Національний університет біоресурсів і
природокористування України

Створення веб-застосунку за допомогою фреймворку Django. Основи роботи з проектом та базою даних.

Частина 2

Лектор: к.ф.-м.н., доцент Кириченко Віктор Вікторович

Вступ до Django

Що таке Django?

- Високорівневий Python фреймворк
- Створений для швидкої розробки
- Випущений у 2005 році
- Названий на честь джазового гітариста

Основні переваги

- Повний стек (batteries-included)
- Безпека за замовчуванням
- Масштабованість
- Чітка структура проекту

Архітектура

- Шаблон MVC (Model-View-Controller)
- У Django це MVT (Model-View-Template)
- ORM для роботи з БД
- Власна система шаблонів

Де використовується?

- Instagram, Pinterest, Disqus
- Системи управління контентом
- Соціальні мережі
- Електронні комерційні платформи

Архітектура Django

MVT (Model-View-Template) архітектура

Model
Структура даних

View
Логіка обробки

Template
Представлення

Як це працює?

1. Користувач надсилає запит через URL
2. URL dispatcher направляє запит до відповідного View
3. View взаємодіє з Model для отримання/збереження даних
4. View передає дані в Template для відображення
5. Template генерує HTML та повертає відповідь користувачу

MVT vs MVC

- MVC: Model-View-Controller
- Django View ≈ MVC Controller
- Django Template ≈ MVC View

Переваги архітектури

- Розділення відповідальності
- Легкість тестування
- Повторне використання коду

Створення Django проекту

Встановлення Django

```
# Встановлення за допомогою pip
pip install django

# Перевірка версії
python -m django --version
```

- Потрібен Python 3.8 або новіший
- Рекомендується використовувати віртуальне середовище

Створення проекту

```
# Створення нового проекту
django-admin startproject myproject

# Перехід до директорії проекту
cd myproject
```

- Створює базову структуру проекту
- Автоматично генерує файли конфігурації

Структура проекту

Основні файли:

- `manage.py` - утиліта управління
- `settings.py` - налаштування
- `urls.py` - маршрутизація URL
- `wsgi.py` - WSGI конфігурація

Запуск сервера:

```
python manage.py runserver
```

Сервер запускається за адресою `http://127.0.0.1:8000/`

Початкове налаштування

- Змінити `SECRET_KEY` у файлі settings.py
- Налаштувати `DEBUG = False` для продакшену
- Додати дозволені хости у `ALLOWED_HOSTS`
- Налаштувати базу даних у `DATABASES`

Структура Django проекту

Структура директорій та файлів

```
myproject/
  manage.py
  myproject/
    __init__.py
    settings.py
    urls.py
    wsgi.py
    asgi.py
```

Основні файли проекту

- **manage.py**
  Утиліта командного рядка для взаємодії з проектом
- **settings.py**
  Основний файл конфігурації проекту
- **urls.py**
  Оголошення URL-маршрутів проекту
- **wsgi.py/asgi.py**
  Точки входу для WSGI/ASGI-сумісних веб-серверів

Структура додатку (app)

```
myapp/
  __init__.py
  admin.py
  apps.py
  migrations/
  models.py
  tests.py
  views.py
  urls.py
```

Файли додатку

- `models.py` - моделі даних
- `views.py` - логіка обробки запитів
- `urls.py` - URL-маршрути додатку
- `admin.py` - налаштування адмін-панелі

Важливі директорії

- `migrations/` - файли міграцій БД
- `templates/` - HTML-шаблони
- `static/` - статичні файли (CSS, JS, зображення)
- `media/ – завантажені користувачами файли`

Створення Django додатків

Що таке додаток (app)?

- Модульна частина проекту з власною функціональністю
- Містить моделі, представлення, шаблони та URL
- Може бути використаний у різних проектах
- Сприяє повторному використанню коду

Створення додатку

```
# Створення нового додатку
python manage.py startapp myapp

# Додавання додатку до проекту
# у файлі settings.py
INSTALLED_APPS = [
  ...
  'myapp',
]
```

Структура додатку

Основні файли:

- `models.py` - моделі даних
- `views.py` - логіка обробки
- `urls.py` - маршрутизація
- `admin.py` - адмін-панель
- `tests.py` - тести

Додаткові файли:

- `apps.py` - конфігурація додатку
- `migrations/` - міграції БД
- `templates/` – HTML-шаблони
- `static/` – статичні файли

Інтеграція додатку

- Додати до `INSTALLED_APPS` у settings.py
- Підключити URL-маршрути у головному urls.py
- Виконати міграції для створення таблиць БД

Кращі практики

- Створюйте додатки з чітко визначеною функціональністю
- Дотримуйтесь принципу єдиної відповідальності
- Використовуйте іменні простори для URL

Вступ до моделей Django

Що таке моделі?

- Класи Python, що відображають структуру даних
- Кожна модель відповідає таблиці в базі даних
- Атрибути моделі - поля таблиці БД
- Наслідуються від `django.db.models.Model`

Приклад моделі

```python
from django.db import models

class Student(models.Model):
  """Модель для представлення студента"""
  first_name = models.CharField(max_length=50)
  last_name = models.CharField(max_length=50)
  email = models.EmailField()
  enrollment_date = models.DateField()
  is_active = models.BooleanField(default=True)

  def __str__(self):
    return f"{self.first_name} {self.last_name}"
```

Основні типи полів

Текстові поля:

- `CharField`
- `TextField`
- `EmailField`
- `URLField`

Числові поля:

- `IntegerField`
- `FloatField`
- `DecimalField`
- `BooleanField`

Дати та час:

- `DateField`
- `TimeField`
- `DateTimeField`

Опції полів

- `null=True` - дозволяє NULL в БД
- `blank=True` - дозволяє порожнє значення
- `default=значення` - значення за замовчуванням
- `unique=True` - унікальне значення

Метадані моделі

- Внутрішній клас `Meta`
- `db_table` - назва таблиці
- `ordering` - порядок сортування
- `verbose_name` - відображувана назва

Поля та зв'язки моделей Django

Типи зв'язків

- One-to-One (Один-до-одного)
  Кожен об'єкт однієї моделі пов'язаний з одним об'єктом іншої моделі
- One-to-Many (Один-до-багатьох)
  Один об'єкт моделі може мати багато пов'язаних об'єктів іншої моделі
- Many-to-Many (Багато-до-багатьох)
  Об'єкти однієї моделі можуть мати багато зв'язків з об'єктами іншої моделі

Приклад зв'язків

```python
class Author(models.Model):
  name = models.CharField(max_length=100)

class Book(models.Model):
  title = models.CharField(max_length=200)
  author = models.ForeignKey(Author, on_delete=models.CASCADE)

class Reader(models.Model):
  name = models.CharField(max_length=100)
  books = models.ManyToManyField(Book)
```

Поля для зв'язків

ForeignKey

- Визначає зв'язок один-до-багатьох
- Обов'язковий параметр `on_delete`
- Створює поле з ідентифікатором

OneToOneField

- Визначає зв'язок один-до-одного
- Також потребує `on_delete`
- Гарантує унікальність зв'язку

ManyToManyField

- Визначає зв'язок багато-до-багатьох
- Створює проміжну таблицю
- Може містити додаткові поля

Опції зв'язків

- `on_delete` - дія при видаленні пов'язаного об'єкта
- `related_name` - назва зворотного зв'язку
- `null=True` - дозволяє NULL значення
- `blank=True` - дозволяє порожнє значення

Робота зі зв'язками

- Доступ до пов'язаних об'єктів через атрибути
- Використання `select_related` для ForeignKey
- Використання `prefetch_related` для ManyToMany
- Фільтрація за зв'язками через `__`

Принципи Django ORM

Що таке ORM?

- Object-Relational Mapping (об'єктно-реляційне відображення)
- Техніка перетворення даних між несумісними системами типів
- Об'єкти Python ↔ таблиці бази даних
- Дозволяє працювати з БД без написання SQL-запитів

Як працює Django ORM?

- Моделі Django відображаються на таблиці БД
- Запити до БД формуються через QuerySet API
- ORM автоматично генерує SQL-запити
- Підтримує різні СУБД (SQLite, PostgreSQL, MySQL тощо)

Приклад роботи з ORM

Створення та отримання об'єктів:

```python
# Створення об'єкта
student = Student.objects.create(
  first_name="Іван",
  last_name="Петренко"
)

# Отримання всіх об'єктів
students = Student.objects.all()

# Фільтрація
active_students = Student.objects.filter(
  is_active=True
)
```

Оновлення та видалення:

```python
# Оновлення об'єкта
student = Student.objects.get(id=1)
student.first_name = "Петро"
student.save()

# Видалення об'єкта
student = Student.objects.get(id=2)
student.delete()

# Масове оновлення
Student.objects.filter(
  is_active=False
).update(is_active=True)
```

Переваги Django ORM

- Абстракція від конкретної СУБД
- Захист від SQL-ін'єкцій
- Простота використання
- Інтеграція з іншими компонентами Django

Обмеження ORM

- Менша гнучкість порівняно з чистим SQL
- Можлива менша продуктивність для складних запитів
- Складніші запити можуть бути менш читабельними
- Потреба вивчення API Django ORM

Міграції бази даних

Що таке міграції?

- Файли, що описують зміни в моделях даних
- Версіонована система керування схемою БД
- Дозволяють змінювати структуру БД без втрати даних
- Зберігаються у директорії `migrations/`

Основні команди

```
# Створення міграцій
python manage.py makemigrations

# Застосування міграцій
python manage.py migrate

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

Робочий процес міграцій

1. Зміна моделей
Внесення змін у файл models.py

2. Створення міграцій
Виконання makemigrations

3. Застосування змін
Виконання migrate

4. Перевірка
Перевірка результатів

Кращі практики

- Завжди перевіряйте міграції перед застосуванням
- Не редагуйте міграційні файли вручну
- Використовуйте `--dry-run` для тестування
- Створюйте резервні копії перед складними міграціями

Поширені проблеми

- Конфлікти міграцій при роботі в команді
- Втрата даних при зміні типів полів
- Проблеми зі зворотною сумісністю
- Складні міграції для великих таблиць

Налаштування панелі адміністратора Django

Що таке адмін-панель?

- Вбудований інтерфейс для управління даними
- Автоматично генерується на основі моделей
- Дозволяє CRUD-операції без кодування
- Гнучко налаштовується під потреби проекту

Створення суперкористувача

```
# Створення адміністратора
python manage.py createsuperuser
```

- Введіть ім'я користувача, email та пароль
- Адмін-панель доступна за адресою `/admin/`

Реєстрація моделей в адмін-панелі

Базова реєстрація:

```
# myapp/admin.py

from django.contrib import admin
from .models import Student

# Проста реєстрація
admin.site.register(Student)
```

Розширена реєстрація:

```
# myapp/admin.py

class StudentAdmin(admin.ModelAdmin):
  list_display = ('first_name', 'last_name', 'email')
  search_fields = ('first_name', 'last_name')
  list_filter = ('is_active',)

admin.site.register(Student, StudentAdmin)
```

Налаштування адмін-панелі

- `list_display` - поля для відображення
- `list_filter` - фільтри для списку
- `search_fields` - поля для пошуку
- `ordering` - порядок сортування

Користувацькі дії

- Створення власних дій для адмін-панелі
- Перевизначення шаблонів адмін-панелі
- Створення власних форм для редагування
- Додавання валідації даних у формах

Робота з панеллю адміністратора Django

Інтерфейс адмін-панелі

- Головна сторінка зі списком зареєстрованих моделей
- Списки об'єктів з можливістю пошуку та фільтрації
- Форми для створення та редагування об'єктів
- Історія змін для кожного об'єкта

CRUD-операції

- **Create** - створення нових об'єктів
- **Read** - перегляд списків та окремих об'єктів
- **Update** - редагування існуючих об'єктів
- **Delete** - видалення об'єктів

Налаштування відображення моделей

Основні опції ModelAdmin:

```
class StudentAdmin(admin.ModelAdmin):
  list_display = ('first_name', 'last_name')
  list_filter = ('is_active', 'enrollment_date')
  search_fields = ('first_name', 'last_name')
  ordering = ('last_name', 'first_name')
  list_per_page = 25
```

Робота з формами:

```
class StudentAdmin(admin.ModelAdmin):
  fields = ('first_name', 'last_name', 'email')
  exclude = ('is_active',)
  readonly_fields = ('enrollment_date',)
  fieldsets = (
    ('Основна інформація', {
      'fields': ('first_name', 'last_name')
    }),
  )
```

Користувацькі дії

- Створення власних дій для списку об'єктів
- Визначення дій через методи `actions`
- Масові операції над об'єктами

```
def make_active(modeladmin, request, queryset):
  queryset.update(is_active=True)
make_active.short_description = 'Активувати вибрані'
```

Розширені можливості

- Inline-редагування зв'язаних об'єктів
- Перевизначення шаблонів адмін-панелі
- Додавання медіа-файлів до моделей
- Створення власних віджетів для полів

Створення представлень (views) в Django

Що таке представлення (view)?

- Функція або клас, що обробляє HTTP-запити
- Приймає запит і повертає відповідь
- Відповідає за бізнес-логіку застосунку
- Взаємодіє з моделями для отримання даних

Структура представлення

```python
from django.http import HttpResponse

def my_view(request):
  # Логіка обробки запиту
  data = "Привіт, світе!"

  # Формування відповіді
  return HttpResponse(data)
```

Типи представлень

Функціональні представлення

```python
from django.shortcuts import render

def student_list(request):
  students = Student.objects.all()
  context = {
    'students': students
  }
  return render(request, 'students/list.html', context)
```

- Прості та зрозумілі
- Підходять для простої логіки

Класові представлення

```python
from django.views.generic import ListView

class StudentListView(ListView):
  model = Student
  template_name = 'students/list.html'
  context_object_name = 'students'
  paginate_by = 10
```

- Повторне використання коду
- Підходять для складних завдань

Основні методи класових представлень

- `get()` - обробка GET-запитів
- `post()` - обробка POST-запитів
- `get_context_data()` - формування контексту
- `get_queryset()` - отримання набору даних

Поширені класи представлень

- `ListView` - відображення списку об'єктів
- `DetailView` - відображення одного об'єкта
- `CreateView` - створення нового об'єкта
- `UpdateView` - редагування об'єкта

Передача даних у представлення

Отримання даних з моделей

- QuerySet - колекція об'єктів з бази даних
- `Model.objects.all()` - всі об'єкти
- `Model.objects.filter()` - фільтрація
- `Model.objects.get()` - один об'єкт

Приклад отримання даних

```
def student_list(request):
  # Отримання всіх студентів
  students = Student.objects.all()

  # Фільтрація активних студентів
  active_students =
Student.objects.filter(
    is_active=True
  )

  # Отримання одного студента
  student = Student.objects.get(id=1)
```

Оптимізація запитів

- `select_related` - для ForeignKey
- `prefetch_related` - для ManyToMany
- `only()` та `defer()` - вибіркові поля
- `annotate()` - агрегація даних

Контекст представлення

Передача контексту у шаблон

```
def student_detail(request, pk):
  student = Student.objects.get(pk=pk)
  courses = student.course_set.all()

  context = {
    'student': student,
    'courses': courses,
    'title': 'Деталі студента'
  }

  return render(request,
'students/detail.html', context)
```

Використання контексту в шаблоні

```
<h1>{{ title }}</h1>

<h2>{{ student.first_name }}</h2>

<h3>Курси:</h3>
<ul>
  {% for course in courses %}
    <li>{{ course.name }}</li>
  {% endfor %}
</ul>
```

Кращі практики

- Використовуйте значущі імена змінних
- Мінімізуйте кількість запитів до БД
- Використовуйте пагінацію для великих наборів даних
- Обробляйте винятки для `get()`

Основи шаблонів Django

Що таке шаблони?

- Текстові файли з HTML-розміткою та спеціальними тегами
- Відповідають за представлення даних користувачу
- Дозволяють відокремити логіку від відображення
- Зберігаються у директорії `templates/`

Структура шаблону

```
<!DOCTYPE html>
<html>
<head>
  <title>{{ title }}</title>
</head>
<body>
  <h1>{{ heading }}</h1>
  <p>{{ content }}</p>
</body>
</html>
```

Поширені фільтри

- `lower`, `upper` - зміна регістру
- `date` - форматування дати
- `length` - довжина списку/рядка
- `default` - значення за замовчуванням

Мова шаблонів Django (DTL)

Змінні

```
<p>{{ student.name }}</p>
<p>{{ student.email|lower }}</p>
<p>{{ student.enrollment_date|date:"d.m.Y"
}}</p>
```

- Вивід змінних через `{{ }}`
- Фільтри через `|`

Теги

```
{% if student.is_active %}
  <p>Активний студент</p>
{% else %}
  <p>Неактивний</p>
{% endif %}

{% for course in courses %}
  {{ course.name }}
{% endfor %}
```

- Логіка через `{% %}`
- Умови, цикли, блоки

Коментарі

```
{# Однорядковий коментар #}

{% comment %}
Багаторядковий
коментар
{% endcomment %}
```

- Не відображаються у HTML
- Для документації

Контекст та змінні шаблонів

Передача даних у шаблон

- Контекст - словник Python з даними для шаблону
- Передається третім аргументом у `render()`
- Може містити змінні, функції, об'єкти
- Доступний у шаблоні через імена ключів

Приклад передачі контексту

```
def course_detail(request, pk):
  course = Course.objects.get(pk=pk)
  students = course.student_set.all()

  context = {
    'course': course,
    'students': students,
    'now': timezone.now()
  }

  return render(request,
'courses/detail.html', context)
```

Контекст-процесори

- Функції, що додають дані до контексту
- Працюють автоматично для всіх шаблонів
- Вбудовані: `debug`, `request`, `media`
- Можна створювати власні процесори

Використання змінних та фільтрів

Доступ до змінних

```
<h1>{{ course.name }}</h1>
<p>{{ course.description }}</p>

<ul>
  {% for student in students %}
    <li>{{ student.name }}</li>
  {% endfor %}
</ul>
```

Фільтри

```
<p>{{ course.name|upper }}</p>
<p>{{ now|date:"d.m.Y H:i" }}</p>
<p>{{ course.description|truncatewords:30 }}
</p>
<p>{{ students|length }} студентів</p>
```

Кращі практики

- Використовуйте значущі імена змінних
- Підготуйте дані у view, а не у шаблоні
- Використовуйте фільтри для форматування
- Уникайте складної логіки в шаблонах

Наслідування шаблонів

Що таке наслідування?

- Механізм повторного використання коду шаблонів
- Дозволяє створювати базові шаблони з загальною структурою
- Дочірні шаблони можуть перевизначати частини базового
- Сприяє підтримці єдиного стилю сайту

Базовий шаблон (base.html)

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}Мій сайт{%
endblock %}</title>
</head>
<body>
  <header>
    {% block header %}{% endblock %}
  </header>
  <main>
    {% block content %}{% endblock %}
  </main>
</body>
</html>
```

Вкладені блоки

- Блоки можуть бути вкладеними один в одного
- Можна створювати складні ієрархії шаблонів
- Дозволяє гнучко керувати структурою сторінки

```
{% block content %}
  <div class="container">
    {% block main %}{% endblock %}
    {% block sidebar %}{% endblock %}
  </div>
{% endblock %}
```

Розширення базового шаблону

Дочірній шаблон

```
{% extends "base.html" %}

{% block title %}Сторінка студента{% endblock %}

{% block content %}
  <h1>{{ student.name }}</h1>
  <p>{{ student.email }}</p>
{% endblock %}
```

Використання super()

```
{% extends "base.html" %}

{% block header %}
  <div class="logo">
    {{ block.super }}
    <img src="logo.png">
  </div>
{% endblock %}
```

Кращі практики

- Створюйте логічну ієрархію шаблонів
- Розміщуйте шаблони у піддиректоріях
- Використовуйте значущі імена для блоків
- Використовуйте `{% include %}` для повторюваних елементів

Маршрутизація URL в Django

Що таке маршрутизація URL?

- Процес зіставлення URL-адрес з функціями представлення
- Визначає, який код виконувати для кожного запиту
- Реалізується через файл `urls.py`
- Підтримує параметри в URL

Приклад файлу urls.py

```
from django.urls import path
from . import views

urlpatterns = [
  path('', views.index, name='index'),
  path('students/', views.student_list, name='student_list'),
  path('students/<int:pk>/', views.student_detail,
name='student_detail'),
  path('about/', views.about, name='about'),
]
```

Типи URL-шаблонів

Статичні URL

```
path('about/', views.about)
```

- Фіксовані шляхи без параметрів
- Прості у визначенні

URL з параметрами

```
path('student/<int:id>/',
views.student_detail)
```

- Передають параметри у view
- Підтримують типи: int, str, slug, uuid

Регулярні вирази

```
re_path(r'^student/(?P<id>\d+)/$',
views.student_detail)
```

- Складніші шаблони
- Більше гнучкості

Включення URL-адрес додатків

- Використання `include()` для підключення URL-адрес додатків
- Дозволяє організувати URL-адреси за модулями

```
# Головний urls.py
path('students/', include('students.urls'))
```

Кращі практики

- Використовуйте іменовані URL-адреси
- Завершуйте URL-адреси слешем `/`
- Використовуйте `reverse()` та `{% url %}` для генерації URL
- Створюйте логічну структуру URL-адрес

Шаблони URL та представлення

Зв'язок URL з представленнями

- Кожен URL-шлях пов'язаний з функцією представлення
- Параметри з URL передаються як аргументи у view
- URL-шляхи можуть бути іменованими для зручності
- Використання `include()` для підключення URL додатків

Функція reverse()

- Генерує URL-адресу за іменем шаблону
- Уникнення жорсткого кодування URL

```
from django.urls import reverse

# У view
url = reverse('student_detail', kwargs=
{'pk': student.id})

# У шаблоні
<a href="{% url 'student_detail'
pk=student.id %}">Деталі</a>
```

Приклад зв'язку URL з view

```
# urls.py
urlpatterns = [
  path('students/<int:pk>/',
views.student_detail,
name='student_detail'),
]

# views.py
def student_detail(request, pk):
  student = Student.objects.get(pk=pk)
  context = {'student': student}
  return render(request,
'students/detail.html', context)
```

Передача параметрів з URL

Типи параметрів

```
# Цілі числа
path('student/<int:id>/', views.student_detail)

# Рядки
path('category/<slug:slug>/', views.category_detail)

# UUID
path('post/<uuid:uuid>/', views.post_detail)
```

- int - цілі числа
- str - рядки (за замовчуванням)
- slug - URL-дружні рядки

Використання параметрів у view

```
def student_detail(request, pk):
  # Отримання студента за ID
  student = Student.objects.get(pk=pk)

  # Обробка помилки, якщо студента не існує
  try:
    student = Student.objects.get(pk=pk)
  except Student.DoesNotExist:
    raise Http404("Студент не знайдений")
```

Огляд практичної вправи

Мета вправи

- Закріпити теоретичні знання на практиці
- Створити простий веб-застосунок з нуля
- Відпрацювати основні етапи розробки
- Отримати досвід роботи з Django

Завдання вправи

1. Створення моделі даних
2. Проведення міграцій бази даних
3. Створення представлення (view)
4. Розробка шаблону для відображення даних
5. Налаштування маршрутизації URL

Очікуваний результат

Функціональність

- Сторінка зі списком об'єктів
- Сторінка детальної інформації
- Можливість додавання через адмін-панель
- Коректна робота посилань

Навички

- Робота з моделями та ORM
- Виконання міграцій
- Створення представлень
- Робота з шаблонами
- Налаштування URL-маршрутів

Тривалість

- Загальна тривалість: 20-25 хвилин
- Створення моделі: 5 хвилин
- Міграції: 3 хвилини
- Представлення та шаблони: 10 хвилин
- URL-маршрути: 2 хвилини

Поради для успіху

- Дотримуйтесь покрокової інструкції
- Перевіряйте результат на кожному етапі
- Не соромтеся ставити запитання
- Експериментуйте з додатковими полями

Вправа: Створення моделі

Крок 1: Визначення моделі

1. Відкрийте файл `models.py` у вашому додатку
2. Імпортуйте модуль моделей Django
3. Створіть клас, що наслідує `models.Model`
4. Визначте поля моделі з відповідними типами

Приклад моделі "Студент"

```python
from django.db import models

class Student(models.Model):
  first_name = models.CharField(max_length=50)
  last_name = models.CharField(max_length=50)
  email = models.EmailField(unique=True)
  enrollment_date = models.DateField(auto_now_add=True)
  is_active = models.BooleanField(default=True)

  def __str__(self):
    return f"{self.first_name} {self.last_name}"
```

Крок 2: Вибір типів полів

Текстові поля

- `CharField` - короткий текст
- `TextField` - довгий текст
- `EmailField` - email адреса

Числові поля

- `IntegerField` - цілі числа
- `FloatField` - дробові числа
- `BooleanField` - так/ні

Дати та час

- `DateField` - дата
- `DateTimeField` - дата і час
- `TimeField` - час

Крок 3: Налаштування опцій

- `max_length` - максимальна довжина
- `unique=True` - унікальне значення
- `default=значення` - значення за замовчуванням
- `auto_now_add=True` - автоматично при створенні

Крок 4: Перевірка моделі

- Перевірте синтаксичну правильність коду
- Переконайтесь, що всі поля мають відповідні типи
- Додайте метод `__str__()` для зручного відображення
- Збережіть файл перед переходом до міграцій

Вправа: Виконання міграцій

Крок 1: Створення міграцій

1. Відкрийте термінал у кореневій папці проекту
2. Виконайте команду для створення міграцій
3. Перевірте створені файли міграцій
4. Переконайтесь, що зміни відповідають моделі

Команди для міграцій

```
# Створення міграцій
python manage.py makemigrations

# Застосування міграцій
python manage.py migrate

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

Крок 2: Структура файлів міграцій

Директорія міграцій

```
myapp/
  migrations/
    __init__.py
    0001_initial.py
    0002_...
```

- Файли з префіксом номера та опису
- Автоматично генеруються Django

Зміст файлу міграції

```
class Migration(migrations.Migration):
  initial = True

  dependencies = [
  ]

  operations = [
    migrations.CreateModel(
      name='Student',
      fields=[...]
    )
  ]
```

Крок 3: Застосування міграцій

- Виконайте команду `python manage.py migrate`
- Перевірте вивід - він показує застосовані міграції
- Переконайтесь, що таблиці створено у базі даних
- Використовуйте `showmigrations` для перевірки статусу

Поширені проблеми

- Несумісність міграцій при роботі в команді
- Проблеми зі зміною типів полів з даними
- Видалення застосованих міграцій
- Рішення: `--fake` або ручне редагування

Вправа: Відображення даних у шаблонах (Частина 1)

Крок 1: Створення представлення

1. Відкрийте файл `views.py` у вашому додатку
2. Імпортуйте необхідні модулі
3. Створіть функцію представлення
4. Отримайте дані з моделі та передайте у шаблон

Приклад представлення

```python
from django.shortcuts import render
from .models import Student

def student_list(request):
  students = Student.objects.all()
  context = {
    'students': students
  }
  return render(request, 'students/list.html', context)
```

Крок 2: Створення шаблону

Структура директорій

```
myapp/
  templates/
    students/
      list.html
      detail.html
```

- Створіть директорію `templates`
- У ній створіть піддиректорію з назвою додатку

Приклад шаблону

```
<h1>Список студентів</h1>
<ul>
  {% for student in students %}
    <li>
      {{ student.first_name }} {{ student.last_name }}
    </li>
  {% endfor %}
</ul>
```

Вправа: Відображення даних у шаблонах (Частина 2)

Крок 3: Налаштування URL

- Створіть файл `urls.py` у додатку
- Визначте URL-шляхи для представлень

```
# myapp/urls.py
from django.urls import path
from . import views

urlpatterns = [
  path('', views.student_list, name='student_list'),
]
```

Крок 4: Підключення URL

- Відкрийте головний файл `urls.py` проекту
- Додайте `include()` для підключення URL додатку

```
# project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
  path('admin/', admin.site.urls),
  path('students/', include('myapp.urls')),
]
```

Кращі практики розробки на Django

Структура проекту

- Дотримуйтесь модульної структури додатків
- Створюйте додатки з чітко визначеною функціональністю
- Використовуйте віртуальні середовища
- Розділяйте налаштування для розробки та продакшену

Робота з моделями та БД

- Використовуйте значущі імена полів та моделей
- Оптимізуйте запити до БД (select_related, prefetch_related)
- Використовуйте індекси для часто використовуваних полів
- Не зберігайте великі файли в базі даних

Безпека

Захист від CSRF

- Використовуйте {% csrf_token %} у формах
- Налаштуйте CSRF middleware

Захист від XSS

- Довіряйте автоматичному екрануванню
- Використовуйте |safe тільки за потреби

Захист паролів

- Використовуйте вбудовані системи аутентифікації
- Не зберігайте паролі у відкритому вигляді

Якість коду

- Дотримуйтесь PEP 8 для Python коду
- Пишіть документацію та коментарі
- Використовуйте систему контролю версій (Git)
- Створюйте тести для вашого коду

Продуктивність

- Використовуйте кешування (cache)
- Оптимізуйте статичні файли (CSS, JS, зображення)
- Налаштуйте стиснення та CDN
- Використовуйте асинхронні завдання (Celery)

Висновки

Основні тези лекції

1. **Django - потужний фреймворк** для створення веб-застосунків, що надає готові рішення для більшості завдань розробки.
2. **Архітектура MVT** (Model-View-Template) дозволяє чітко розділити логіку роботи з даними, обробку запитів та відображення.
3. **ORM Django** спрощує роботу з базами даних, дозволяючи працювати з об'єктами Python замість написання SQL-запитів.
4. **Міграції** забезпечують контрольовані зміни структури бази даних при розвитку проекту.
5. **Адміністративна панель** Django надає зручний інтерфейс для керування даними без необхідності створювати додатковий код.
6. **Система шаблонів** дозволяє створювати гнучкі та безпечні інтерфейси користувача з можливістю повторного використання коду.
7. **Маршрутизація URL** забезпечує гнучке керування переходами між сторінками веб-застосунку.
8. **Дотримання кращих практик** розробки на Django дозволяє створювати надійні, безпечні та масштабовані веб-застосунки.