Rozbudowa modelu danych

Zajęcia Prymus 2020

Agnieszka Rudnicka rudnicka@agh.edu.pl

Rozbudowa modelu danych
- Relacyjne bazy danych
  - Problem z polem autora w obecnym modelu
  - Relacje i klucz obcy
  - Graficzna reprezentacja modelu danych
- Ulepszamy nasze modele
  - Model autora
    - Pole z obrazem ImageField
    - Etykieta verbose_name
    - Meta-dane modelu
  - Poprawki w modelu książek
    - Tworzymy migracje
    - Aplikujemy migrację
    - Konfiguracja plików MEDIA
    - Kontrola
    - Rejestrujemy autora w panelu administracyjnym
    - Tworzymy autorów przez panel admina
  - Recenzje książek
    - Model recenzji
    - Migracje modelu recenzji
- Nowe widoki dla użytkowników
  - Widoki klasowe
    - Przykładowy widok klasowy
    - Widok klasowy listy książek
    - Podpinanie widoków klasowych w urls.py
- Dalsze prace

Relacyjne bazy danych

Do tej pory nasza aplikacja posiadała dość prosty i ubogi model danych. Była to tylko pojedyncza tabela przechowująca książki (`class Book(models.Model):` w pliku `books/models.py`).

Pora na utworzenie oddzielnego modelu na autorów oraz recenzje.

Problem z polem autora w obecnym modelu

Wróćmy do pliku z naszymi modelami, `books/models.py`. Aktualnie autor jest polem tekstowym.

```
class Book(models.Model):
    ...
    author = models.CharField(null=True, max_length=128)
```

Jednak takie rozwiązanie szybko robi się problematyczne:

- jeśli mamy kilka książek tego samego autora, to informacje o nim się powtarzają w wielu wpisach (redundancja danych), *mówiąc prościej - marnotrawstwo miejsca przez powtarzanie tych samych informacji*
- jest większa szansa popełnienia błędu i powstania różnych zapisów tego samego imienia/nazwiska, co później może się przekładać na problemy z wyszukaniem wszystkich książek danego autora
- przy aktualizacji informacji o autorze trzeba zaktualizować wszystkie książki, które były przez niego napisane, co się przekłada na N edycji zamiast jednej
- i inne...

Relacje i klucz obcy

Jedną z podstawowych zalet baz danych których używamy (SQLite, PostgreSQL i innych) jest możliwość tworzenia relacji między modelami.

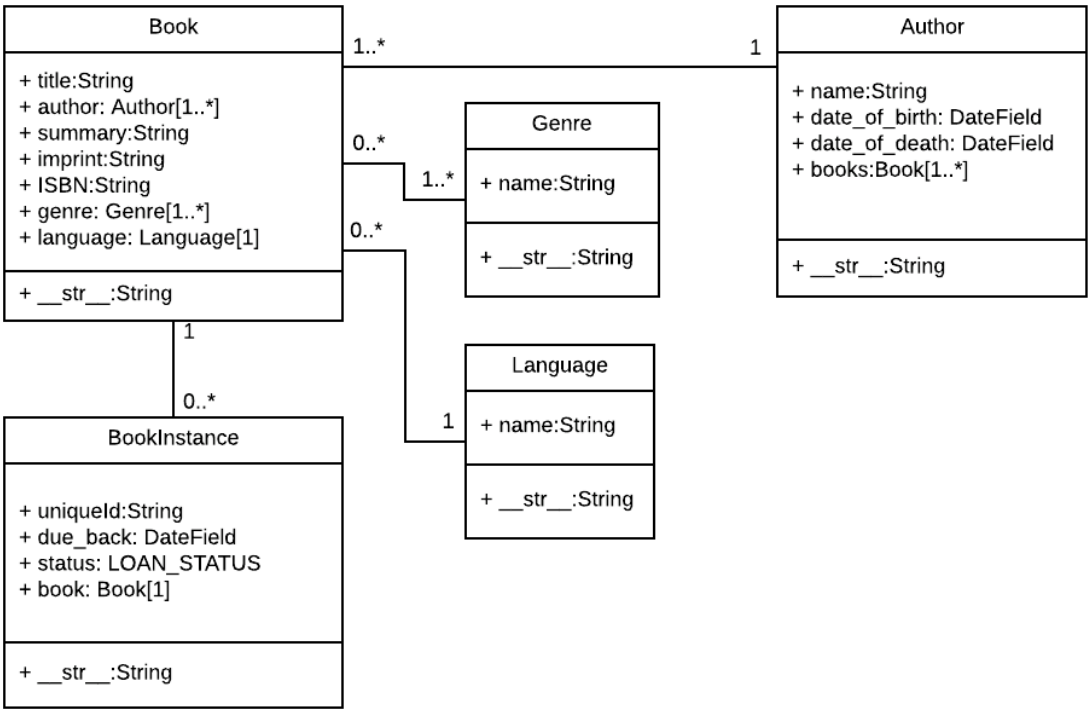
Biorąc na warsztat przykład z autorem i książkami, rozwiązaniem jakie często znajdziemy w praktyce jest wydzielenie osobnej tabeli na dane o autorach. W ten sposób będziemy mieli książki w jednej tabeli a autorów w drugiej. Unikniemy redundancji danych, problemów z aktualizowaniem wielu wpisów i innych.

Bazy relacyjne pozwalają zdefiniowanie specjalnego pola, w którym będzie przechowywany identyfikator wiersza z innej, "obcej" tabeli. Stąd też polska nazwa "klucz obcy" i angielska "foreign key".

Ten klucz obcy, to nic innego jak pole `id`, które jest automatycznie definiowane przez framework Django dla każdego modelu (jeśli użytkownik nie zdefiniuje własnego). `id` jest też często nazywany "kluczem głównym". Oznacza to, że identyfikuje on rekordy/wiersze będąc unikatowym i jednoznacznym. Tak jak numer PESEL się nie powtarza i zawsze identyfikuje jednego człowieka, tak **klucz główny identyfikuje jeden i tylko jeden wpis w tabeli bazy danych**.

Graficzna reprezentacja modelu danych

To jak przechowujemy dane w bazie często jest o wiele bardziej skomplikowane niż jedna, czy dwie tabele. Aby było łatwiej zrozumieć co z czym się łączy tworzy się modele danych np przy pomocy UML (Unified Modeling Language).



Powyższy diagram UML przedstawia coś podobnego do tego co będziemy tworzyć [źródło].

Ulepszamy nasze modele

Stwórzmy osobny model na autora i wykorzystajmy mechanizmy baz relacyjnych do stworzenia powiązać autora z książkami.

Model autora

W pliku `books/models.py` gdzie opisaliśmy uprzednio model książki, dodajmy teraz model autora:

```
class Author(models.Model):
    first_name = models.CharField(verbose_name="imię", max_length=100)
    last_name = models.CharField(verbose_name="nazwisko", max_length=100)
    about = models.TextField(verbose_name="o autorze", blank=True)
    photo = models.ImageField(verbose_name="zdjęcie", blank=True)

    class Meta:
        ordering = ["last_name", "first_name"]
        verbose_name = "autor"
        verbose_name_plural = "autorzy"

    def __str__(self):
        return "Autor: " + self.first_name + " " + self.last_name
```

Pole z obrazem ImageField

Mamy tutaj model zawierający pole na imię, nazwisko, opis oraz zdjęcie autora. To ostatnie jest szczególnie ciekawe. Django pozwala nam tworzyć pola, które przechowują ścieżkę do pliku. Rzadko kiedy przechowuje się pliki wgrane przez użytkowników w bazie danych. Django domyślnie w polu `ImageField` przechowuje informacje o nazwie pliku. Ustawienia gdzie tych plików szukać będą już specyficzne dla projektu, czasem nawet serwera.

Etykieta verbose_name

Kolejna nowością jest `verbose_name`, które zostało zdefiniowane dla każdego pola. Jest to coś w rodzaju domyślnej etykiety wyświetlanej jeśli żadna inna nie została zapewniona. Dzięki temu zabiegowi, w panelu admina zamiast angielskich nazw zmiennych zobaczymy polskie etykiety.

Meta-dane modelu

Ostatnim dodatkiem jest podklasa `class Meta`. Tutaj definiuje się meta-dane dotyczące modelu, między innymi domyślne sortowanie elementów. Pozwala to zapewnić kolejność zwracanych danych z bazy, co jest szczególnie ważne przy widokach z paginacją/stronicowaniem. No i wprowadza odrobinę ładu, bo sortowanie po nazwisku/imieniu jest dla nas (ludzi) naturalne.

Poprawki w modelu książek

Skoro już wzbogacamy nasz model o etykiety, dodajmy je również do modelu książki:

```
class Book(models.Model):
    title = models.CharField(verbose_name="tytuł", max_length=100)
    short_description = models.TextField(verbose_name="opis")
    published_at = models.DateField(verbose_name="data publikacji")

    # !!!
    author = models.ManyToManyField(to="books.Author", verbose_name="autorzy",
related_name="books")

    class Meta:
        ordering = ["title"]
        verbose_name = "książka"
        verbose_name_plural = "książki"

    def __str__(self):
        return "Książka: " + self.title
```

Przy okazji musimy zmienić również pole `author`. Będzie ono teraz kluczem obcym, co jest odzwierciedlone za pomocą pola `ForeignField()` w frameworku Django.

Tworzymy migracje

Na początek wykonajmy polecenie, które pozwala podglądnąć jaka migracja będzie wygenerowana. Niestety zakończy się ono błędem:

```
manage.py makemigrations --dry-run -v 3

SystemCheckError: System check identified some issues:

ERRORS:
books.Author.photo: (fields.E210) Cannot use ImageField because Pillow is not
installed.
        HINT: Get Pillow at https://pypi.org/project/Pillow/ or run command "python -m
pip install Pillow".
```

Doinstalujmy więc bibliotekę wymaganą do obsługi obrazów i pół `ImageField`.

```
pip install Pillow
```

Nie zapomnijmy dodać też biblioteki do pliku wymagań `requirements.txt` !

Teraz skrypt `manage.py makemigrations` powinien się pomyślnie zakończyć.

```
manage.py makemigrations

Migrations for 'books':
  books\migrations\0003_auto_0000000_0000.py
    - Create model Author
    - Change Meta options on book
    - Remove field author from book
    - Alter field published_at on book
    - Alter field short_description on book
    - Alter field title on book
    - Add field author to book
```

Aplikujemy migrację

Żeby odpowiednie tabele zostały utworzone, a istniejące zaktualizowane trzeba standardowo zaaplikować migrację.

`manage.py migrate`

```
manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, books, contenttypes, sessions
Running migrations:
  Applying books.0003_auto_0000000_0000... OK
```

Konfiguracja plików MEDIA

Zanim sprawdzimy nasze zmiany dodajmy jeszcze konfigurację gdzie mają być wgrywane pliki "media", czyli obrazy wgrane do pól typu `FileField` oraz `ImageField`.

Na samym końcu pliku `settings.py` w katalogu konfiguracji projektu odszukajmy linijki dotyczące plików statycznych i dodajmy jedną z `MEDIA_ROOT`:

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.0/howto/static-files/

STATIC_URL = '/static/'
MEDIA_ROOT = os.path.join(BASE_DIR, '/media/')  # <-- nowe

# Activate Django-Heroku.
django_heroku.settings(locals())
```

Dzięki temu pliki statyczne będą wgrywane do podkatalogu `/media` w głównym folderze z naszymi plikami projektu.

Kontrola

Uruchommy aplikację i zaglądnijmy do panelu administracyjnego.

Na pierwszy rzut oka niewiele się zmieniło. Są jedynie etykiety po polsku. Niestety nie ma śladu po modelu autora. Ale czy na pewno? Zobaczmy widok tworzenia nowej książki.

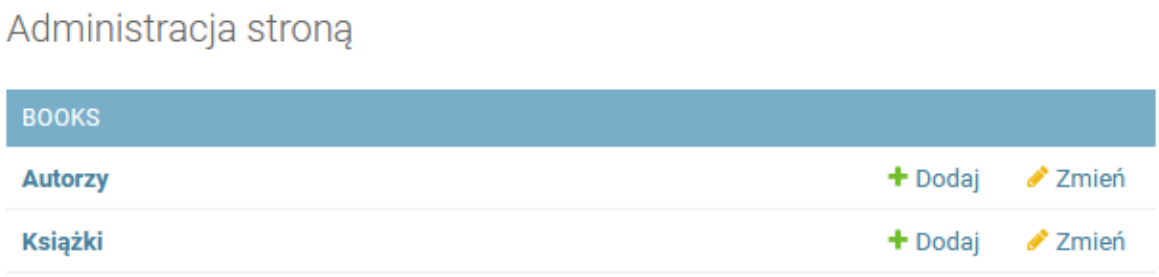
Pojawiło się pole wielokrotnego wyboru, jednak nie można nic wybrać ani dodać... bowiem nie zarejestrowaliśmy naszego nowego modelu jako edytowalnego przez panel administracyjny.

Rejestrujemy autora w panelu administracyjnym

Udajmy się do `books/admin.py`.

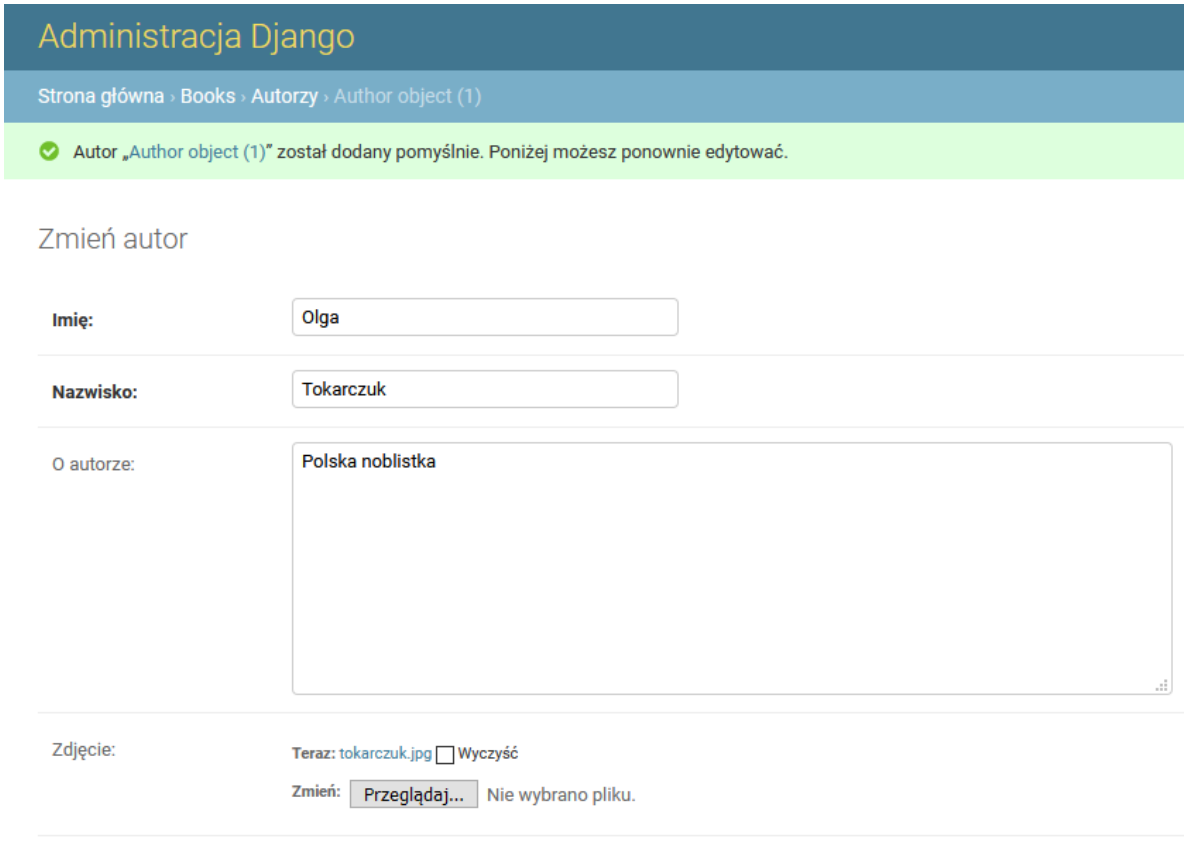
```
from django.contrib import admin

# Register your models here.
from books.models import Book, Author  # Author dodany

admin.site.register(Book)
admin.site.register(Author)  # nowe
```

Zapiszmy plik i odświeżmy stronę w przeglądarce (z panelem administracyjnym). Zaraz obok książek powinna się pojawić sekcja z autorami.



Tworzymy autorów przez panel admina

Możemy już tworzyć autorów razem ze zdjęciami.



Stwórzmy jeszcze kilku autorów i przejdźmy do widoku szczegółów jakiejś książki.

Możemy teraz do każdej książki przypisać autora. A dokładniej musimy przypisać przynajmniej jednego, ponieważ nie oznaczyliśmy pola autora jako `blank=True` co by pozwalało na pozostawienie go pustym.

Recenzje książek

Mamy już model autora i książki, które są ze sobą powiązane. Pora na model recenzji oraz widoki szczegółów, które wyświetlą te dane użytkownikom. W tej chwili tylko administratorzy je widzą i mogą modyfikować.

Model recenzji

W pliku `books/models.py`

```
class Review(models.Model):
    book = models.ForeignKey(to=Book, verbose_name="recenzowana książka",
on_delete=models.CASCADE)
    author = models.CharField(verbose_name="autor recenzji", max_length=250)
    content = models.TextField(verbose_name="treść recenzji")
    is_recommended = models.BooleanField(verbose_name="polecam innym")

    class Meta:
        verbose_name = "recenzja"
        verbose_name_plural = "recenzje"
```

Model zawiera:

- relację (konkretniej klucz obcy) do książki;
- pole tekstowe na podpis użytkownika-autora recenzji;
- pole tekstowe na recenzję;
- pole prawda/fałsz, które domyślnie będzie reprezentowane przez checkbox, czy użytkownik poleca książkę.

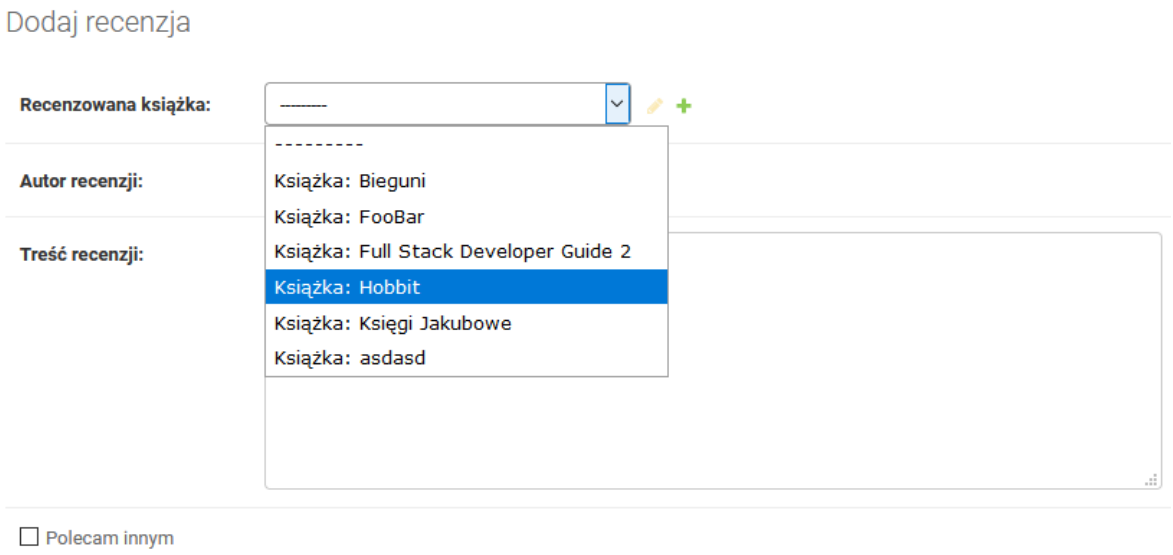
Migracje modelu recenzji

Następnym krokiem jest wygenerowanie migracji i zaaplikowanie jej. Nie zapomnijmy również dodać do panelu administracyjnego!

```
manage.py makemigrations

Migrations for 'books':
  books\migrations\0004_review.py
    - Create model Review
```

```
manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, books, contenttypes, sessions
Running migrations:
  Applying books.0004_review... OK
```

Jeśli wszystko poszło jak wyżej i dodaliśmy model `Review` do panelu administracyjnego w `books/admin.py` powinniśmy móc zarządzać recenzjami przez panel administracyjny.



Nowe widoki dla użytkowników

Widoki klasowe

Mechanizmem, którego do tej pory nie używaliśmy są widoki klasowe. W Internecie można je znaleźć pod nazwą "Class Based Views".

Polecam stronę z przeglądem widoków klasowych: https://ccbv.co.uk/ a także dokumentację: https://docs.djangoproject.com/en/3.0/topics/class-based-views/intro/

W skrócie - jest to podejście do pisania widoków z wykorzystaniem klas. Podobnie jak klasy definiują nam modele, tutaj klasy definiują widoki. Jest to proste i wyręcza nas z pisania części powtarzalnego kodu.

Przykładowy widok klasowy

Załóżmy, że mamy widok, który w zależności od metody zwraca 2 różne rzeczy (podobnie jak było z formularzem rejestracji na zajęciach o użytkownikach).

```
from django.http import HttpResponse

def my_view(request):
    if request.method == 'GET':
        # ...
        return HttpResponse('result')
    if request.method == 'POST':
        # ...
        return HttpResponse('another result')
```

Gdybyśmy wykorzystali widoki klasowe, to można to zapisać tak jak poniżej.

```
from django.http import HttpResponse
from django.views import View

class MyView(View):
    def get(self, request):
        # ...
        return HttpResponse('result')

    def post(self, request):
        # ...
        return HttpResponse('another result')
```

Na pierwszy rzut oka może nie wydawać się to krótsze, lub w jakikolwiek sposób lepsze. Zauważmy jednak, że zamiast pisać jedną dłuższą funkcję, która kolejno sprawdza typ metody zapytania i potem wykonuje kod, tworzymy jedynie klasę z dwoma metodami (`def get()` oraz `def post()`), które zawierają tylko ten kod, który dotyczy danego przypadku.

Widok klasowy listy książek

Może następny przykład będzie bardziej przekonujący, oto lista książek dotychczas:

```
from django.shortcuts import render
from books.models import Book

def book_list(request):
    books = Book.objects.all()
    context = {
        "books": books,
    }
    return render(request, template_name="book_list.html", context=context)
```

A to lista książek jako widok klasowy:

```
from django.views.generic import ListView
from books.models import Book

class BookListView(ListView):
    model = Book
```

Et voilà!

Django wyciągnie domyślnie wszystkie książki, bo podaliśmy model `Book` i poszuka szablony HTML na podstawie nazwy tego modelu. W tym przypadku będzie to `book_list.html`. Jeśli nasz szablon nazywa się inaczej możemy podać dodatkowe pole `template_name`, o tak:

```
class BookListView(ListView):
    model = Book
    template_name = "book_list.html"
```

Podpinanie widoków klasowych w urls.py

Aby wykorzystać taki widok klasowy z `urls.py` musimy go zaimportować a następnie wywołać funkcję `as_view()`.

```
from myapp.views import MyView

urlpatterns = [
    path('about/', MyView.as_view()),
]
```

Dzieje się tak, ponieważ Django oczekuje, że w `urlpatterns` znajdzie się lista funkcji, które można wywołać z zapytaniem, które przyszło od użytkownika. Dlatego każda klasa dziedzicząca po `View`, `ListView` i innych klasach widoków ma specjalną metodę `as_view()`, która zwraca specjalną funkcje przyjmującą jako argument zapytanie, tak jak w przypadku naszych małych funkcji widoków.

Dalsze prace

W następnych krokach zróbmy rzeczy, które już umiemy:

- widok listy autorów;
- widok listy recenzji;
- linki, które pozwolą na przechodzenie między recenzjami, autorami i książkami.