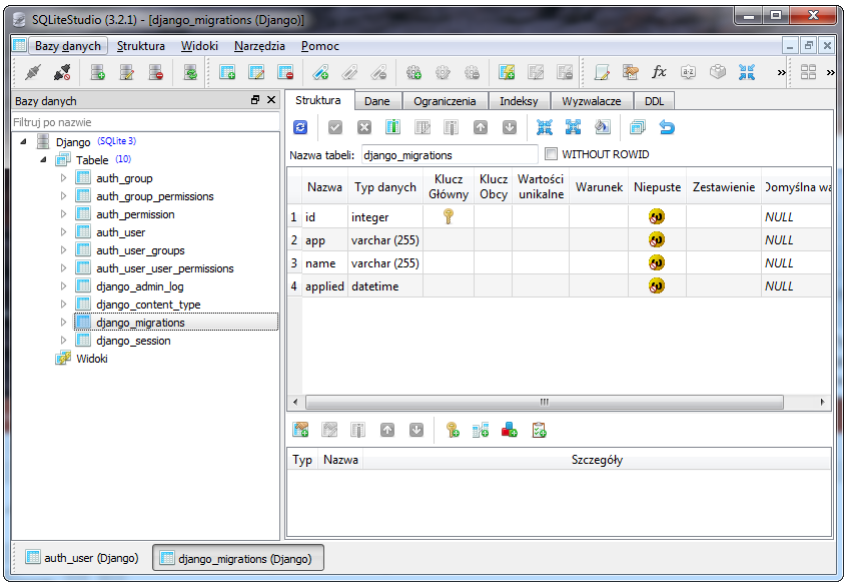
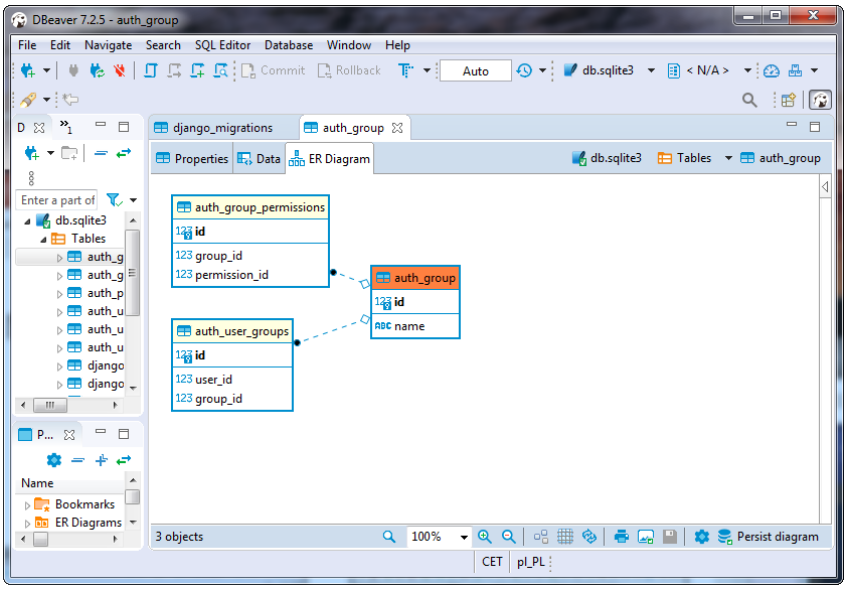
Django

Narzędzia SQLite

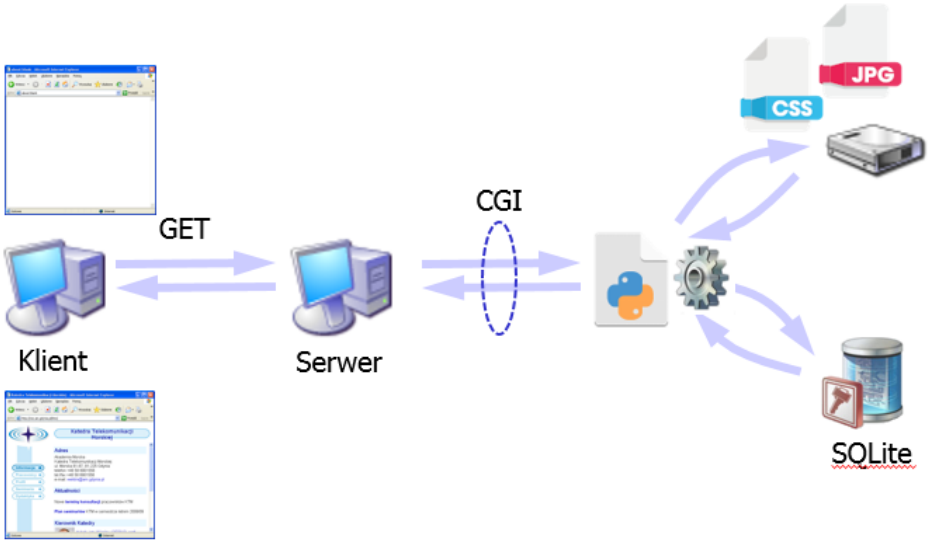
- SQLiteStudio
  Darmowy program do zarządzania bazami danych SQLite;
  Ma narzędzia exportu/importu baz danych do różnych formatów danych (XML, JSON, …); Bez wersji instalacyjnej ("portable"):
  https://www.sqlitetutorial.net/download-install-sqlite/



- DBeaver Community
  Darmowa wersja rozbudowanego programu do zarządzania różnymi typami baz danych, w tym SQLite; Jest dostępny zarówno w wersji "portable", jak i z instalatorem
  https://dbeaver.io/

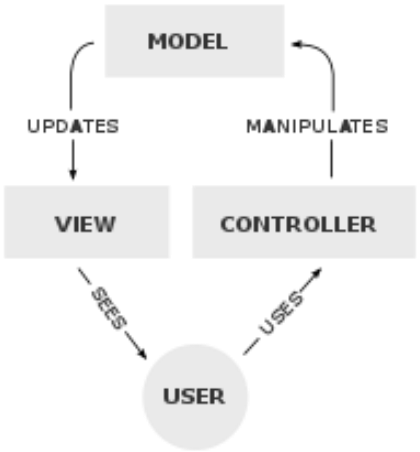


Dynamiczne strony WWW – CGI



Model MVC

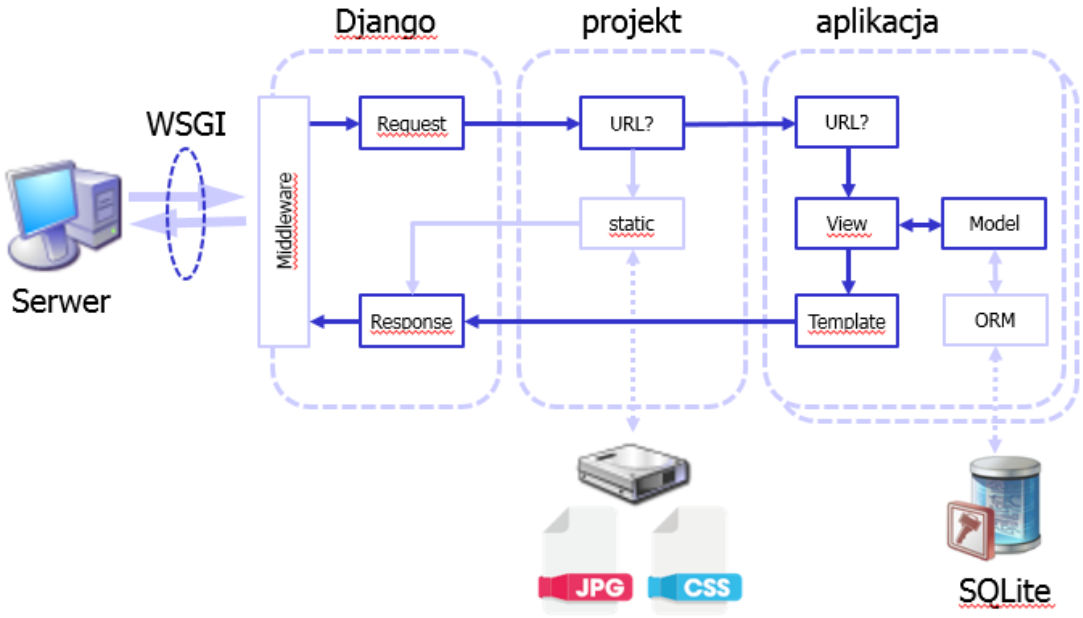
- Django jest zbudowany w oparciu o wariant modelu MVC (Model-View-Controller):
  – Model (model) – logika biznesowa
  – View (widok) – sposób prezentacji danych w GUI
  – Controller (kontroler) – pośredniczy między modelem a widokiem (aktualizacja modelu oraz odświeżanie widoków)
- Model MVC separuje logikę aplikacji od interfejsu użytkownika – umożliwia zmianę części modelu (np. bazy danych) bez zmian w widoku i na odwrót



Warianty modelu MVC

- MVP (model-view-presenter)
- MVVC (model-view-view model)
- Django: MVT (model-view-template)
  Sugestia z dokumentacji (formalnie takiego modelu nie ma):
  – Model (model) – klasy z ORM i logiką biznesową,
  – View (widok) – wybiera dane z modelu (wg adresu URL)
  – Template (szablon) – sposób prezentacji danych
  Model MVT Django separuje również dane (view) od sposobu ich prezentacji (template); Kontrolerem z modelu MVC w Django jest sam Django

Dynamiczne strony WWW – Django



Generowanie strony WWW przez Django:

- **Serwer WWW odbiera żądanie HTTP**; Jeżeli z konfiguracji serwera i URL żądania wynika, że żądanie ma być obsłużone przez skrypt WSGI, serwer przygotowuje dane o żądaniu w odpowiednim formacie i przekazuje do WSGI
- **Django odbiera dane z WSGI** i tworzy obiekt HttpRequest;
  Dane z odebrane WSGI są przekształcane, np. tworzone są oddzielne struktury na dane z formularzy oraz ciasteczka – dzięki temu dostęp do potrzebnych danych jest wygodniejszy; HttpRequest jest słownikiem języka Python
- **Django "dekoduje" URL żądania** i przeszukuje pliki urls.py projektu oraz aplikacji – na tej podstawie wybiera widok (tj. znajduje funkcję widoku)
- **Django wywołuje funkcję widoku**, przekazując jej HttpRequest; Dodatkowo, jeżeli we wzorcu URL widoku były zmienne (np. /archiwum/<rok>), funkcja widoku otrzymuje te zmienne; Widok jest odpowiedzialny za wygenerowanie obiektu HttpResponse – można go utworzyć bezpośrednio w funkcji widoku, jednak typowo wykorzystuje się do tego szablony i modele
- **Funkcja widoku tworzy potrzebny model** (albo modele) i wywołuje odpowiednie funkcje modelu, zgodnie z żądaniem oraz ewentualnie danymi z formularzy. Model aktualizuje swój stan (jeżeli to potrzebne) i dostarcza potrzebne dane.
- **Funkcja widoku tworzy kontekst i renderuje szablon** – szablon to zbiór wszystkich danych potrzebnych do wygenerowania strony WWW; Widok renderuje (tworzy) dokument HTML; Następnie tworzy obiekt HttpResponse (zawierający dokument HTML oraz typ zawartości i kod odpowiedzi), który zwraca jako swój rezultat;
- **Django przekształca HttpResponse** zgodnie z wymaganiami WSGI i zwraca do serwera WWW, który odsyła go przeglądarce
- W generowaniu odpowiedzi może być użyty Middleware;
  Klasy Middleware są wskazane w konfiguracji i są aktywowane na różnych etapach generowania odpowiedzi (zależnie od funkcji, które mają zdefiniowane);
  Django ma kilka użytecznych Middleware, ale można też definiować własne
- Obiekty statyczne (pliki CSS, obrazki, skrypty Javascript, …) powinny być umieszczone w folderze aplikacji wskazanym w konfiguracji, domyślnie "static", i jego podfolderach; Są odsyłane do serwera bez udziału widoków;

Struktura projektu Django:

```
<projekt>
 - settings.py
 - urls.py
 - wsgi.py
 - <aplikacja>
 - <aplikacja>
```

- Settings – konfiguracja projektu
- Urls – plik startowy dekodowania adresów URL
  (ciąg dalszy znajduje się w poszczególnych aplikacjach)
- Foldery aplikacji – każda aplikacja ma własny folder

Struktura aplikacji (bardzo prostej) Django:

```
<aplikacja>
 - <static>
 - urls.py
 - views.py
 - models.py
 - <templates>
    - szablon.html
```

- Folder <static> - pliki statyczne
- Urls – plik dekodowania adresów URL aplikacji
- Views.py – plik widoków
- Models.py – plik modeli
- Folder <templates> - pliki szablonów

Konfiguracja Django (<projekt>/settings.py):

- Włączenie trybu debugowania – powoduje wyświetlanie szczegółowych raportów o błędach w oknie przeglądarki;
  Należy wyłączyć w wersji produkcyjnej projektu

```
DEBUG = True
```

- Zainstalowane aplikacje – należy **dopisać** nazwy własnych aplikacji – bez tego Django nie znajdzie np. plików szablonów

```
INSTALLED_APPS = [
    […]
    'nazwa-aplikacji',
]
```

- Plik URL – startowy plik dekodowania adresów żądań

```
ROOT_URLCONF = 'project.urls'
```

- Baza danych – lokalizacja i ew. dane uwierzytelniania połączenia z bazą danych, domyślnie SQLite,

```
DATABASES = {
  'default':
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': BASE_DIR / 'db.sqlite3',
```

```
    }
}
```

- Folder plików statycznych – domyślnie "static", można zmienić

```
STATIC_URL = '/static/'
```

- Ustawienia regionalne

```
LANGUAGE_CODE = 'pl-pl'
TIME_ZONE = 'Europe/Warsaw'
USE_I18N = True
USE_L10N = True
USE_TZ = True
```

Dekodowanie adresów URL żądań:

Plik urls.py projektu

```
from django.contrib import admin
from django.urls import include, path

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

- Początkowo zawiera tylko ścieżkę do panelu zarządzania, "admin/" – aplikacji generycznej Django
- Funkcja include dołącza plik urls aplikacji
- Powinien uwzględniać stronę główną serwisu ("pusty" URL)
- Ważne! Adres powinien kończyć się znakiem "/"

Plik urls.py aplikacji

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

urlpatterns = [
  path('', views.blog, name='blog'),
  path('all/', views.blog, name='blog'),
  path('arch/<int:year>/', views.year),
]
```

- Rozpoznawane adresy URL są połączeniem adresów z plików urls projektu i aplikacji, np. "blog/all/" = "blog/" + "all/"
- W adresach można używać zmiennych i wyrażeń regularnych
- Można stosować aliasy

Typy zmiennych w adresach URL

- Typy zmiennych są zdeterminowane przez konwertery:
  – str – dowolny niepusty łańcuch znaków, bez separatora "/"
  – int – liczba całkowita nieujemna
  – slug – łańcuch znaków z myślnikami, "litery-i-cyfry-np-13"
  (popularny w portalach informacyjnych i systemach CMS)

```
'arch/<int:year>/'
'arch/<int:year>/<int:month>'
'notes/<slug:title>/'

r'^arch/(?P<year>[0-9]{4})/$'
```

- Można też używać wyrażeń regularnych (ostatni przykład wyżej) oraz definiować własne konwertery

Nazwy adresów URL i funkcja reverse()

- Plik urls.py aplikacji

```
urlpatterns = [
  path('', views.blog, name='blog'),
  path('arch/<int:year>/', views.year),
]
```

- Drugi argument wskazuje plik widoku, który zostanie użyty do wygenerowania strony
- Trzeci opcjonalny argument nazwany (name='<nazwa>') rejestruje nazwę adresu URL – taki adres można uzyskać przy pomocy funkcji reverse i użyć np. w odnośnikach menu

```
from django.urls import reverse
loginPage = reverse('login')
```

ORM

Motywacja

- Mapowanie obiektowo-relacyjne ORM (object-relational mapping) – technika programistyczna polegająca na konwersji niekompatybilnych systemów zapisu danych:
    - W programowaniu obiektowym OOP dane zazwyczaj nie są skalarne (obiekty często zawierają inne obiekty, które mogą zawierać jeszcze inne obiekty); Obiekty są też wyposażone w metody, służące m.in. do modyfikacji stanu obiektu
    - W relacyjnych bazach danych, w tym najbardziej popularnych SQL DBMS, dane w relacjach (tabelach) z zasady są skalarne,
      a zamiast zawierania obiektów są powiązania między tabelami
- Różnic między OOP a jest więcej, np. obiekty o identycznym stanie są rozróżnialne (mają różne adresy/referencje), za to rekordy (wiersze) tabel – nie (dlatego wyposaża się je w klucze)

Przeznaczenie

- Korzystanie z baz danych w programach OOP może przebiegać wg dwóch scenariuszy:
    - Program wykorzystuje dane w postaci skalarnej, tak jak są zapisane w bazie danych
    - Program tworzy złożone struktury danych, które konwertuje na wartości skalarne podczas zapisywania do bazy danych oraz konwertje je z powrotem na obiektowe podczas wczytywania
- ORM jest implementacją drugiego z ww. scenariuszy,
  polega na zautomatyzowaniu procesu odczytu/zapisu danych tak, aby zachować relacje między obiektami i poprawnie je odtworzyć w programie OOP
- ORM może tworzyć i modyfikować tabele w bazie danych na podstawie postaci/zmian definicji obiektów ("code first");
  Proces zmian bazy danych przez ORM to tzw. migracja

Zalety i wady

- Zalety
    - Znaczne zmniejszenie ilości kodu, który trzeba napisać
    - Brak zapytań SQL w kodzie programu
      (nie trzeba go znać, unika się błędów w zapytaniach, …)
    - Zwiększone bezpieczeństwo aplikacji
      (ORM automatycznie wykonuje walidację danych)
    - Zmniejszenie nakładu pracy i przyśpieszenie tworzenia aplikacji
- Wady
    - Całkowite poleganie na ORM, bez znajomości baz danych, prowadzi do powstania źle zaprojektowanych baz danych
    - Zwiększenie złożoności obliczeniowej – spowolnienie aplikacji
- Zalety najwyraźniej przewyższają wady – ORM jest coraz powszechniej stosowane, w C# (Entity Framework), Java (EJB), PHP (Laravel, RedBean, …), Python (Django, SQLAlchemy, …)

ORM w Django

- W implementacji wykorzystuje dwa mechanizmy:
  - Dziedziczenie (django.db.models.Model); Modele dziedziczą szereg metod i właściwości, np. objects, all(), filter(), …
  - Metaprogramowanie; Atrybuty dynamiczne (dodawane do już istniejącego obiektu), których nazwy pochodzą od nazw pól modelu, np. autors_set
- Wspiera podejście "code first" – tworzy bazę danych na podstawie modeli, z uwzględnieniem powiązań między tabelami (jeden-do-wielu, wiele-do-wielu-, jeden-do-jednego)
- Może obsługiwać istniejącą bazę danych – może utworzyć modele na podstawie istniejącej bazy danych:

```
py manage.py inspectdb > models.py
```

- Obsługuje różne systemy SQL DBMS – SQLite, PostgreSQL, …
- Można użyć innego ORM, niż wbudowany, np. Peewee, …

Modele

Klasa modelu

- Klasa musi dziedziczyć po django.db.models.Model
- Reprezentuje relację (tabelę) w bazie danych;
  Django dodaje klucz główny tabeli, o nazwie id

```
from django.db import models

class Post(models.Model):
  author = models.ForeignKey('auth.User')
  title = models.CharField(max_length=200)
  text = models.TextField()
  def __str__(self):
    return self.title
```

- Przez panel administracyjny projektu (dostępny pod adresem URL "/admin") mona zarządzać zawartością tabeli, ale pod warunkiem rejestracji modelu;
- Rejestracji dokonuje się w pliku admin.py aplikacji Django:

```
# <projekt>/<aplikacja>/admin.py

from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

- Model powinien definiować metodę __str__ – która jest wykorzystywana w panelu administracyjnym do wyświetlania listy rekordów bazy danych:

Typy pól modelu

- AutoField – pole IntegerField, ale dodatkowo z włączoną opcją autoinkrementacji, często wykorzystywane jako klucz główny tabeli – tego typu jest klucz tabeli tworzony przez Django
- BigAutoField – jak AutoField, ale z gwarancją zakresu wartości UInt64 (AutoFiled – zależy od używanej bazy danych)
- SmallAutoField – jak AutoField, ale zakres wartości Int16

- DateField – data, w programie reprezentowana przez obiekt datetime.date;
  Opcje:
  auto_now – automatyczna aktualizacja przy każdej modyfikacji
  auto_now_add – aktualizacja przy pierwszym zapisie
- DateTimeFiled – data i czas, obiekt datetime.datetime;
  Opcje jak DateField
- TimeField – czas, obiekt datetime.time;
  Opcje jak DateField

- CharField – łańcuch znaków, nieprzesadnie długi (dla dłuższych napisów jest pole TextField);
  Wymagany argument: max_length (maksymalna liczba znaków)

- TextField – łańcuchy znaków o dużej długości (ograniczenie tylko w bazie danych);
  Argument max_length jest opcjonalny
- SlugField – podklasa CharField, łańcuch znaków w notacji "slug"
- URLField – podklasa CharField, przy zapisie dodatkowo sprawdzane czy zawiera prawidłowy adres URL
- EmailField – podklasa CharField, przy zapisie sprawdzane czy zawiera prawidłowy adres email

- FileField – pole związane z przede wszystkim z przesyłaniem plików przez formularze HTML;
  Opcjonalne argumenty określają sposób zapisania pliku, zawsze w systemie plików, a nie w bazie danych – w bazie danych zapisywane są informacje o pliku (np. ścieżka), a nie plik;
  FieldFile – właściwość (pole) FileField, dająca dostęp do pliku; dostarcza pola name (względna ścieżka pliku), size (rozmiar) oraz metody open() i close()
- ImageField – podklasa FileField, dedykowana do obsługi plików graficznych, ma dodatkowe pola z rozmiarami obrazka;
  Wymaga zainstalowania dodatkowej biblioteki Pythona, Pillow

- IntegerField – liczba całkowita, zakres przynajmniej Int32, niezależnie od wykorzystywanej bazy danych
- BigIntegerField – liczba całkowita, zakres Int64
- FloatField – liczba rzeczywista w notacji zmiennoprzecinkowej
- DecimalField – liczba rzeczywista, w notacji stałoprzecinkowej,
  o dowolnie wybieranym zakresie oraz precyzji zapisu – służą do tego argumenty max_digit oraz decimal_places

Opcje pól

- Definiując pole można podać opcjonalne atrybuty – opcje pola; Wybrane, częściej używane opcje:
  – null – "puste" pole w bazie danych będzie zapisane jako NULL
    domyślnie null=False
  – blank – pole może być puste – dotyczy to walidacji, a nie bazy danych (jak null)
    domyślnie blank=False
  – default – wartość domyślna lub funkcja, która zwraca taką wartość,
    wywoływana w chwili tworzenia obiektu
  – primary_key – jeżeli True, to pole będzie kluczem głównym tabeli, z wszelkimi konsekwencjami – np. musi być unikalne; Jeżeli w modelu nie ma pola z opcją primary_key, Django doda pole AutoField o nazwie id, które będzie kluczem tabeli

Relacje w modelach

- ORM w Django obsługuje wszystkie rodzaje relacji (jeden-do-wielu, wiele-do-wielu, jeden-do-jednego); Każdemu typowi relacji odpowiada specjalny rodzaj pola modelu, który taką relację tworzy; Obsługa relacji, np. pobieranie powiązanych rekordów, jest realizowana automatycznie;
- Relacja jest definiowana zawsze tylko po jednej stronie; nazwa pola tworzącego relację daje dostęp do powiązanych rekordów w drugim modelu, a w drugim modelu dodawany jest służący do tego dynamiczny atrybut, <model>_set
- Najczęściej wykorzystywane są relacje jeden-do-wielu, rzadziej wiele-do-wielu, a relacje jeden-do-jednego bardzo rzadko

Relacja jeden-do-wielu

- ForeignKey – typ pola, który definiuje relację jeden-do-wielu;
  Dodaje się go po stronie "jeden", np. wpis może mieć wiele komentarzy, ale każdy komentarz dotyczy jednego wpisu:

```
from django.db import models

class Post(models.Model):
  # ...

class Comment(models.Model):
  post = models.ForeignKey(Post)
  # ...
```

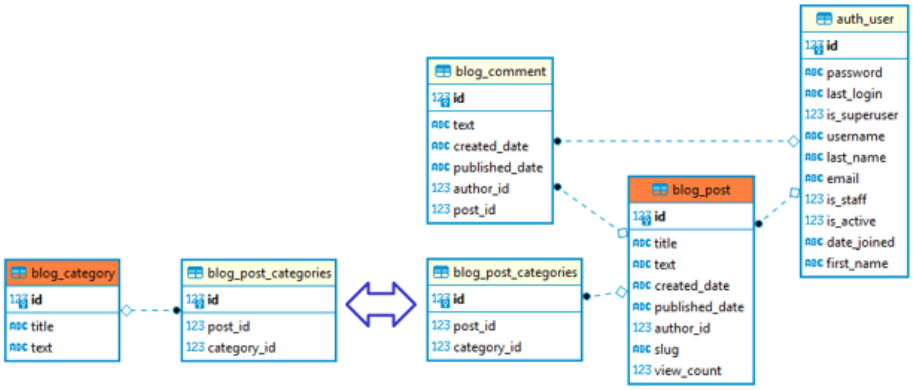
- Jeżeli modele są deklarowane w odwrotnej kolejności, to nazwę modelu trzeba jako łańcuch znaków, np. 'Post'
- Opcje pola ForeignKey:
  – on_delete – reakcja na usunięcie obiektu, np. CASCADE – usuwa powiązane pola
    PROTECT – uniemożliwia usunięcie powiązanego obiektu (i jeszcze kilka innych)
  – limit_choices_to – ogranicza wybór powiązanych obiektów w panelu administracyjnym Django do spełniających określone kryteria
  – related_name – nazwa relacji odwrotnej, domyślnie <model>_set
    (w przykładzie Post-Comment: post_set)
  – to_field – pole w docelowym obiekcie, do którego prowadzi
    powiązanie – domyślnie jest to klucz główny

Relacja wiele-do-wielu

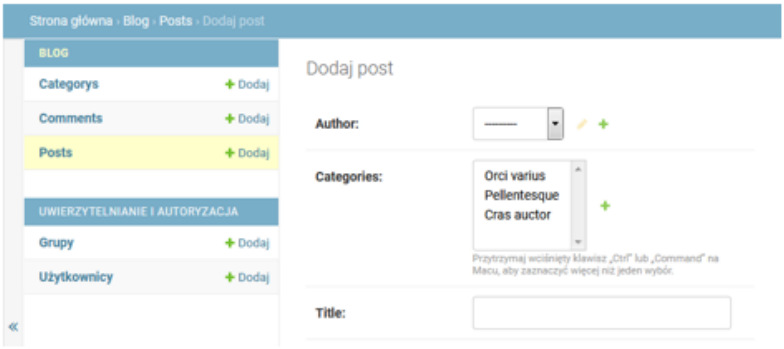
- ManyToMantField – typ pola definiujący relację wiele-do-wielu;
  Dodaje się go po tylko po jednej stronie relacji, a relacja jest symetryczna:

```
from django.db import models

class Post(models.Model):
 categories=models.ManyToManyField('Category')
 # ...

class Category(models.Model):
 # ...
```

- Jeżeli modele są deklarowane w odwrotnej kolejności, to nazwę modelu trzeba podać w postaci łańcucha znaków
- Relacja wiele-do-wielu jest "sztuczna" w bazach danych SQL, tworzy się ją za pomocą trzeciej tabeli, która ma obustronne relacje jeden-do-wielu – Django sam tworzy taką tabelę



- Relacja wiele-do-wielu jest symetryczna (w przykładzie Post-Category, po stronie Post jest pole categories, natomiast po stronie Category – automatyczne pole post_set)
- Wybór strony, po której jest zdefiniowana relacja, jest istotny jeżeli używa się panelu administracji Django – tylko po tej stronie będzie dostępny wybór – tu po stronie Post:



- Django umożliwia użycie własnego modelu (a w efekcie tabeli) pośredniczącej w relacji wiele-do-wielu. Model może mieć dodatkowe pola, charakteryzujące relację:

```
class Category(models.Model):
  posts = models.ManyToManyField(
    Post, through='Membership')
  # ...

class Membership(models.Model):
  # ...
```

- Przykład z dokumentacji Django: muzycy i grupy, relacja może zawierać np. daty początku i końca współpracy

- Opcje pola ManyToMantField :
  – limit_choices_to – ogranicza wybór powiązanych obiektów w panelu administracyjnym Django do spełniających określone kryteria (nie działa, kiedy użyto through)
  – related_name – nazwa relacji odwrotnej, domyślnie <model>_set (w przykładzie Post-Comment: post_set)
  – through – nazwa modelu pośredniczącego w relacji; w modelu pośredniczącym trzeba zdefiniować relacje jeden-do-wielu
  – through_fields – nazwy pól tabeli pośredniczącej, tworzące relację – potrzebne kiedy Django nie może sam ich wykryć z powodu niejednoznaczności (tj. kiedy relacji jest więcej)

Metadane modeli

- Metadane ("dane o danych") modeli można podać w klasie Meta, zdefiniowanej wewnątrz klasy modelu:

```
from django.db import models

class Category(models.Model):
  # ...

  class Meta:
    db_table = 'blog_categories'
    verbose_name_plural = 'categories'
```

- Wybrane metadane:
  – managed – czy tabela jest zarządzana przez Django, co obejmuje tworzenie i modyfikacje tabeli – jeżeli False, to dla tabeli nie są wykonywane migracje (domyślnie True)
  – app_label – jeżeli model pochodzi z aplikacji innej, niż bieżąca, nazwa tej aplikacji
  – db_table – nazwa tabeli w bazie danych (domyślnie: <aplikacja>_<model>, np. blog_post)
  – verbose_name, verbose_name_plural – nazwa tabeli w panelu administracyjnym
  – ordering – pole lub lista pól, stanowiące domyślne kryterium sortowania tabeli
  – get_latest_by – jw. dla dla funkcji latest() i earliest(),

```
  ordering = '-date_created'
  get_latest_by = [ 'x', '-y' ]
```

Migracje w Django

- Django tworzy tabele w bazie danych na podstawie modeli, może także zmodyfikować tabele na skutek zmian modeli (podejście "code first") – zmian bazy danych to tzw. migracja;
- Przebieg wszystkich migracji jest zapisywany w folderze "migrations" aplikacji
- Migracja nie obejmuje modeli, dla których w klasie Meta ustawiono opcję managed = False
- Migracje realizuje się z linii poleceń środowiska wirtualnego:

```
(venv) C:\Django\project>py manage.py makemigrations blog
Migrations for 'blog':
  blog\migrations\0007_post_visited.py
    - Add field visited to post
(venv) C:\Django\project>py manage.py migrate blog
Operations to perform:
  Apply all migrations: blog
Running migrations:
  Applying blog.0007_post_visited... OK
Migracje
Migracje w Django
```

QuerySet

- Kolekcja obiektów pobieranych z modelu, przez manager modelu (domyślna nazwa: objects) i jedną z funkcji do tego służących, np. all():

```
posts = Post.objects.all()
```

- QuerySet jest kolekcją "leniwie" inicjalizowaną – może być filtrowany i sortowany bez przeprowadzania operacji na bazie danych, dopiero odwołanie do jego elementów powoduje wykonanie zapytania SQL w bazie danych:
  – iterowanie w pętli for,
  – indeksowanie
  – wycinanie z określeniem kroku
  – wywołanie funkcji len() lub
  – wywołanie funkcji repr()
  – przekształcenie w listę, przez wywołanie funkcji list()

QuerySet – metody

- all() – wszystkie rekordy
- filter(**kwargs) – rekordy spełniające kryteria określone przez argumenty, tzw. field lookups: `<pole modelu>__<kryterium>=<wartość>`, przy czym dostępne kryteria zależą od typu pola, np.:
  ```
  text__contains = 'tekst wyszukiwania'
  value__in = [ 7, 13 ]
  created__year = 2020
  ```
- exclude(**kwargs) – rekordy niespełniające kryteriów
- order_by(*fields) – rekordy posortowane wg wskazanych pól (minus przed nazwą pola – porządek malejący)
  ```
  order_by('last_name', 'first_name', '-age')
  ```
- only(*fields) – tylko wskazane pola zamiast całych rekordów

QuerySet – metody, które nie zwracają QuerySet

- get(**kwargs) – zwraca pojedynczy obiekt, spełniający kryteria field lookups (jak w funkcjach filter i exclude); Rzuci wyjątek programowy, jeżeli nie ma obiektu spełniającego kryteria albo jeżeli takich obiektów jest więcej niż jeden

```
post = Post.objects.get(id=1)
```

- create(**kwargs) – tworzy nowy obiekt i zapisuje go w bazie;
  argumenty powinny dostarczyć wszystkie wymagane pola modelu

```
post = Post.objects.create(
  title = '…',
  text = '…'
  )
```

- first() – zwraca pierwszy rekord

```
first = Posts.objects.order_by('date').first()
```

- last() – zwraca ostatni rekord

  Funkcje first() i last() nie rzucają wyjątku, jeżeli rezultat jest pusty – inaczej niż funkcje latest(), earliest() i get()

- latest(*field) – zwraca ostatni rekord, przy sortowaniu według wskazanego pola; jeżeli klasa Meta określa kryterium get_latest_by, to argument można pominąć
- earliest(*field) – działa jak latest(), tylko odwraca kierunek sortowania; domyślne kryterium w klasie Meta: get_earliest_by

QuerySet – przykłady

- Przykłady dla modelu Post z poradnika Django Girls

```
# wszystkie wpisy, kolejność nieokreślona
posts = Post.objects.all()

# jw., według dat, od końca
posts = Post.object.order_by('-created_date')

# tylko wpisy z 2020 roku
posts = Post.objects.filter(created_date__year=2020)

# jw., posortowane alfabetycznie wg autorów
posts = Post.objects.filter(
  created_date__year=2020).order_by('author')
```

Widoki

Funkcja widoku

- Po zdekodowaniu adresu URL żądania, Django wywołuje funkcję widoku, przekazując jej szczegóły żądania w postaci obiektu request (typu HttpRequest)
- Funkcja widoku jest odpowiedzialna za wygenerowanie kompletnej strony WWW, którą zwraca jako rezultat (obiekt typu HttpResponse). Funkcja widoku może wygenerować odpowiedź sama, ale zwykle używa się do tego celu szablonów.
- Za wczytanie i renderowanie szablonu odpowiada obiekt klasy Dispatcher ("dyspozytor"). Można tworzyć własne obiekty dispatcherów albo używać systemu szablonów spoza Django, jednak najczęściej używa się funkcji-skrótu render
- Wbudowane szablony Django najlepiej jest używać poprzez funkcję skrótu render – render wczytuje dispatchera, znajduje szablon, renderuje go i zwraca gotowy obiekt HttpResponse;
- Funkcji render trzeba przekazać obiekt żądania (request), nazwę szablonu oraz kontekst – wszystkie dane potrzebne do wygenerowania strony, zebrane w słowniku języka Python:

```
from django.shortcuts import render
#...

def blog(request):
  return render(
    request, 'blog.html', { <kontekst> }
  )
```

- Funkcja widoku zwykle tworzy obiekt modelu, przez funkcje modelu aktualizuje jego stan w bazie danych (w razie potrzeby) oraz pobiera dane potrzebne do wyświetlenia strony

```
from django.shortcuts import render
from .models import Post

def blog(request):
  posts = Post.objects.all()
  return render(
    request, 'blog.html', {'posts':  posts}
  )
```

- Dodatkowo, jeżeli we wzorcu URL widoku były zmienne (np. /archiwum/<int:year>), funkcja widoku otrzymuje te zmienne w postaci dodatkowych argumentów :

```
from django.shortcuts import render
from .models import Post

def archives(request, year):
  posts = Post.objects.filter(
    date_created__year = year
  )
  return render(
    request, 'arch.html', {'posts':  posts}
  )
```

Szablony

- Django ma własny system szablonów (oparty na Ninja), ale umożliwia użycie dowolnego innego systemu szablonów
- Szablon służy do wygenerowania dokumentu HTML, który zostanie odesłany jako odpowiedź na żądanie HTTP;
- Pliki szablonów należy umieszczać w folderze "templates", który powinien być umieszczony w folderze aplikacji; Django znajdzie szablon pod warunkiem, że aplikacja zostanie dopisana do listy aplikacji w pliku konfiguracji projektu, "settings.py"

```
<projekt>
 - settings.py
 - <aplikacja>
   - templates
     - szablon.html
```

Kontekst szablonu

- Kontekst szablonu to słownik języka Python, przekazany funkcji renderującej szablon z funkcji widoku – powinien zawierać wszystkie potrzebne dane:

```
def blog(request):
  posts = Post.objects.all();
  catts = Categories.object.all();

  return render(
    request, 'blog.html',
    {'posts': posts, 'catts': catts, …}
  )
```

- Django przed renderowaniem szablonu tworzy obiekt dedykowanego podtypu kontekstu, RequestContext, do którego dodaje obiekt żądania request (typu HttpRequest), zawartość kontekstu przekazanego z funkcji widoku oraz dane dodawane przez tzw. procesory kontekstu – ich lista jest w konfiguracji projektu, w pliku settings.py:

```
'OPTIONS': {
  'context_processors': [
    'django.template.context_processors.debug',
    'django.template.context_processors.request',
    'django.contrib.auth.context_processors.auth',
    'django.contrib.messages.context_processors.messages'
  ],
},
```

- Przy standardowych opcjach projektu w kontekście szablonu znajdują się:
  - Dane przekazane do kontekstu z funkcji widoku
  - user – obiekt typu auth.User, z danymi aktualnie zalogowanego użytkownika, dane jak w panelu administracyjnym Django, albo obiekt AnonymousUser, jeżeli nikt nie jest zalogowany; Można wykorzystać w menu użytkownika (np. zaloguj/wyloguj)
  - perms – obiekt PermWrapper, z uprawnieniami użytkownika, jak w panelu administracyjnym: `<aplikacja>.<add/view/change/delete>_<model>`, np. blog.add_post, blog.view_post, …, blog.add_comment, …
  - request - obiekt żądania HttpRequest, zawierający wszystkie parametry żądania, przetworzone dane z formularzy itd.

Szablony

- Pliki szablonów są plikami html – dzięki czemu edytory kodu zapewniają wsparcie jak dla plików html (kolorowanie składni, autouzupełnianie, intellisense, …) z dopisanymi w języku szablonów informacjami, które obejmują zmienne, filtry oraz znaczniki:

```
{{ zmienna }}
{{ zmienna | filtr }}
{% znacznik %} … {% znacznik_końcowy %}
```

- Django ma obszerną kolekcję filtrów i znaczników, pokrywającą większość potrzeb, ale można też tworzyć własne

Zmienne

- Wartości zmiennych mogą pochodzić wyłącznie z kontekstu przekazanego do szablonu przez funkcję render, z obiektami dodanymi przez procesory kontekstu (user, perms, request, …)
- Zmienne kontekstu zwykle nie są zmiennymi typów prostych. Język szablonów umożliwia pobieranie wartości z kolekcji (słowników, list, krotek, …), obiektów, a nawet funkcji; Niezależnie od typu, stosuje się notację kropkową: a.b.c.d…
- Funkcja renderująca próbuje odczytać wartość na różne sposoby, w określonej kolejności:
  - Pozycja słownika: a[b],
    Kluczem może być łańcuch (ma pierwszeństwo) lub liczba
  - Atrybut lub metoda obiektu: a.b albo a.b()
    Metoda może być wywołana tylko jeżeli nie ma argumentów
  - Indeks a[int(b)]
- Przykłady poprawnych odwołań do zmiennych kontekstu:

```
def fun():
  return 'funkcja'
data = { 'a': 'A', 'f': fun, 'l': [7,13,36] }
{{ data.a }}    # data['a']
{{ data.f }}    # data['f']()
{{ data.l.2 }}  # data['l'][int('2')]
```

Filtry

- Filtry to funkcje, które służą do modyfikowania wartości zmiennych w szablonach. Przyjmują jeden albo dwa argumenty (pierwszym jest zawsze zmienna szablonu)

```
{{ user.name | upper }}
{{ post.text | truncatewords: 10 }}
{{ book.authors | join: ", " }}
```

- Można połączyć działanie kilku filtrów, za każdym razem używając symbolu przetwarzania potokowego, tj. znaku "|":

```
{{ user.email | lower | urlize }}
```

- Wiele filtrów oczekuje zmiennej określonego typu – najczęściej łańcucha znaków lub liczby, rzadziej daty, czasu, słownika, … Należy zwrócić na to uwagę w opisie filtru
- Argument może być podany w postaci literału, ale może być wartością innej zmiennej należącej do kontekstu

```
{{ list1|add:  list2 }}
```

- Typ argumentu zależy od filtru, zazwyczaj to łańcuch znaków lub liczba całkowita – dla literałów łańcuchów cudzysłowy lub apostrofy są obowiązkowe (inaczej napis jest nazwą zmiennej), dla literałów liczb – opcjonalne:

```
{{ list|join:", " }}  # 1, 2, 3
{{ list|join: ,   }}  # błąd!
{{ list.2|add:"2" }}  # 5
{{ list.2|add: 2  }}  # 5
```

Przykładowe filtry

- date:"format" – formatuje datę zgodnie, do wyboru jest ~40 znaków formatujących
- time – jw., czas
- filesizeformat – zamienia liczbę na czytelny format w KB lub MB, np. 34.5 MB
- join:"str" – zamienia kolekcję na łańcuch, używając argumentu jako separatora wartości, np. `[1,2,3]|join:", "` -> `"1, 2, 3"`
- first – zwraca pierwszy element listy/krotki lub łańcucha znaków
- last – jw., ostatni
- length – zwraca długość kolekcji lub łańcucha znaków
- linebreaks – zamienia znaki końca linii w tekście na HTML (jeden znak \n na
, dwa znaki \n pod rząd – na <p>)
- lower – zamienia tekst na pisany małymi literami;
- upper – jw., dużymi
- title – jw., każdy wyraz dużą literą
- random – wybiera losowy element listy/krotki
- striptags – usuwa wszystkie znaczniki HTML
- truncatewords:N – obcina łańcuch do pierwszych N słów
- truncatewords_html:N – jw., ale pomija zawartość znaczników
- urlize – znajduje adresy www/email i zmienia je w łącza
- urlizetrunc:N – jw., ale skraca wyświetlany adres do N znaków
- yesno:"tak,nie,nic" – zamienia wartość bool/None na napis

Znaczniki

- Są bardziej złożone i bardziej różnorodne niż filtry; Niektóre zwracają łańcuchy znaków (np. url), inne są odpowiednikiem instrukcji języka Python (if-elif-else oraz for-empty) albo dołączają zewnętrzne biblioteki lub szablony (np. load, extends)
- Niektóre znaczniki maja znacznik zamykający – jeżeli tak, to jest on obowiązkowy

Znacznik if-elif-else

- Odpowiednik instrukcji warunkowej if-elif-else Pythona, włącza do obiektu HttpResponse określony fragment szablonu, zależnie od wartości warunku;
- Jako warunek można podać zmienną (jej wartość jest wówczas konwertowana na typ bool wg reguł Pythona) albo wyrażenie z użyciem operatorów:
  – logicznych (and, or)
  – równości (==, !=, is), zawierania (in) i relacji (>, >=, <, <=)
- W wyrażeniach warunków można używać dowolnych filtrów, np. filtru length, który zwraca długość listy/krotki
- Sekcje elif oraz else są opcjonalne
- Znacznik zamykający zawsze jest obowiązkowy
- Przykład

```
{% if data|length > 10 %}
  <p>Cała lista: kliknij tutaj…
{% elif data|length %}
  <p>{{ data|join : ", " }}
{% else %}
  <p>brak danych
{% endif %}
```

Znacznik for-empty

- Odpowiednik instrukcji iteracyjnej for Pythona, najczęściej używany do włączenia do obiektu HttpResponse wszystkich elementów kolekcji, odpowiednio sformatowanych; Działa również na wyrażeniach generatorów, łańcuchach itp.
- Może rozpakować kolekcję, może iterować od końca (dopisek "reverse")
- Dostarcza kilka przydatnych zmiennych, które można wykorzystać do lepszego renderowania kolekcji, np.:
  – forloop.first – True jeżeli jest wykonywana pierwsza iteracja
  – forloop.last – jw., jeżeli ostatnia iteracja
  – forloop.counter – numer iteracji (iteracje liczone od 1)
- Sekcja empty (opcjonalna) jest wykonywana, jeżeli kolekcja jest pusta
- Znacznik zamykający zawsze jest obowiązkowy
- Przykład

```
{% for url, title in searchresults reverse %}
  <p><a href='{{ url }}'>{{ title }}</a>
  {% if not forloop.last %}
    <hr>
  {% endif %}

{% empty %}
  <p>Brak wyników…

{% endfor %}
```

Znacznik url

- Zwraca adres URL zdefiniowany w plikach url projektu albo bieżącej aplikacji.
- Pierwszym argumentem znacznika url jest nazwa wzorca url (nadana w pliku url). Dodatkowe, opcjonalne argumenty zastępują wartości we wzorcu url – o ile są w nim zmienne
- Przykładowy plik url:

```
urlpatterns = [
  path('main/', views.main, name='main'),
  path('post/<int:id>/', views.post, name='post'),
]
```

- Przykład użycia znacznika url (oczywiście odnośniki trzeba by umieścić w innych znacznikach, jak <p> albo <li> itp.):

```
<a href="{% url 'main' %}">Strona główna</a>

{% for p in posts %}
  <a href="{% url 'post' p.id %}">{{ p.title }}</a>
{% endfor %}
```

Znaczniki block, extends i include

- Służą do budowania hierarchicznych systemów szablonów wielokrotnego użycia
- Szablon główny zawiera strukturę dokumentu HTML oraz elementy powtarzające się na wszystkich podstronach serwisu, takie jak np. nagłówek, menu główne, stopka; W miejsce elementów zmiennych – np. menu bocznego, głównej zawartości – definiuje tylko bloki – znacznik "block"; Blok może być pusty, ale może też zawierać domyślną treść
- Szablon potomny najpierw włącza szablon główny – znacznik "extends", musi to być pierwszy znacznik pliku – a następnie definiuje treść bloków – znacznik "block" - które zastąpią bloki szablonu głównego o tych samych nazwach; Jeżeli któryś blok szablonu głównego nie wystąpi w szablonie potomnym, to zostanie w nim treść "domyślna"
- Przykład, (1) szablon główny (plik 'page.html'):

```
<body>
  <header> … <header>
  <nav>
    {% block main_menu %}
      domyślna treść menu
    {% endblock %}
  </nav>
  <main>
    {% block main_contents %}
      domyślna treść główna
    {% endblock %}
  </main>
  <footer> … </footer>
</body>
```

- Przykład, (2) szablon potomny (plik 'post.html')
  W szablonie potomnym nie występuje blok "main_menu", zatem zostanie wzięta domyślna wartość tego bloku z szablonu 'page.html'

```
{% extends 'page.html' %}

{% block main_contents %}
   zawartość główna
{% endblock %}
```

- Znacznik "include" włącza wskazany szablon i renderuje go. Nazwa włączanego szablonu może mieć postać literału albo zmiennej kontekstu
- Do włączanego szablonu można przekazać dodatkowe zmienne, definiując je za pomocą instrukcji "with"
- Znacznik "include" nie ma zamknięcia, a włączany szablon jest wczytywany w całości – nie definiuje się w nim bloków
- W większości przypadków stosuje się znaczniki block i extends, jednak odpowiednio użyty znacznik include także może być bardzo przydatny
- Wszystkie pliki szablonów domyślnie mają ten sam kontekst; Tylko kontekst szablonu włączanego przez "include" może być zmieniony instrukcją "with" (ale nie jest to obowiązkowe)
- Przykład, (3) szablon potomny (plik post.html');
  Szablon potomny włącza jeden z dostępnych wariantów menu, jednocześnie dodając do kontekstu włączanego szablonu zmienną, której wartość bierze z własnego kontekstu

```
{% extends 'page.html' %}

{% block aside %}

  {% include 'about-author-aside.html'
       with author=post.author %}

{% endblock %}
```

Inne znaczniki

- load – wczytuje dodatkowe biblioteki filtrów i znaczników, np.
  `{% load static %}`

- autoescape off – wyłącza zamianę potencjalnie niebezpiecznych znaków (np. "<" i ">") na encje HTML; Wymaga zamknięcia
- cycle – najczęściej używany w pętli for, przy każdej iteracji wybiera swój kolejny argument, a po ich wyczerpaniu powtarza argumenty od początku. Argumenty podaje się bez przecinków, mogą to być literały tekstowe lub zmienne kontekstu, np.:

```
{% for …
  <p class='{% cycle "red" "blue" %}'> …
{% endfor %}
```

- firstof – wybiera pierwszy niepusty argument
- now – bieżąca data i czas, formatowanie jak przy filtrze date

Znaczniki "niestandardowe"

- Do funkcji renderującej dodawane są automatycznie tylko wybrane znaczniki i filtry. Django ma ich więcej, można też definiować własne albo pobrać je z sieci. Aby z nich skorzystać, trzeba najpierw do szablonu dodać bibliotekę
- Często używanym znacznikiem "niestandardowym" jest static, który zwraca bezwzględną ścieżkę do plików statycznych:

```
{% load static %}

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <link rel="…" href="{% static 'style.css' %}">
</head>
…
```

Zadania

Proszę utworzyć w projekcie Django z poprzedniego ćwiczenia model do zapisywania komentarzy do wpisów, następnie zmodyfikować funkcję widoku wpisu i szablon tak, aby komentarze były wyświetlane razem z wpisem, a w tym celu:

1. Dodać model Comment
   Model można wzorować na modelu Post – komentarz powinien mieć treść (bez tytułu) oraz datę utworzenia. Model powinien być powiązany relacją jeden-do-wielu z autorem (analogicznie jak model Post) oraz taką samą relacją w wpisem
2. Wykonać migrację

   Efekty sprawdzić za pomocą programu do zarządzania SQLite, najlepiej DBeaver;
   W szczególności należy sprawdzić czy powiązania pomiędzy modelami są prawidłowe.
3. Zarejestrować model Comment w pliku admin.py, dzięki czemu będzie dostępny w panelu administracyjnym Django

   Należy pamiętać o dodaniu nowego modelu do instrukcji importu
4. Zalogować się do panelu administracyjnego, utworzyć konta 2-3 użytkowników, dodać kilka wpisów i do każdego z nich (oprócz jednego!) kilka komentarzy.
   Wpisy i komentarze powinny być przypisane na zmianę różnym użytkownikom

   Efekty sprawdzić za pomocą programu do zarządzania SQLite, najlepiej DBeaver
5. Zmienić funkcję widoku pojedynczego wpisu tak, aby do kontekstu dodawała również komentarze powiązane z wpisem (można użyć QuerySet i funkcji filter(), ale znacznie wygodniej jest wykorzystać relację odwrotną w modelu Post, domyślnie `comment_set`)
6. Zmienić szablon potomny dla pojedynczego postu tak, aby pod wpisem wyświetlał komentarze do tego wpisu; Należy użyć znacznika szablonów `for-empty`, przy czym sekcja `empty` powinna wyświetlać komunikat o braku komentarzy;

   Przy każdym wpisie i komentarzu powinny być wyświetlane informacje o autorze, np. nazwa, adres email itp.