одним запитом (швидше)
- `Model.objects.get_or_create()` - отримує або створює, якщо не існує

Використання бібліотеки Faker (опціонально):

```
from faker import Faker
fake = Faker('uk_UA')  # Українська локалізація

# Генерація реалістичних даних
name = fake.company()
email = fake.email()
phone = fake.phone_number()
address = fake.address()
```

Практична реалізація seeders

Створіть директорії та файл для команди:

```
mkdir -p store/management/commands
touch store/management/__init__.py
touch store/management/commands/__init__.py
```

Створіть файл `store/management/commands/seed.py` :

```
# store/management/commands/seed.py
from django.core.management.base import BaseCommand
from store.models import Category, Supplier, Product, Customer
from decimal import Decimal


class Command(BaseCommand):
    help = 'Наповнення бази даних демонстраційними даними'

    def handle(self, *args, **kwargs):
        self.stdout.write('Початок наповнення БД...')
```

```
        # Створення категорій
        categories = [
            Category(name='Ноутбуки'),
            Category(name='Смартфони'),
            Category(name='Телевізори'),
            Category(name='Побутова техніка'),
            Category(name='Аудіотехніка'),
        ]
        Category.objects.bulk_create(categories)
        self.stdout.write(self.style.SUCCESS(f'Створено {len(categories)}

        # Додайте створення інших даних...

        self.stdout.write(self.style.SUCCESS('База даних успішно наповнен
```

Запуск seeder:

```
python manage.py seed
```

Крок 10: Розробка адміністративної панелі

Відредагуйте файл `store/admin.py`:

```
#store/admin.py
from django.contrib import admin
from django.utils.html import format_html
from .models import Category, Supplier, Product, Customer


@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['id', 'name', 'products_count']
    search_fields = ['name']
    ordering = ['name']

    def products_count(self, obj):
        """Кількість товарів у категорії"""
        return obj.products.count()
    products_count.short_description = 'Кількість товарів'


@admin.register(Product)
```

```
class ProductAdmin(admin.ModelAdmin):
    list_display = ['id', 'sku', 'name', 'category', 'default_supplier',
    list_filter = ['category', 'default_supplier']
    search_fields = ['sku', 'name']
    ordering = ['name']
    list_per_page = 20


# Налаштування заголовків адмін-панелі
admin.site.site_header = "Адміністрування інтернет-магазину"
admin.site.site_title = "Shop Admin"
admin.site.index_title = "Панель управління"
`

#### Організація адмін-панелі у окремі файли (рекомендовано)

Аналогічно до моделей, адміністративні класи також краще організувати у с

**Крок 10.1:** Створіть структуру для admin:

```bash
# Видаліть файл admin.py
rm store/admin.py

# Створіть директорію для admin класів
mkdir store/admin

# Створіть файли
touch store/admin/__init__.py
touch store/admin/category.py
touch store/admin/product.py
touch store/admin/customer.py
touch store/admin/order.py
```

Крок 10.2: Приклад організації admin класів:

```
# store/admin/category.py
from django.contrib import admin
from store.models import Category


@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
```
```

```
    list_display = ['id', 'name', 'products_count']
    search_fields = ['name']
    ordering = ['name']

    def products_count(self, obj):
        """Кількість товарів у категорії"""
        return obj.products.count()
    products_count.short_description = 'Кількість товарів'


# store/admin/product.py
from django.contrib import admin
from django.utils.html import format_html
from store.models import Product


@admin.register(Product)
class ProductAdmin(admin.ModelAdmin):
    list_display = ['id', 'sku', 'name', 'category', 'default_supplier',
    list_filter = ['category', 'default_supplier']
    search_fields = ['sku', 'name']
    ordering = ['name']
    list_per_page = 20

    def price_colored(self, obj):
        """Відображення ціни з кольоровим індикатором"""
        if obj.list_price:
            if obj.list_price > 40000:
                color = 'red'
            elif obj.list_price > 20000:
                color = 'orange'
            else:
                color = 'green'
            return format_html(
                '<span style="color: {};">{} грн</span>',
                color,
                obj.list_price
            )
        return '-'
    price_colored.short_description = 'Ціна (кольорова)'
```

Крок 10.3: Імпортуйте всі admin класи у `store/admin/__init__.py`:

```
# store/admin/__init__.py
from django.contrib import admin

# Імпортуємо всі admin класи
from .category import CategoryAdmin
from .product import ProductAdmin
from .customer import CustomerAdmin
from .order import OrderAdmin

# Налаштування заголовків адмін-панелі
admin.site.site_header = "Адміністрування інтернет-магазину"
admin.site.site_title = "Shop Admin"
admin.site.index_title = "Панель управління"
```

Переваги:

- Кожна адміністративна панель у своєму файлі
- Легше знаходити та модифікувати налаштування
- Зручніше при розробці в команді
- Можна легко вимкнути певну адмін-панель, закоментувавши імпорт

Крок 11: Запуск сервера

```
python manage.py runserver
```

Відкрийте браузер за адресою: `http://127.0.0.1:8000/admin/`

5. Завдання для самостійної роботи

Базовий рівень (обов'язково):

Модифікація моделей:

- Додайте поле `description` (опис) до моделі `Product`
- Додайте поле `discount` (знижка у відсотках) до моделі `Product`
- Створіть та застосуйте міграції

Робота з Django Shell:

```
python manage.py shell
```

Виконайте наступні запити:

```
from store.models import *

# Отримати всі категорії
Category.objects.all()

# Знайти товар за артикулом
Product.objects.get(sku='SKU001')

# Отримати всі товари категорії "Ноутбуки"
Product.objects.filter(category__name='Ноутбуки')

# Отримати товари дорожче 30000 грн
Product.objects.filter(list_price__gt=30000)
```

Розширення адмін-панелі:

- Додайте фільтри для моделі `Product` за ціною
- Додайте можливість пошуку товарів за категорією
- Додайте сортування за різними полями

Середній рівень:

Складні запити ORM:

- Отримати всі замовлення за останній місяць
- Знайти найпопулярніший товар (за кількістю замовлень)
- Розрахувати середній чек замовлення
- Отримати список клієнтів, які зробили більше 2 замовлень

Створення власних методів моделі:

- Метод `get_orders_total()` для моделі `Customer`
- Метод `is_in_stock()` для моделі `Product`
- Метод `get_unpaid_amount()` для моделі `Order`

Високий рівень:

Агрегація та анотація:

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

# Товари з кількістю замовлень
Product.objects.annotate(orders_count=Count('order_items'))

# Клієнти з загальною сумою покупок
Customer.objects.annotate(total_spent=Sum('orders__items__unit_price'))
```

Оптимізація запитів:

- Використайте `select_related()` для оптимізації запитів з ForeignKey
- Використайте `prefetch_related()` для оптимізації запитів

6. Контрольні питання

1. Що таке ORM і які його переваги перед прямими SQL-запитами?
2. Яка різниця між методами `filter()` та `get()` в Django ORM?
3. Що таке міграції і навіщо вони потрібні?
4. Як працює каскадне видалення в Django (`on_delete`)?
5. Що таке `related_name` і навіщо він потрібен?
6. Яка різниця між `CharField` та `TextField`?
7. Що таке `verbose_name` та `verbose_name_plural`?
8. Як створити унікальний індекс на поле моделі?
9. Що таке `InlineModelAdmin` і коли його використовувати?
10. Як оптимізувати запити в Django ORM?

7. Корисні посилання

- Офіційна документація Django
- Django ORM QuerySet API
- Django Models
- Django Admin
- Django Migrations

8. Висновки

В ході виконання лабораторної роботи студенти навчилися:

- Встановлювати та налаштовувати Django проект

- Створювати моделі та визначати зв'язки між ними
- Генерувати та застосовувати міграції
- Наповнювати базу даних демонстраційними даними
- Налаштовувати адміністративну панель Django
- Виконувати запити до БД за допомогою Django ORM
- Оптимізувати роботу з базою даних