Wykład 5: Interakcja z bazą danych w Django (draft)

Date: 2015-11-08
tags: zaj5, wykład, materiały
category: materiały

Struktura projektu Django

Projekty django mają określoną strukturę, zawierają przynajmniej takie pliki:

`settings.py`

: Jest to moduł Pythona zawierający wszystkie ustawienia projektu Django, takie jak: domyślny język, lista zainstalowanych aplikacji (i wiele innych) Plik `settings.py` zawiera również ustawienia zależne od konkretnego środowiska wdrożeniowego, takie jak: adres bazy danych, adres usługi cache itp, ścieżka do przechowywania plików.

`manage.py`

: Skrypt służący do zarządzania projektem Django, pozwalającym na wykonywanie rożnych operacji, takich jak: zarządzanie wersją schematu bazy danych, tworzenie administratorów, uruchamianie testów itp.

`wsgi.py`

: Skrypt służący do integracji Django z serwerami WWW (apache, nginx, uwsgi).

`urls.py`

: Zawiera informacje jaki widok odpowiada za przetworzenie jakiej ścieżki.

Do tego w projekcie znajduje się kilka aplikacji, aplikacja zawiera takie pliki:

`models.py`

: Zawiera klasy odpowiedzialne za interakcje z bazą danych.

`views.py`

: Zawiera widoki, --- czyli klasy i funkcje odpowiedzianle za odpowiadanie na zapytania HTTP.

`tests.py`

: Zawierają testy (o tym później).

`admin.py`

Zawiera klasy definiujące interfejs administracyjny (o tym później).

ORM Django

Pojęcie ORM

ORM to ogólny termin oznaczający Object Relational Mapping (mapowanie z obiektów na relacje), czyli mechanizm pozwalający na: tworzenie tabel, zapis i odczyt danych **bez pisania kodu sql**.

W zasadzie każdy framework WWW zawiera jakiś mechanizm ORM, orm jest stosowany z takich powodów:

- Umożliwia osiągnięcie przenośności kodu między bazdami danych. Język `SQL` został ustandaryzowany bardzo późno, więc (w zasadzie) nie da się pisać przenośnego kodu `SQL` (tj. kodu SQL który działa na więcej niż jednej bazie danych).

  Note

  Niekompatybilności mogą być drobne, np. w postgresql do generowania systetycznych kluczy służą sekwencje, a w mysql należy używać kolumn auto increment.
  Jednak już kilka takich niekompatybilności powoduje, że utrzymanie aplikacji, która komunikuje się z kilkoma bazami danych jest uciążliwe (bez użycia ORM)

- Powoduje, że nie trzeba pisać ręcznie 99% zapytań SQL.
- Powoduje, że nie trzeba ręcznie odczytywać odpowiedzi z serwera SQL. Warstwa ORM wykonuje zapytanie i zwraca gotowe instancje modelu.

Note

Przenośność aplikacji między bazami danych jest rzeczą cenną i przydatną, ale nie zawsze jest grą wartą świeczki. Część operacji łatwiej jest wykonać na poziomie bazy danych, niż w aplikacji, skorzystanie z funkcjonalności dostępnych w jednej bazie danych oczywiście uniemożliwia przenośność na inne.
W praktyce przenośność między bazami danych ważna jest tylko dla **reużywalnych** aplikacji, np. takich jak zarządzanie użytkownikami, dla typowych stron, które będziecie pisać często warto jest z przenośności zrezygnować.

Konfiguracja bazy danych w pliku `settings.py`

Django obsługuje aplikacje wykorzystujące **wiele** baz danych, ale w naszych przykładach, zawsze będziemy korzystać z jednej bazy daych.

Bazy danych konfigurowane są za pomocą ustawienia `DATABASES` w pliku `settings.py`.

Domyślnie wygląda ono tak:

```
DATABASES = {
  'default': {
      'ENGINE': 'django.db.backends.sqlite3',
      'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Definiuje ono, że Django korzysta z jednej bazy danych, w tym wypadku jest to baza danych `sqlite` (nie poleca się korzystania z `sqlite` w środowisku produkcyjnym).

Jeśli korzystasz z jednej bazy danych powinna ona być w słowniku `DATABASES` pod kluczem `default`.

> **Note**
> Baza danych sqlite jest dość przydatna do szybkiego przeprowadzania testów jednostkowych w aplikacji.

Rodzaj bazy danych definiuje klucz `ENGINE`, dla bazy danych postgresql jest to `django.db.backends.postgre`

By podłączyć się do lokalnej bazy danych można zastosować taką definicję bazy danych:

```
DATABASES = {
    'default': {
        'ENGINE':
          'django.db.backends.postgresql_psycopg2',
        'NAME': 'zaj4-rano', # Nazwa bazy danych
    }
}
```

Tak potraktowane Django spróbuje podłączyć się do bazy danych za pomocą gniazd linuksa na **na użytkownika o tej samej nazwie, jak użytkownik OS, który włącza Django**.

Jeśli chcecie się połączyć do bazy danych na innym komputerze, należy:

```
DATABASES = {
    'default': {
        'ENGINE':
          'django.db.backends.postgresql_psycopg2',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

> **Note**
> Uwaga --- w plikach settings.py znajdują się hasła do baz danych **w tekście jawnym**, umieszczanie ich w serwisach typu github jest niebezpiecznym pomysłem.
> Jak zarządzać plikami settings (i między innymi chronić hasła) powiem później.

Definiowanie modeli

Polecam przejrzeć dokumentacje Django (nie będe z niej pytał na kolokwiach, ale warto przejrzeć!):

- https://docs.djangoproject.com/en/1.8/topics/db/models/
- https://docs.djangoproject.com/en/1.8/ref/models/fields/

Model w `Django` odpowiada tabeli, instancja modelu odpowiada pojedyńczemu wierszowi w danej tabeli.

Modele są jedynym i ostatecznym źródłem informacji o bazie danych.

Modele umieszczamy (zwykle) w pliku `models.py`. Przykład (części) pliku `models.py` z aplikacji która wygenerowała mi dane na zajęcia 2.

```
class Student(models.Model):

  name = models.CharField(max_length=100)
  courses = models.ManyToManyField("Course", db_table="student_course")

  class Meta:

    db_table='student'


class Course(models.Model):

  name = models.CharField(max_length=100)

  lecturers = models.ManyToManyField(
    "Course", through="CourseInstance", related_name="courses")
  rooms = models.ManyToManyField(
    "Room", through="CourseInstance", related_name="rooms")


  class Meta:

    db_table='course'

class Mark(models.Model):

  student = models.ForeignKey('Student', null=False)
  course = models.ForeignKey('Course', null=False)

  mark = models.PositiveSmallIntegerField(null=False)

  class Meta:

    db_table='mark'
```

Klasy modeli zawierają statyczne pola, które definiują kolumny w baie danych, np.

pole `mark` w modelu `mark` zawiera "małą całkowitą liczbę", a pole `course` jest kluczem obcym to modelu `course`.

> **Note**
>
> Każda z tabel będzie dodatkowo zwierała kolumnę `id` będącą syntetycznym kluczem głownym. Nie musiałem definiować jej jawnie, ponieważ w Django **każda tabela** zawsze musi mieć **syntetyczny klucz głowny**.

Dodatkowo każdy model może (nie musi!) zawierać wewnętrzną klasę `Meta`, która zawiera informacje opisujące **cały model** --- albo przynajmniej **więcej niż jedną kolumnę**, jedno z pól `Meta` pozwala określić jawnie nazwę tabeli.

Samo zdefiniowanie modelu nie powoduje, że tabelka automagicznie znajdzie się w bazie danych.

> **Note**
>
> Mechanizm, którego Django używa do odczytiwania kolumn z modeli i generowania wszystkich metod modelu (o tym dokładnie jakie metody są generowane powiem w dalszej części wykładu), jest bardzo ciekawy, jednak dość zaawansowany (i słabo udokumentowany, najlepsza dokumentacja procesu, --- którą znam to książka "The Django Book", która jednak dotyczy django `1.0`. Czyli bardzo starego.
> Generalnie mechanizm ten używa metaklas, które pozwalają na zmiane zawartości typu przed jego zdefiniowaniem.
> W innych językach by osiągnąc tego typu rozwiązania, po prostu pisze się generatory kodu (np. JPA w Javie SE potrzebowała generatorów kodu dla bardziej zaawansowanych zastosowań).

Pojęcie migracji

W praktyce nie da się stworzyć aplikacji na zasadzie: "Napiszmy wszystkie modele, stwórzmy bazę danych i wgrajmy ją na serwer", aplikacja ewoluuje, baza danych musi ewolować z nią. Dotyczy to również **produkcyjnej bazy danych**.

Django ewoluuje bazę danych za pomocą migracji, migracje to specjalne pliki Pythona, które opisują stan tabel powiązanych z daną aplikacją, w jakiejś chwili czasu (modele opisują stan bazy danych **w chwili aktualnej**).

Django zapamiętuje, która wersja aplikacji jest zainstalowana w danej bazie danych, i może zmigrować dane z bazy danych do wersji najnowszej.

Tworzenie migracji

By stworzyć migrację należy napisać polecenie:

```
./manage.py makemigrations
```

powstanie nam wtedy katalog `migrations`, zawierający plik `0001_initial.py`, czyli pierwszą migrację do wgrania do bazy danych.

Z zawartością podobną do:

```
class Migration(migrations.Migration):

  dependencies = [
  ]

  operations = [
      migrations.CreateModel(
          name='Course',
          fields=[
              ('id', models.AutoField(
                verbose_name='ID', auto_created=True,
                serialize=False, primary_key=True)),
              ('name', models.CharField(max_length=100)),
          ],
      ),
      migrations.CreateModel(
          name='Mark',
          fields=[
              ('id', models.AutoField(
                verbose_name='ID', auto_created=True,
                serialize=False, primary_key=True)),
              ('mark', models.PositiveSmallIntegerField()),
              ('course', models.ForeignKey(to='zaj2_schema_app.Course')),
          ],
      ),
      migrations.CreateModel(
          name='Student',
          fields=[
              ('id', models.AutoField(
                verbose_name='ID', auto_created=True,
                  serialize=False, primary_key=True)),
              ('name', models.CharField(max_length=100)),
              ('courses', models.ManyToManyField(
              db_table='student_course', to='zaj2_schema_app.Course')),
          ],
      ),
      migrations.AddField(
          model_name='mark',
          name='student',
          field=models.ForeignKey(to='zaj2_schema_app.Student'),
      ),
  ]
```

Plik ten zawiera instrukcje pozwalające zainstalowanie tabel opisanych w pliku `models.py`.

Jeśli dokonasz jakichś zmian w pliku `models.py` i ponownie wykonasz `./manage.py makemigrations` w katalogu `migrations` pojawi się kolejny plik, którego nazwa zaczyna się od `0002`. Np. jeśli postanowie zmienić nazwy tabel, mighracja może wyglądać tak:

```
class Migration(migrations.Migration):
```

```
    dependencies = [
        ('zaj2_schema_app', '0001_initial'),
    ]

    operations = [
        migrations.AlterModelTable(
            name='course',
            table='course',
        ),
        migrations.AlterModelTable(
            name='mark',
            table='mark',
        ),
        migrations.AlterModelTable(
            name='student',
            table='student',
        ),
    ]
```

Tutaj proszę zwrócić uwagę na pole `dependencies`, określa ono porządek migracji, tj. mówi, że przed wgraniem danej migracji wgrana musi być migracja o nazwie: `0001_initial` z aplikacji: `zaj2_schema_app`.

> **Note**
> Migracje mogą służyć do **fajniejszych** rzeczy, niż wgrywanie zmian generowanych z plików `models.py`, mogą na przykład wygrywać napisany przez Was kod SQL, który definiuje dodatkowe funkcjonalności w bazie danych.

Wgrywanie migracji do bazy danych

Samo stworzenie migracji, również, nie spowoduje, że nowy schemat automagicznie pojawi się, a bazie danych, w tym celu należy wydać polecenie:

```
./manage.py migrate
```

Wykonywanie zapytań z Django ORM

Dodawanie danych

By dodać dane do bazy danych należy stworzyć instancję modelu, a następnie wywołać na niej funkcje save.

```
>>> s = Student()
>>> s.name="foo"
>>> s.id is None # ID nadaje dopiero baza danych
```

```
True
>>> s.save() # Zapisanie do bazy danch
>>> s.id # Id jest już dostępne.
1001
```

> **Note**
> Nie jest to wydajna metoda kiedy chcecie dodać np. milion rekordów, by stworzyć takie zapytanie należy skorzystać z metody bulk_create

Tak samo wykonuje się polecenia update, jeśli zmienię stworzony obiekt student, to wywołanie metody `save` wykona polecenie update.

Wykonywanie selectów

Do interakcji z bazą danych służy atrybut `objects` na klasie reprezenującej model.

By wybrać wszystkich studentów muszę napisać:

```
from zaj2_schema_app.models import *
studenci = Student.objects.all()
>>> studenci[0].name
'Rebecca Maille'
```

By sprawdzić jakie zapytanie zostanie wykonane można napisać:

```
>>> print(str(studenci.query))
SELECT "student"."id", "student"."name" FROM "student"
```

Zauważcie, że zapytanie to zwraca obiekty typu student.

> **Note**
> Funkcja objects zwraca dość specyficzny typ, a mianowicie `django.db.models.query.py`, możecie o nim myśleć jak o liście.
> Faktycznie quersyet jest obiektem **leniwym**, tj. w chwili wywołania `studenci = Student.objects.all()` żadne zapytanie nie trafi do bazy danych, zapytanie trafi do bazy danych, kiedy odczytuje się dane z QuerySeta.
> Dokładne zasady ewaluowania querysetów opisane są tutaj.

Wykonywanie selectów, filtrowanie danych

Do filtrowania służy funkcja `filter`, która pozwala wybierać wiersze z tabeli, np. by wybrać studenta o określonym imieniu należy:

```
Student.objects.filter(name= 'Rebecca Maille')
```

W tym wypadku zostanie wykonane zapytanie:

```
'SELECT "student"."id", "student"."name" FROM "student" WHERE "student"."name"
```

Wybieranie modeli po kluczu głównym

Domyślnie klucz głowny jest w kolumnie o nazwie `id`, można stworzyć własny klucz głowny w kolumnie o innej nazwie. By wybrać dane względem klucza głownego zawsze można użyć filtru `pk=`, który wybierze dane względem klucza głownego, bez względu na nazwę kolumny.

```
Student.objects.filter(pk= 1)
```

Zwróci:

```
'SELECT "student"."id", "student"."name" FROM "student" WHERE "student"."id" =
```

Szablony Django

> **Note**
> Jest to materiał trochę dodatkowy, zapoznanie się z nim nie jest wymagane do zaliczenia następnych zajęć. Ale może to zaliczenie znacznie ulatwić, ponieważ za pomocą szablonów django **łatwiej** będzie Wam pisać kod html.
> Jak będziecie mieli więcej czasu to zapraszam do lektury.

Filozofia szablonów Django

Szablony django powstały dość późno, i architektura bazuje na zdiagnozowanych problemach z innymi silnikami szablonów: `JSP` i `PHP`.

JSP i PHP pozwalało na umieszczanie wewnątrz szablonow dowolnego kodu, JSP pozwalało na wykonywanie kodu Javy, a PHP na wykonywanie kodu PHP. Programiści Django stwierdzili, że nie jest to najlepsze rozwiązanie, ponieważ powoduje, że w szablonach zaczyna pojawiać się **logika** aplikacji, co jest niepożądane.

Kod szablonów nie powinien:

- Sprawdzać uprawnień do wykonywania zadaia.
- Wykonywać zapytań SQL (w Django: konstruować nowych querysetów)
- Robić nietrywialnego przetwarzania dnaych.

Tworzenie szablonów

Szablony Django to pliki HTML z dodatkowymi tagami (język opiszę za chwilę), zwykle umieszczamy je w katalogu `templates` w danej aplikacji.

Jeśli mam aplikację `zaj4` to szablony powinienem umieścić w katalogu: `templates/zaj4`, tak by aplikacja wyglądała tak:

```
zaj4/
  templates/
    zaj4/
```

```
        students.html
    admin.py
    models.py
    views.py
    ...
```

By wykorzystać szablon `students.html` muszę w widoku wykonać następujący kawałek kodu:

```
from django.shortcuts import render

from .models import Students

def index(request):
    students = Question.objects.order_by('name') # lisa studentów
    template = loader.get_template('zaj4/students.html') # Ładuje szablon
    context = {'students': students} # Definiuje dane z których korzysta szablo
    return HttpResponse(template.render(context)) # Wyświetlam szablon i tworzę
```

Język szablonów Django

Język szablonów django zawiera dwie konstrukcje:

- Wyświetlenie zmiennej `{{ zmienna }}`.
- Wykonanie tagu `{% tag %}`

Zmienna do wyświetlenia musi znajdować się w kontekście, jeśli jej brakuje nic nie zostanie wyświetlone.

Mogę też odwołać się do atrybutów zmiennej: `{{ student.name }}` wyświetli mi imię i nazwisko studenta.

Tagi w Django

Tag `{% for %}`

Tag `for` służy do iteracji i np. tworzenia wylistowań:

```
<ul>
{% for student in students %}
    <li>{{ student.name }}</li>
{% empty %}
  <li>Sorry, no students in this list.</li>
{% endfor %}
</ul>
```

Tag `{% if %}`

Tag `if` służy wykonuje instrukcję warunkową:

```
<ul>
{% if students %}
  {% for student in students %}
    <li>{{ student.name }}</li>
  {% endfor %}
{% else %}
  <li>Sorry, no students in this list.</li>
{% endif %}
</ul>
```