Spis treści

Podstawy Pythona

Wstęp do języka

Python jest dynamicznie typowanym językiem interpretowanym którego składnia jest bardzo zbliżona do języka angielskiego, nie używa tylu znaków interpunkcyjnych jak języki wywodzące się z C (jak Java lub PHP). Bardzo ważnym aspektem są wcięcia, które są wymagane oraz w obrębie pliku muszą używać tego samego stylu (spacje lub tabulacja). Jest to najczęstszy ból dla początkujących lecz z czasem staje się odruchowe i powoduje czysty i czytelny kod.

Wykonywanie kodu Pythona może następować na dwa sposoby: uruchamianie zawartości pliku oraz z poziomu interaktywnej konsoli która pozwala na przyjemne eksperymentowanie z językiem.

Instalacja

Windows

Instalator Pythona dla systemów Windows dostępny jest pod adresem: https://www.python.org/download/releases/2.7.6

Linux

Każda dystrybucja powinna posiadać paczki Pythona, jedynie należy je zainstalować używając swojego managera pakietów.

OS X

Na każdym systemie OS X dostępny jest Python 2.7.x.

Pip

Jedną z cech Pythona pozwalającą na szybsze tworzenie projektów jest modularność. Jednym miejscem gdzie zbierane są wszystkie biblioteki to repozytoruim PyPi: https://pypi.python.org/pypi.

Aby łatwiej instalować paczki PyPi powstało narzędzie `pip`. Dzięki niemu, można jedną linią poleceń zainstalować wymaganą paczkę w wymaganej wersji lub nawet zainstalować wszystkie zależności z pliku tekstowego. Aby zainstalować `pip` należy:

Windows

1. Pobrać plik: https://raw.github.com/pypa/pip/master/contrib/get-pip.py
2. Uruchomić wiersz poleceń (`cmd.exe`)
3. Wykonać plik `get-pip.py` za pomocą Pythona: `C:\Python27\python.exe get-pip.py`
4. Dodajć ścieżki `C:\Python27` oraz `C:\Python\Scripts` do zmiennej środowiskowej PATH

Linux

Pip powinien być dostępny w managerze pakietów jako `python-pip`.

OS X

W systemie domyślnie zainstalowany jest przestarzały manager paczek Pythona o nazwie `easy_install`. Można go użyć do zainstalowania pipa:

```
$ sudo easy_install pip
```

requirements.txt

`pip` posiada parametr `-r` w którym można podać plik tekstowy zawierający listę paczek które mają zostać zainstalowane. Zwyczajowo taki plik nazywany jest `requirements.txt` i jest umieszczony w głównym katalogu projektu. Aby stworzyć taki plik, najszybszym sposobem jest zrzucenie zainstalowanych paczek komendą:

```
$ pip freeze > requirements.txt
```

aby zainstalować paczki z pliku należy wykonać:

```
$ pip install -r requirements.txt
```

Virtualenv oraz virtualenvwrapper

Każdy projekt posiada swój zestaw zależności z różnymi wersjami paczek. Aby móc pracować z wieloma takimi zestawami, powstało narzędzie o nazwie `virtualenv` oraz nakładka na niego, ułatwiające nimi zarządzanie. Na początek należy je zainstalować:

Windows

```
> pip install virtualenv virtualenvwrapper-win
```

Linux i OSX

```
$ sudo pip install virtualenv virtualenvwrapper
```

Oto kilka podstawowych komend virtualenvwrapper:

- mkvirtualenv - tworzenie nowego środowiska
- cdvirtualenv - przejście do folderu środowiska
- workon - przełączenie na inne środowisko

Aby więc utworzyć nowe środowisko można uruchomić:

```
$ mkvirtualenv pierwszyprojekt
$ cdvirtualenv
```

Teraz wszystkie paczki instalowane z tego terminala, będą przechowywane w oddzielnym folderu, dzięki czemu nie będą powodować konfliktów z innymi.

Edytory i IDE

Istnieje wiele edytorów/IDE obsługujących Pythona lecz dwa są najczęściej wybierane:

- Sublime Text http://www.sublimetext.com/ - darmowy edytor z potężnym systemem wtyczek pozwalającym na wygodną pracę z Pythonem i Django
- PyCharm http://www.jetbrains.com/pycharm/ - płatne IDE świetnie zintegrowane z Pythonem oraz Django oferując introspecję, obsługę virtualenv, git oraz wiele innych.

Interaktywna konsola Pythona

Wspomniana wcześniej interaktywna konsola Pythona to jedna z najsilniejszych jego stron. Można ją jednak jeszcze bardziej wzmocnić instalując paczkę IPython która dodaje takie funkcje jak podpowiadanie komend oraz paczek, zaawansowany podgląd zmiennych, obsługa profilera oraz debuggera i wiele, wiele innych. Instalacja wygląda tak samo jak w przypadku virtualenv (przed instalacją najlepiej zamknąć wszystkie terminale z Pythonem i otworzyć nowy):

```
$ pip install ipython
```

Teraz aby dostać się do konsoli Pythona (a konkretnie IPythona) należy uruchomić `ipython` który powinien przywitać podobnym komunikatem:

```
$ ipython
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
Type "copyright", "credits" or "license" for more information.

IPython 2.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

W tej konsoli można wykonywać dowolny kod Pythona i obserwować jego wynik:

```
In [1]: 1 + 1
Out[1]: 2

In [2]: name = 'Ned'

In [3]: len(name)
Out[3]: 3
```

Poza prostymi jednolinijkowymi poleceniami, można pisać wielolinijkowy kod i testować go "na żywo":

```
In [1]: for i in range(1, 10):
   ...:     if i % 2:
   ...:         print i
   ...:
1
3
5
7
9
```

Należy zauważyć, że IPython automatycznie dodaje wcięcia gdy są potrzebne (należy je dodawać ręcznie jeśli używa się zwykłej konsoli Pythona).

Typy danych

Python zawiera wszystkie podstawowe typy danych takie jak `int`, `float`, `string` czy `boolean` oraz kilka "tablicowych" typów znanych z innynych języków pod inną nazwą:

Lista

ang. `list`, czyli podstawowa tablica indeksowana liczbami, odpowiednik `Array` w JavaScripcie/Ruby/PHP. Deklarowana za pomocą nawiasów kwadratowych:

```
In [1]: l = [1, 'a', 3.0]

In [2]: l
Out[2]: [1, 'a', 3.0]

In [3]: l[1]
Out[3]: 'a'
```

Aby wyciągnąć tylko fragment listy, Python stosuje specjalną sładnię `[start:stop]`:

```
In [1]: l = ['a', 'b', 'c', 'd', 'e']

In [2]: l[1:3]
Out[2]: ['b', 'c']

In [3]: l[1:]
Out[3]: ['b', 'c', 'd', 'e']

In [4]: l[:1]
Out[4]: ['a']

In [5]: l[-1:]
Out[5]: ['e']

In [6]: l[:-1]
Out[6]: ['a', 'b', 'c', 'd']
```

Tupla

ang. `touple`, jest odmianą listy której nie można modyfikować. Definiowana z użyciem nawiasów okrągłych.

```
In [1]: t = (1, 'a', 3.0)

In [2]: t
Out[2]: (1, 'a', 3.0)

In [3]: t[1]
Out[3]: 'a'

In [4]: t2 = (2,)

In [5]: t2
Out[5]: (2,)
```

Należy pamiętać aby przy inicjacji jednoelementowej tupli, dodać przecinek ponieważ w przeciwnym wypadku zamiast tupli otrzymamy zmienną o wartości przekazanej przy inicjacji:

```
In [6]: t3 = (2)

In [7]: t3
Out[7]: 2
```

Tupla, choć nie edytowalna, może podlegać łączeniu oraz innym operatorom:

```
In [1]: t1 = (1, 2)

In [2]: t2 = (3, 4)

In [3]: t1 + t2
Out[3]: (1, 2, 3, 4)

In [4]: t1 * 3
Out[4]: (1, 2, 1, 2, 1, 2)

In [5]: 2 in t1
Out[5]: True
```

Te same operatory mogą być używane przy listach.

Słownik

ang. `dictionary`, odpowiednik tablic asocjacyjnych z PHP/JavaScript, Map w Javie lub hashy z Ruby. Do inicjacji używa się nawiasów klamrowych:

```
In [1]: foo = { 'name': 'Jon', 'surname': 'Snow' }

In [2]: foo['name'] + ' ' + foo['surname']
Out[2]: 'Jon Snow'
```

Kluczami mogą być wszystkie hashowalne typy jak `string`, `int` czy `float`.

Moduły

Python zamiast dodawania przestrzeni nazw używa tzw. modułów czyli folderu zawierającego plik `__init__.py` (który może być pusty) oraz pliki Pythona lub kolejne moduły. Tak więc struktura katalogów kodu staje się mapą jego modułów. Importowanie modułów odbywa się za pomocą funkcji `import`:

```
In [1]: import os

In [2]: os.name
Out[2]: 'posix'

In [3]: from datetime import datetime

In [4]: datetime.today()
Out[4]: datetime.datetime(2014, 5, 21, 22, 2, 20, 9976)
```

IPython podpowiada (po naciśnięciu klawisza `Tab`) zarówno słowa kluczowe jak i zawartość modułów:

```
In [1]: from da
datetime  dateutil

In [1]: from datetime import
MAXYEAR        MINYEAR        date           datetime       datetime_CAPI  time           timed
elta      tzinfo

In [1]: from datetime import date

In [2]: date.
date.ctime          date.fromtimestamp  date.isoweekday     date.month          date.resolution
date.today          date.year
date.day            date.isocalendar    date.max            date.mro            date.strftime
date.toordinal
date.fromordinal    date.isoformat      date.min            date.replace        date.timetuple
date.weekday
```

Przyjętym sposobem organizacji importów jest następująca kolejność:

1. Standardowe moduły pythona
2. Moduły instalowane via `pip`
3. Moduły z projektu

Każda sekcja powinna być oddzielona pustą linią. Przykładowe importy powinny wyglądać następująco:

```python
import os
from datetime import datetime

from django.shortcuts import render
from django.core.urlresolvers import reverse

from .models import Photo
from .forms import PhotoForm
```

Funkcje

Funkcje deklarowane są za pomocą słowa kluczowego `def`:

```
In [1]: def name():
   ...:     print 'Ned'
   ...:

In [2]: name()
Ned
```

Python obsługuje zarówno funkcje z parametrami nazwanymi jak i nienazwanymi. Możliwe jest także deklarowanie parametru opcjonalnego, poprzez nadanie mu wartości początkowej:

```
In [1]: def sons(first, second, third=''):
   ...:     print first, second, third
   ...:

In [2]: sons('Robb', 'Bran')
Bran Robb

In [3]: sons(b='Bran', a='Robb')
Bran Robb

In [4]: sons('Robb', 'Bran', 'Rickon')
Robb Bran Rickon
```

Dodatkowo funkcje mogą przyjmować dwa specjalne parametry: `*args` i `**kwargs` czyli:

- **args** - lista przekazanych argumentów
- ***kwargs** - słownik przekazanych argumentów nazwanych

Deklarując te parametry, funkcja może otrzymywać zmienną ilość parametrów:

```
In [1]: def show(*args, **kwargs):
   ...:     for arg in args:
   ...:         print arg
   ...:     for key in kwargs:
   ...:         print key, ':', kwargs[key]
   ...:

In [2]: show(1, 'b', name='Jon', surname='Snow')
1
b
surname : Snow
name : Jon
```

Oba specjalne parametry są także bardzo przydatne przy przekazywaniu otrzymanych parametrów do innej funkcji bez potrzeby ich przepisywania.

Klasy

Definiowanie klas w Pythonie wykonuje się słowem kluczowym `class`. Oczywiście są obsługiwane takie mechanizmy jak dziedziczenie są także obsługiwane:

```
In [1]: class Stark:
   ...:     def surname(self):
   ...:         print 'Stark'
   ...:

In [2]: class Ned(Stark):
   ...:     def name(self):
   ...:         print 'Ned'
   ...:

In [3]: person = Ned()

In [4]: person.name()
Ned

In [5]: person.surname()
Stark
```

Należy zauważyć, że metody klasy posiadają parametr `self` (odpowiednik `this` z JavaScriptu) który automatycznie jest do nich przekazywany i stanowi powiązanie z resztą klasy.

Wyjątki

W Pythonie wszelkie nieoczekiwane wartości lub wyniki wewnątrz funkcji wyrzucają wyjątki (w odróżnieniu od zwracania kodów błędu). Takie wyjątki można przechwytywać i obsługiwać:

```
In [1]: starks = ['Brab', 'Robb']

In [2]: print starks[2]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-2-2a578b39aa7e> in <module>()
----> 1 print starks[2]

IndexError: list index out of range

In [3]: try:
   ...:     print starks[2]
   ...: except IndexError:
   ...:     print 'Index too big'
   ...:
Index too big
```

Lambdy

Jednym z ciekawszych elementów Pythona są lambdy, które pomagają wprowadzić odrobinę programowania funkcyjnego gdzie funkcje mogą być przekazywane lub zwracane:

```
In [1]: def transform(n):
   ...:     return lambda x: x + n
   ...:

In [2]: f = transform(3)

In [3]: f(4)
Out[3]: 7
```

Dekoratory

Kolejnym elementem Pythona wartym uwagi są dekoratory. Dekoratory to specjalne funkcje które "oplatają" inne funkcje, pozwalając na modyfikację ich działania. Użycie dekoratora oznacza się symbolem `@` przed deklaracją funkcji:

```
In [1]: def lower(func):
   ...:     def wrapper(*args, **kwargs):
   ...:         return str(func(*args, **kwargs)).lower()
   ...:     return wrapper
   ...:

In [2]: @lower
   ...: def motto():
   ...:     return 'A Lannister Always Pays His Debts.'
   ...:

In [3]: motto()
Out[3]: 'a lannister always pays his debts.'
```

Podstawy Django

Pierwszy projekt

Pora rozpocząć pierwsze kroki z Django. Należy rozpocząć od stworzenia świeżego virtualenv-a:

```
$ mkvirtualenv photogram
New python executable in photogram/bin/python
Installing setuptools.............done.
Installing pip...............done.
(photogram)$ cdvirtualenv
```

Następnie, należy zainstalować w nim paczkę Django:

```
$ pip install django
Downloading/unpacking django
  Downloading Django-1.6.5.tar.gz (6.6MB): 6.6MB downloaded
  Running setup.py egg_info for package django

    warning: no previously-included files matching '__pycache__' found under directory '*'
    warning: no previously-included files matching '*.py[co]' found under directory '*'
Installing collected packages: django
  Running setup.py install for django
    changing mode of build/scripts-2.7/django-admin.py from 644 to 755

    warning: no previously-included files matching '__pycache__' found under directory '*'
    warning: no previously-included files matching '*.py[co]' found under directory '*'
    changing mode of /Users/suda/.virtualenvs/photogram/bin/django-admin.py to 755
Successfully installed django
Cleaning up...
```

Gdy Django jest już zainstalowane, można stworzyć nowy projekt:

```
$ django-admin.py startproject photogram
```

lub na systemie Windows:

```
> python Scripts\django-admin.py startproject photogram
```

Nowo utworzony projekt powinien mieć następującą strukturę folderów:

```
photogram
|
+- photogram
|  |
|  +- __init__.py
|  |
|  +- settings.py
|  |
|  +- urls.py
|  |
|  +- wsgi.py
|
+- manage.py
```

Zadania poszczególnych plików to:

- `mange.py` - nakładka na `django-admin.py` pozwalająca na zarządzanie projektem
- `__init__.py` - plik deklarujący Pythonowi iż ninejszy folder jest modułem
- `settings.py` - plik ustawień Django
- `urls.py` - główne deklaracje adresów URL
- `wsgi.py` - konfiguracja WSGI, protokołu służącego do komunikacji z serwerem WWW

Aby zobaczyć stworzony projekt w przeglądarce, należy uruchomić wbudowany serwer WWW:

```
$ cd photogram
$ python manage.py runserver
Validating models...

0 errors found
May 24, 2014 - 17:05:02
Django version 1.6.5, using settings 'photogram.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C
```

Po otwarciu podanego podanego adresu w przeglądarce, powinna pojawić się strona z gratulacjami.

django-extensions

Przed przejściem dalej, zalecam zainstalowanie paczki `django-extensions` która poza wieloma dodatkowymi funkcjami, posiada bardziej rozbudowany serwer WWW oraz konsolę:

```
$ pip install django-extensions Werkzeug ipython
```

Po zainstalowniu paczki, należy zmodyfikować plik `settings.py` a konkretnie zmienną `INSTALLED_APPS` dodając do niej `django_extensions`:

```
INSTALLED_APPS = (
    ...
    'django_extensions',
)
```

Następnie pod koniec pliku dodając ustawienie IPython jako domyślnej konsoli:

```
SHELL_PLUS = "ipython"
```

Od teraz aby przetestestować projekt, należy uruchomić:

```
$ python manage.py runserver_plus
```

Pierwsza aplikacja

Projekty Django składają się z małych, modularnych części zwanych aplikacjami. Ideą Django jest DRY czyli "nie powtarzaj się", dlatego aplikacje powinny być pisane tak aby bez modyfikacji można było je użyć w innym projekcie. Mając wielką bazę użytkowników, Django doczekało się wielu zewnętrznych aplikacji do blogów, for, galerii, zarządzania treścią itd. Przed rozpoczęciem pisania nowej funkcjonalności od zera, proponuję przejrzenie strony Django Packages https://www.djangopackages.com/ aby sprawdzić czy już ktoś tego nie zrobił.

Aby stworzyć aplikację, należy wykonać:

```
$ python manage.py create_app main
```

Został utworzony folder o nazwie `main` który zawiera:

```
main
|
+- __init__.py
|
+- forms.py
|
+- models.py
|
+- urls.py
|
+- views.py
```

Poszczególne pliki to:

- `forms.py` - deklaracje formularzy służące do generowania HTML oraz ich walidacji
- `models.py` - deklaracje modeli przechowujących dane
- `urls.py` - deklaracje adresów URL używanych przez aplikację
- `views.py` - widoki służące do odpowiadania na zapytania HTTP

MVC/MTV

Django korzysta z metodologii MVC czyli model-widok-kontroler. Jej ideą jest rozdzielenie kodu aplikacji na trzy części:

- **kontroler** - służy do komunikacji z modelem, modyfikując jego stan lub wartości oraz z widokiem który instruuje w jaki sposób prezentować dane z modelu
- **model** - służy do zarządzania danymi w aplikacji (np. przechowywania ich w bazie danych)
- **widok** - służy do prezentowania danych z modelu dla użytkownika

Istnieje popularna idea o "grubym modelu i chudym kontrolerze". Dyktuje ona aby jak najwięcej logiki przechowywać w modelach a pozostawić kontrolery do "spinania" widoków z kontrolerami.

Django implementuje zmodyfikowaną wersję MVC o nazwie MTV czyli model-szablon-widok gdzie:

- **model** deklaruje strukturę przechowywanych danych
- **szablon** jest plikiem HTML używającym specjalnej składni aby renderować przekazane mu dane
- **widok** pobiera potrzebne modele i przekazuje je do szablonu wykonując jak najmniejszą ilość logiki

Dochodzi do tego **definicja adresów URL** w pliku `urls.py` która definiuje pod jakim adresem wyświetlić który widok.

Kontrolery oraz pliki urls.py

Aby stworzona aplikacja była "zauważona" przez Django i reagowała na zapytania, należy wykonać kilka kroków. Po pierwsze należy dodać ją do `INSTALLED_APPS` w `settings.py`:

```
INSTALLED_APPS = (
    ...
    'main',
)
```

Następnie, trzeba napisać pierwszy widok w `views.py`:

```
# -*- encoding: utf-8 -*-

from django.http import HttpResponse

def hello(request):
        return HttpResponse('Hello World!')
```

Należy tu zwrócić uwagę na trzy rzeczy:

1. Na początku pliku znajduje się linia `# -*- encoding: utf-8 -*-`. Informuje ona Pythona o tym, że plik jest kodowany UTF-8. Brak tej linii w połączeniu z użyciem polskich znaków diakrytycznych, będzie wywoływać błędy kodowania.
2. Widok pobiera parametr `request`. Jest to parametr typu `django.http.HttpRequest` automatycznie przekazywany przez Django. Zawiera takie informacje jak metadane HTTP (np. nagłówki), dane POST/GET i wiele innych.
3. Widok zwraca instancję klasy `django.http.HttpResponse`. Widoki zawsze zwracają tą klasę (lub jej potomka), nawet jeśli zwracają czysty tekst.

Szczegółowe informacje n.t. `HttpRequest` i `HttpResponse` znajdują się w dokumentacji Django: https://docs.djangoproject.com/en/1.6/ref/request-response/.

Mając działający widok, należy go przypisać do konkretnego adresu URL. Robi się to poprzez dopisanie go do pliku `urls.py`. Można zauważyć, że w projekcie znajdują się dwa takie pliki: jeden w katalogu głównym projektu i drugi w aplikacji. Plik w katalogu głównym powinien kierować ścieżkę URL na plik w katalogu aplikacji, który następnie rozdzieli podścieżki na swoje widoki. Taka konstrukcja posiada kilka zalet:

- przy większych projektach, pliki ze ścieżkami są podzielone na małe, czytelniejsze fragmenty
- łatwość zmiany ścieżki całej aplikacji np. z */forum/* na */community/*
- aplikację można skopiować do innego projektu i "podpiąć" jedną linią

Aktualnie główny plik `urls.py` wygląda tak:

```
1  from django.conf.urls import patterns, include, url
2
3  from django.contrib import admin
4  admin.autodiscover()
5
6  urlpatterns = patterns('',
7      # Examples:
8      # url(r'^$', 'photogram.views.home', name='home'),
9      # url(r'^blog/', include('blog.urls')),
10
11     url(r'^admin/', include(admin.site.urls)),
12 )
```

Jak widać jest wpięta aplikacja panelu administracyjnego ale o tym potem. Po tej linii należy dodać:

```
url(r'^', include('main.urls')),
```

co skieruje wszystkie adresy (taki zabieg pozwala na wyświetlanie strony głównej z poziomu aplikacji) na plik `urls.py` aplikacji `main`:

```
1  try:
2      from django.conf.urls import *
3  except ImportError:  # django < 1.4
4      from django.conf.urls.defaults import *
5
6  from . import views
7
8  urlpatterns = patterns('',
9      url(r'^$', views.hello, name='hello'),
10 )
```

- łapanie wyjątku przy importowaniu zostało stworzone przez `django_extensions` przy tworzeniu aplikacji. Jako, że 1.6 jest aktualną wersją, można usunąć cały blok `try … except` i zamienić na pojedyńczy import `from django.conf.urls import *`.
- import widoku wykonany jest za pomocą wygodnego triku w postaci znaku `.`, pozwalającego na zaimportowanie modułów używając względnej ścieżki. Przydatny sposób oszczędzający czas przy refactoringu

- definicja ścieżki posiada parametr `name` który używany jest przy generowaniu adresu do widoku z poziomu kodu

Po odświeżeniu adresu aplikacji, zamiast komunikatu z gratulacjami, powinien pojawić się tekst z widoku.

Jak można było zauważyć, ścieżki w Django używają wyrażeń regularnych. Pozwalają także na wychwytywanie fragmentów ścieżki i przekazywanie ich do widoku:

```
url(r'^(?P<name>[\w]+)/$', views.hello_name, name='hello_name'),
```

Oczywiście wymaga to dodania widoku `hello_name`:

```
def hello_name(request, name):
        return HttpResponse('Hello %s!' % name)
```

Nazwane grupy w wyrażeniu regularnym (rozpoczynające się od `?P<...>`) zostaną przekazane do widoku pod tą samą nazwą.

Modele oraz ORM

Jedną z najważniejszych cech aplikacji jest dostęp do bazy. Django dzięki swojemu modułowi ORM (mapper relacji obiektów), pozwala na prosty i wygodny dostęp do danych bez względu na serwer jaki jest używany. Django standardowo obsługuje:

- PostgreSQL (zalecane)
- MySQL
- SQLite
- Oracle

Zalecana baza PostgreSQL posiada wiele zalet nad MySQL oraz umożliwia Django bezpieczniejsze operacje na danych przy użyciu transakcji. Na początek, dla lokalnego developmentu można używać SQLite, który jest automatycznie skonfigurowany po stworzeniu projektu, co widać w pliku `settings.py` w zmiennej `DATABASES`:

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

Jak widać, zmienna `DATABASES` jest słownikiem, ponieważ Django może obsługiwać kilka baz danych jednocześnie.

Pierwszy model

W Django schemat bazy danych przechowywany jest w postaci modeli które deklarowane są jako klasy dziedziczące po `django.db.models.Model` w plikach `models.py`. Tworząc projekt na podobieństwo aplikacji Instagram (którego backend jest napisany w Django) model zdjęć mógłby wyglądać tak:

```python
from django.contrib.auth.models import User

class Photo(models.Model):
        user = models.ForeignKey(User, null=False, blank=False)
        title = models.CharField(max_length=255, null=True, blank=True)
        date = models.DateTimeField(auto_now_add=True, null=False, blank=False)
        image = models.ImageField(upload_to="images/", null=True, blank=True)
```

Pierwsza własność `user` deklaruje klucz obcy `ForeignKey` do modelu `User` który jest wbudowanym modelem Django i służy do przechowywania użytkowników. Przy deklaracji tego pola, przekazano dwa parametry:

- **null** - informuje czy zapisywać puste obiekty w bazie danych jako NULL
- **blank** - informuje czy to pole jest wymagane. Jeśli przy próbie zapisu modelu wszystkie wymagane pola nie będą uzupełnione, wywoła to wyjątek

Kolejnym polem jest `title` mające przechowywać tytuł zdjęcia. Zostało ono zadeklarowane jako `CharField` czyli ciąg znaków o maksymalnej długości (w bazie danych zazwyczaj reprezentowany jako `VARCHAR`). Posiada ono wymagany parametr `max_length` czyli maksymalną długość ciągu znaków (wszystko poza tą długość zostanie usunięte).

Trzecim polem jest data typu `DateTimeField` przechowująca datę oraz czas. Posiada ona dwa parametry dodatkowe:

- **auto_now** - ustawione na `True` spowoduje ustawienie wartości pola na aktualną datę i czas podczas zapisywania obiektu. Bardzo przydatne przy polach zapisujących ostatnią modyfikację obiektu
- **auto*now*add** - ustawione na `True` spowoduje ustawienie wartości pola na aktualną datę i czas podczas pierwszego zapisywania obiektu. Przydatne przy przechowywaniu daty utworzenia obiektu.

Ostatnim polem jest obrazek który zadeklarowano typem `ImageField`. Typ ten dziedziczy po `FileField` które wymaga parametru `upload_to` wskazującego gdzie względem zmiennej `MEDIA_ROOT` w `settings.py` plik zostanie zuploadowany. Typ `ImageField` dodatkowo sprawdza czy plik jest obrazkiem a nie tylko plikiem. Do tego wymaga paczki `Pillow`:

```
$ pip install pillow
```

Django posiada wiele typów pól, m.in.:

- **BooleanField** - pole typu Boolean mogące przybierać wartości `True` lub `False`
- **SmallIntegerField**, **IntegerField/BigIntegerField** - pola przechowujące liczby całkowite
- **PositiveIntegerField/PositiveSmallIntegerField** - pola przechowujące liczby naturalne
- **TextField** - pole tekstowe bez ograniczenia długości
- **URLField/EmailField/IPAddressField** - pole tekstowe do przechowywania adresów URL/email/IP z ich walidacją
- **TimeField/DateField** - pola podobne do `DateTimeField` lecz służące do przechowywania tylko czasu/daty

Dodatkowo każde pole dziedziczące po `django.db.models.Field` posiada takie atrybuty jak:

- **choices** - pozwala przekazać listę lub tuplę z możliwymi wartościami oraz ich opisami
- **db_index** - równe `True` wymusi nałożenie indeksu w bazie danych. Pola typu `ForeignKey` automatycznie go nakładają, lecz może być przydatny także na innych polach
- **default** - domyślna wartość pola
- **unique** - równe `True` wymusi unikalność wartości tego pola (tj. jego wartość nie może się powtarzać)
- **verbose_name** oraz **help_text** - przechowuje czytelną nazwę pola oraz pomoc. Używane w panelu administracyjnym oraz formularzach

Pełna lista typów pól oraz ich atrybutów, dostępna jest pod adresem: https://docs.djangoproject.com/en/1.6/ref/models/fields/.

Synchronizowanie bazy danych

Aby zpropagować bazę danych schematem zapisanym w modelach, należy uruchomić komendę `syncdb`:

```
$ python manage.py syncdb
Creating tables ...
Creating table django_admin_log
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'suda'):
Email address: admin@suda.pl
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

Poza tworzeniem wymaganych tabel i indeksów, przy pierwszej synchronizacji komenda zapyta o dane super użytkownika który będzie miał pełen dostęp do panelu administracyjnego.

South

Niestety komenda `syncdb` nie potrafi zarządzać dobrze zmianami i migracjami danych/schematu. Dlatego powstała paczka o nazwie South którą należy zainstalować:

```
$ pip install south
```

dodać do `INSTALLED_APPS`:

```
1  INSTALLED_APPS = (
2      ...
3      'south',
4  )
```

oraz zsynchronizować ponownie bazę:

```
$ python manage.py syncdb
```

Następnie, należy zainicjować South w aplikacji:

```
$ python manage.py convert_to_south main
        Creating migrations directory at '/Users/suda/.virtualenvs/photogram/photogram/main/mig
rations'...
Creating __init__.py in '/Users/suda/.virtualenvs/photogram/photogram/main/migrations'...
 + Added model main.Photo
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate main
 - Soft matched migration 0001 to 0001_initial.
Running migrations for main:
 - Migrating forwards to 0001_initial.
 > main:0001_initial
   (faked)
```

Można zauważyć, że w katalogu aplikacji pojawił się folder `migrations`. Zaglądając do niego i pliku `0001_initial.py` można zobaczyć, że migracje to kawałki kodu w Pythonie które można modyfikować. Metoda `forwards()` jest wykonywana przy migracji "do przodu" (do nowszej wersji) a `backwards()` przy cofaniu migracji (np. w przypadku błędu).

Jeśli teraz dodanoby kolejne pole do modelu `Photo` np:

```
published = models.BooleanField(default=False)
```

należałoby przygotować migrację:

```
$ python manage.py schemamigration main --auto
 + Added field published on main.Photo
Created 0002_auto__add_field_photo_published.py. You can now apply this migration with: ./manag
e.py migrate main
```

South automatycznie wykrył dodane pole. Teraz wystarczy dokonać migracji:

```
$ python manage.py migrate main
Running migrations for main:
 - Migrating forwards to 0002_auto__add_field_photo_published.
 > main:0002_auto__add_field_photo_published
 - Loading initial data for main.
Installed 0 object(s) from 0 fixture(s)
```

Poza migracjami schematu, South pozwala także tworzych migracje danych (np. przenoszenie danych pomiędzy tabelami). W ten sposób zarządzając zmianami w bazie, można być pewnym takiego samego schematu na wszystkich maszynach.

ORM

Gdy model jest już stworzony i zmigrowany można zacząć wykonywać operacje CRUD (tworzenie, odczytywanie, aktualizacja, usuwanie). Każda klasa dziedzicząca po `django.db.models.Model` posiada obiekt `objects` będący instancją klasy `QuerySet`. Aby eksplorować możliwości ORM Django, najlepiej jest uruchomić interaktywną konsolę:

```
$ python manage.py shell_plus
```

Następnie można wyświetlić wszystkich użytkowników z bazy (użytkownik jest potrzebny do utworzenia nowej instancji `Photo`):

```
In [1]: User.objects.all()
Out[1]: [<User: suda>]

In [2]: user = User.objects.all()[0]

In [3]: user
Out[3]: <User: suda>
```

Jak widać jest tylko jeden użytkownik (stworzony podczas pierwszego uruchomienia `syncdb`). Teraz można stworzyć pierwszy obiekt `Photo`:

```
In [4]: photo = Photo(user=user)

In [5]: photo.title = u'Pierwsze zdjęcie'

In [6]: photo.save()

In [7]: Photo.objects.all()
Out[7]: [<Photo: Photo object>]
```

Stworzona w ten sposób instancja modelu, zostanie zapisana dopiero po wykonaniu metody `save()`. Innym sposobem jest stworzenie za pomocą metody `create()` własności `objects` która odrazu stworzy obiekt w bazie:

```
In [8]: Photo.objects.create(user=user, title=u'Drugie zdjęcie')
Out[8]: <Photo: Photo object>

In [9]: Photo.objects.all()
Out[9]: [<Photo: Photo object>, <Photo: Photo object>]
```

Można zauważyć, że przy ustawianiu tytułu zdjęcia, przed apostrofem została dodana litera `u`. Oznacza to, że ten ciąg znaków powinien być typu `Unicode` co w przypadku języka polskiego jest wymogiem.

W odróżnieniu do modelu `User` który wypisał nazwę użytkownika, model `Photo` wyświetla się jedynie jako `<Photo: Photo object>`. Aby to zmienić, należy dodać do modelu metodę `__unicode__`:

```
1 def __unicode__(self):
2         return self.title
```

Teraz po zrestartowaniu konsoli (niestety ładuje ona pliki Pythona przy starcie, więc po zmianach należy ją uruchomić ponownie) obiekty wyświetlają się bardziej opisowo:

```
In [1]: Photo.objects.all()
Out[1]: [<Photo: Pierwsze zdjęcie>, <Photo: Drugie zdjęcie>]
```

Poza listowaniem wszystkich obiektów, można wyciągnąć np:

```
In [1]: Photo.objects.first() # Pierwszy obiekt
Out[1]: <Photo: Pierwsze zdjęcie>

In [2]: Photo.objects.last() # Ostatni obiekt
Out[2]: <Photo: Drugie zdjęcie>

In [3]: Photo.objects.count() # Ilość obiektów
Out[3]: 2
```

lub pojedyńczy obiekt po jego kluczu głównym (`pk`):

```
In [4]: Photo.objects.get(pk=1)
Out[4]: <Photo: Pierwsze zdjęcie>
```

Można także filtrować obiekty:

```
In [1]: Photo.objects.filter(title__startswith=u'Pierwsze')
Out[1]: [<Photo: Pierwsze zdjęcie>]

In [2]: Photo.objects.filter(published=False)
Out[2]: [<Photo: Pierwsze zdjęcie>, <Photo: Drugie zdjęcie>]
```

Każde pole można filtrować po jego dokładnej zawartości lub używając specjalnych filtrów używających nazwy pola, dwóch podkreśleń (__) oraz jego nazwy. Jedne z najczęściej używanych:

- **contains/icontains** - sprawdzenie czy wartość zawiera podaną zmienną/test nie zwracający uwagi na wielkość liter
- **in** - sprawdzenie czy wartość zawiera się w przekazanej liście (np. `__in=[1, 2, 3]`)
- **gt/gte/lt/lte** - porównanie wartości liczbowych większych/większych lub równych/mniejszych/mniejszych lub równych do przekazanej liczby
- **startswith/endswith** - wyszukanie początku/końca wartości

Wyciągnięty obiekt można modyfikować:

```
In [1]: photo = Photo.objects.get(pk=1)

In [2]: photo.published = True

In [3]: photo.save()

In [4]: Photo.objects.filter(published=True)
Out[4]: [<Photo: Pierwsze zdjęcie>]
```

lub usunąć:

```
In [5]: photo.delete()

In [6]: Photo.objects.all()
Out[6]: [<Photo: Drugie zdjęcie>]
```

Wszystkie możliwe metody dostępne są pod adresem: https://docs.djangoproject.com/en/1.6/ref/models/querysets/.

Panel administracyjny

Funkcją w Django która najbardziej oszczędza czas jest panel administracyjny automatycznie generowany na podstawie modeli. Po uruchomieniu `runserver_plus`, należy otworzyć w przeglądarce adres http://127.0.0.1:8000/admin/ i zalogować się danymi podanymi przy tworzeniu użytkownika.

Jak widać panel już posiada możliwość zarządzania użytkownikami oraz grupami. Jedną rzecz jaką może przeszkadzać, jest jego język. Aby zmienić go na polski, należy w pliku `settings.py` zmienić wartość zmiennej `LANGUAGE_CODE` na `pl`:

```
LANGUAGE_CODE = 'pl'
```

Po odświeżeniu panelu, powinien być w języku polskim (`runserver` automatycznie restartuje się po zmianie plików Pythona).

Kolejnym krokiem jest dodanie modelu `Photo` do panelu. Aby tego dokonać należy stworzyć plik `admin.py` w katalogu aplikacji:

```
1  from django.contrib import admin
2  from .models import Photo
3
4  class PhotoAdmin(admin.ModelAdmin):
5      pass
6
7  admin.site.register(Photo, PhotoAdmin)
```

Słowo kluczowe `pass` zaraz po deklaracji klasy, informuje Pythona o zakończeniu jej deklarowania (brak wcięcia były uznany za błąd składni). Po zrestartartowaniu serwera (wymagane przy dodawaniu nowych plików) powinna pojawić się nowa sekcja *Main*. Aby dane miały większy sens, najlepiej dodać kilka obiektów używając wbudowanego formularza panelu administracyjnego, najlepiej ustawiając najróżniejsze wartości pól.

Django admin automatycznie tworzy czytelne nazwy z nazw obiektów/aplikacji. Aby jednak posiadać kod w języku angielskim ale opisy po polsku należy zmodyfikować model `Photo` dodając mu podklasę `Meta`:

```
1  class Meta:
2          verbose_name_plural = u'Zdjęcia'
3          verbose_name = u'Zdjęcie'
```

Nazwa modelu już wyświetlana jest poprawnie, lecz po wejściu w obiekt *Drugie zdjęcie* pola nadal są po angielsku. Aby to zmienić, należy ustawić polom atrybut `verbose_name`:

```
class Photo(models.Model):
        user = models.ForeignKey(User, null=False, blank=False, verbose_name=u'U
żytkownik')
        title = models.CharField(max_length=255, null=True, blank=True, verbose_
name=u'Tytuł')
        date = models.DateTimeField(auto_now_add=True, null=False, blank=False,
verbose_name=u'Data')
        image = models.ImageField(upload_to="images/", null=True, blank=True, ve
rbose_name=u'Obraz')
        published = models.BooleanField(default=False, verbose_name=u'Opublikowa
ne')
```

Teraz panel jest o wiele bardziej przyjazny polakom. Kolejnym krokiem byłoby wzbogacenie listy zdjęć o dodatkowe informacje na temat obiektu zamiast samej reprezentacji `__unicode__`. Można to zrobić podmieniając w pliku `admin.py` słowo kluczowe `pass` ustawieniem własności `list_display`:

```
list_display = ('title', 'date', 'published')
```

Teraz na liście wyświetlane są trzy kolumny, po których można sortować wszystkie obiekty. Poza sortowaniem, Django obsługuje także filtrowanie, które można zdefiniować podobnie jak pola wyświetlane na liście:

```
list_filter = ('date', 'published')
```

Z prawej strony listy pojawiła się lista filtrów. Kolejną przydatną funkcjonalnością w panelu administracyjnym jest wyszukiwanie. Aby je dodać, należy zadeklarować które pola mają być wyszukiwane:

```
search_fields = ('title',)
```

Nad listą znajduje się teraz pole na wyszukiwaną frazę. Zaraz pod nią widać rozwijane pole z akcjami. Standardowo znajduje się tam możliwość usunięcia zaznaczonych zdjęć ale można tutaj dodawać własne akcje, jak np. opublikowanie zaznaczonych. Aby dodać akcję należy przed deklaracją klasy `PhotoAdmin` dodać funkcję wykonującą akcję:

```
def publish(modeladmin, request, queryset):
    queryset.update(published=True)
publish.short_description = u'Opublikuj zaznaczone'
```

oraz dodać ją za `search_fields`:

```
actions = (publish,)
```

Lista zdjęć jest już dosyć rozbudowana, można zająć się wzbogacaniem widoku dodawania/edycji danych. Pierwszą poprawką która na tym etapie nie sprawia problemu ale przy większej skali może przeszkadzać to kontrolka używana przy polach `ForeignKey`. Standardowo jest to pole rozwijane zawierające wszystkie możliwe opcje. Niestety jeśli tabela obca

zawiera tysiące i więcej rekordów, może to spowodować bardzo długie ładowanie się edycji. Bardzo prostym rozwiązaniem jest podanie takich pól w `raw_id_fields` ktore zastąpi listę bardziej rozwiniętą kontrolką:

```
raw_id_fields = ('user',)
```

Innymi przydatnymi własnościami które w przypadku tego modelu nie są potrzebne, są bardzo przydatne to para `fields` i `exclude` które pozwalają wyświetlać tylko podane pola lub wykluczyć je z edycji w panelu.

Pełna dokumentacja panelu administracyjnego dostępna jest pod adresem:
https://docs.djangoproject.com/en/1.6/ref/contrib/admin/.

Autoryzacja oraz zarządzanie użytkownikami

Django posiada wbudowane klasy do zarządzania użytkownikami oraz ich grupami. Pozwala to w łatwy sposób autoryzować i ograniczać dostęp do funkcji zarówno w projekcie jak i panelu administracyjnym. Na początek można wyświetlić widoki logowania/wylogowania. Django dostarcza je także. W głównym pliku `urls.py` pod linią panelu administracyjnego należy dopisać:

```
1 url(r'^login/$', 'django.contrib.auth.views.login', {'template_name': 'admin/log
2 in.html'}, name='login'),
  url(r'^logout/$', 'django.contrib.auth.views.logout', {'next_page': '/'}, name='
  logout'),
```

W tym przykładzie użyty jest wygląd logowania z panelu administracyjnego, ale można to bardzo łatwo zmienić. Następnie można ustawić adres na jaki użytkownik zostanie przekierowany po logowaniu, dodając w `settings.py` linikję:

```
1 LOGIN_URL = '/login/'
2 LOGIN_REDIRECT_URL = '/'
```

Teraz pod adresem http://127.0.0.1:8000/login/ widoczna jest strona logowania. Gdy użytkownicy mają możliwość logowania, można dodać pierwszą autoryzację, np. w widoku dodawania zdjęcia. W pliku `views.py` należy dodać import:

```
from django.contrib.auth.decorators import login_required
```

oraz widok:

```
1 @login_required
2 def upload(request):
3         return HttpResponse('Hello %s!' % request.user.username)
```

Widok ten używa dekoratora `@login_required` który jeśli użytkownik próbując wyświetlić ten widok nie będąc zalogowanym, zostanie przekierowany na stronę logowania. Jeśli jest zalogowany, własność `user` zmiennej `request` powinna zawierać instancję klasy `User`. Jeśli nie jest zalogowany, będzie to instancja `AnonymousUser`, najlepszym sposobem na stwierdzenie czy użytkownik jest zalogowany jest użycie metody `is_authenticated()`:

```
if request.user.is_authenticated():
        # zalogowany
else:
        # nie zalogowany
```

Jako, że dekorator wykonuje takie sprawdzenie, można być pewnym, że użytkownik jest zalogowany i posiada własność `username` która zostaje wyświetlana. Kolejnym krokiem jest dodanie widoku do pliku `urls.py` aplikacji przed widokiem `hello_name`:

```
url(r'^upload/$', views.upload, name='upload'),
```

Teraz po wejściu na adres http://127.0.0.1:8000/upload/ użytkownik niezalogowany, zostanie przekierowany na stronę logowania a zalogowany użytkownik zostanie przywitany swoim loginem. Oczywiście po zalogowaniu nastąpi przekierowanie na widok który wymagał logowania.

Widoki oraz formularze

Trzecim, nieomówionym komponentem architektury MTV jest szablon. Django posiada własny język szablonów którego głównym założeniem było wyeliminowanie logiki biznesowej z warstwy prezentacji, dlatego np. nie można wykonywać kodu Pythona wewnątrz szablonów. Pliki szablonów są przechowywane w katalogu `templates` aplikacji. Można zatem stworzyć pierwszy plik bazowy który będzie stanowił podstawę dla innych. Zwyczajowo nazywa się go `base.html`:

```
{% load bootstrap3 %}
<!doctype html>
<html lang="en">
<head>
        <meta charset="UTF-8">
        <title>{% if title %}{{ title|capfirst }} - {% endif %}Photogram</title
>
        {% bootstrap_css %}
    {% bootstrap_javascript %}
</head>
<body>
        <header class="navbar">
                <div class="container">
                        <a href="/" class="navbar-brand">Photogram</a>
                        <ul class="nav navbar-nav navbar-right">
                                {% block navbar-right %}{% endblock %}
                                <li><a href="{% url "logout" %}" class="">Wylog
uj &raquo;</a></li>
                        </ul>
                </div>
        </header>
        <div class="container">
                {% bootstrap_messages %}
                {% block content %}{% endblock %}
        </div>
        <footer><div class="container"><p>&copy;{% now "Y" %} me</p></div></foo
ter>
</body>
</html>
```

W języku szablonów Django istnieją trzy specjalne elementy:

- **zmienne** - zawarte pomiędzy podwójnymi nawiasami klamrowymi (`{{` i `}}`). Renderują one zawartość zmiennej. Jeśli w jej nazwie znajduje się kropka, Django sprawdzi czy zmienna posiada taki indeks, atrybut lub metodę (w tej kolejności).
- **filtry** - służą do modyfikowania wartości zmiennej przed renderowaniem. Istnieje wiele wbudowanych filtrów (można także łatwo dopisywać własne), w powyższym przykładzie użyto `capfirst` który zamienia pierwszą literę na wielką.
- **tagi** - zawarte są pomiędzy `{%` a `%}` dają większą kontrolę nad szablonem. Także można pisać własne. Najważniejsze tagi służą do iterowania, warunków (`if`), tworzenia bloków które mogą zostać nadpisane (`block`) czy wyświetlania tekstu (`now`).

Jedne z przydatniejszych filtrów to:

- **capfirst**, **lower**, **upper** - zmieniają wielkość znaków
- **first**, **last** - wyświetlenie tylko pierwszego/ostatniego elementu
- **escape** - zamiana znaków specjalnych HTML na encje
- **join** - łączenie list
- **truncatechars**, **truncatewords**, **truncatewords_html** - ucinanie ciągu znaków
- **linebreaks**, **linebreaksbr** - zamiana znaku `\n` na tag HTML
- **slugify** - tworzenie tzw. sluga czyli ciągu składającego się tylko ze znaków ANSI i myślników

Najczęściej używane tagi to:

- **if**, **else**, **endif** - warunki
- **for**, **empty**, **endfor** - iterowanie oraz warunek dla pustego zbioru
- **block**, **endblock** - deklaracja lub podmiana (w przypadku dziedziczenia) bloku
- **extends** - dziedziczenie innego szablonu
- **url** - generowanie adresu URL do widoku
- **load** - ładowanie zewnętrznych tagów/filtrów

Lista wszystkich tagów oraz filtrów, znajduje się pod adresem: https://docs.djangoproject.com/en/1.6/ref/templates/builtins/.

W powyższym przykładzie użyto zewnętrznej biblioteki `django-bootstrap3` pozwalającej szybko zintegrować projekt z frameworkiem Bootstrap http://getbootstrap.com/. Należy ją oczywiście doinstalować:

```
$ pip install django-bootstrap3
```

oraz dodać do `INSTALLED_APPS` w `settings.py`:

```
INSTALLED_APPS = (
    ...
    'bootstrap3',
)
```

Stworzony szablon należy wyrenderować. Django posiada skrót króry renderuje szablon i zwraca `HttpResponse` z nim:

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

Można zatem zmienić treść widoku `upload()` na:

```
return render(request, 'base.html', {
        'title': u'przesyłanie zdjęć'
    })
```

Ostatni przekazany parametr to tzw. kontekst szablonu czyli słownik zawierający zmienne które mają zostać do niego przekazane. Po odświeżeniu powinna pojawić się strona z wyrenderowanego szablonu.

Formularze

Mając zdefiniowany model zamiast żmudnie kodować formularz w HTML, można użyć specjalnych klas w Django o nazwie `ModelForm` który potrafi generować je automatycznie. Należy więc otworzyć plik `forms.py` i umieścić w nim:

```
# -*- encoding: utf-8 -*-

from django import forms

from .models import Photo

class PhotoForm(forms.ModelForm):
        class Meta:
                model = Photo
                fields = ['title', 'image']

        def __init__(self, *args, **kwargs):
                super(PhotoForm, self).__init__(*args, **kwargs)
                self.fields['title'].required = True
                self.fields['image'].required = True
```

Widać tu przykład nadpisywania konstruktora z wykonaniem go z superklasy. Jako, że `title` i `image` w modelu mają wlasność `blank` równą `True`, nie są one polami wymaganimi. W tym formularzu powinny być, więc to zachowanie zostaje ręcznie zmienione. Pełna dokumentacja pól formularzy, dostępna jest pod adresem: https://docs.djangoproject.com/en/1.6/ref/forms/fields/.

Następnie należy zmodyfikować widok tak aby używał formularza który należy zaimportować:

```
from .forms import PhotoForm
```

zainicjować i przekazać do szablonu:

```
def upload(request):
        if request.method == 'POST':
                pass
        else:
                form = PhotoForm()

        return render(request, 'upload.html', {
        'title': u'przesyłanie zdjęć',
        'form': form
    })
```

Szablon `upload.html` powinien wyglądać tak:

```
{% extends "base.html" %}
{% load bootstrap3 %}
{% block content %}
        <form action="{% url "upload" %}" method="post" class="form" enctype="m
ultipart/form-data">
            {% csrf_token %}
                {% bootstrap_form form %}
                {% buttons %}
                <button type="submit" class="btn btn-primary">
                    {% bootstrap_icon "upload" %} Wyślij
                </button>
            {% endbuttons %}
        </form>
{% endblock %}
```

Wewnątrz formularza występuje tag `{% csrf_token %}` który służy zabezpieczeniu przed atakami typu CSRF (przekierowywanie użytkownika na atakowaną stronę bez jego wiedzy). Dodaje on ukryte pole formularza podobne do:

```
<input type='hidden' name='csrfmiddlewaretoken' value='qGfJZH9qYlnst9mCd3ZdsamrlY
7qmdyQ' />
```

Token jest automatycznie weryfikowany przez middleware Django, należy jednak pamiętać o dodawaniu tego tagu przy formularzach używających metody **POST**.

Po odświeżeniu strony widać ładny formularz dodawania zdjęć. Aby obsłużyć ich ładowanie na serwer, należy podmienić słowo kluczowe `pass` na:

```
form = PhotoForm(request.POST, request.FILES)
if form.is_valid():
        form.instance.user = request.user
        form.instance.published = True
        form.save()
        messages.success(request, u'Zdjęcie załadowane')
        return redirect(reverse('hello'))
else:
        messages.error(request, u'Proszę poprawić formularz')
```

Jak widać użyto tu kilka metod po raz pierwszy, należy je więc zaimportować:

```
from django.shortcuts import redirect
from django.core.urlresolvers import reverse
from django.contrib import messages
```

Dodano nimi dwie funkcje:

- przekierowanie po zapisaniu formularza które można uzyskać używając skrótu `django.shortcuts.redirect` a adres URL został wygenerowany automatycznie z pliku `urls.py` używając nazwy widoku oraz metody `reverse` która jest odpowiednikiem tagu `url` w szablonie.
- dodanie wiadomości do wyświetlenia dla użytkownika. Jest to wiadomość typu **flash** która znika po odświeżeniu strony.

Można także zauważyć używanie własności `instance` formularza. Własność ta zawiera instancję modelu, którą można operować przed jego zapisaniem.

Taki formularz powinien działać prawidłowo, można więc spróbować wpisywać prawidłowe i nieprawidłowe dane aby sprawdzić jak działa. Na razie dodane zdjęcia można zobaczyć w panelu administracyjnym ale następnym krokiem jest podmiana widoku `hello` na listę ostatnio dodanych zdjęć:

```
from .models import Photo

def hello(request):
        return render(request, 'index.html', {
        'title': u'ostatnie zdjęcia',
        'photos': Photo.objects.filter(published=True).order_by('-date')
    })
```

Następnie należy uzupełnić szablon:

```
{% extends "base.html" %}
{% load bootstrap3 %}
{% block navbar-right %}<li><a href="{% url "upload" %}" class="btn btn-primary
">Dodaj zdjęcie</a></li>{% endblock %}
{% block content %}
        {% for photo in photos %}
                <article>
                        <h1>{{ photo.title }}</h1>
                        <img src="{{ photo.image.url }}" alt="{{ photo.title }}
">
                        <p>Autor: {{ photo.user }}, {{ photo.date|timesince }}
temu</p>
                </article>
        {% endfor %}
{% endblock %}
```

Ostatnia rzecz jaka pozostała to włączenie serwowania załadowanych plików przez `runserver_plus` zmieniając główny plik `urls.py` na:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin
admin.autodiscover()

from django.conf import settings
from django.conf.urls.static import static

urlpatterns = patterns('',
    url(r'^admin/', include(admin.site.urls)),

        url(r'^login/$', 'django.contrib.auth.views.login', {'template_name': '
admin/login.html'}, name='login'),
        url(r'^logout/$', 'django.contrib.auth.views.logout', {'next_page': '/'
}, name='logout'),

    url(r'^', include('main.urls')),

) + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

oraz dodając do `settings.py` linijkę:

```
MEDIA_URL = '/media/'
```

Testy

Bardzo ważną częścią tworzenia oprogramowania jest jego jakość. Jedną z metod jej utrzymywania jest TDD czyli programowanie gdzie w pierwszej kolejności pisze się testy sprawdzające prawidłowość kodu a następnie pisze się kod który je przechodzi. Django posiada wbudowaną obsługę testów jednostkowych. Testy powinny znajdować się w pliku `tests.py` w katalogu aplikacji. Przykładowy test sprawdzający widok `hello()` pod kątem wyświetlania nieopublikowanych zdjęć:

```python
# -*- encoding: utf-8 -*-

from django.test import TestCase
from django.contrib.auth.models import User
from django.test import Client

from .models import Photo
from .views import hello_name

class MainTestCase(TestCase):
    def setUp(self):
            user = User.objects.create_user('tyrion', 'tyrion@kingslanding.gov'
, 'shae')
        Photo.objects.create(user=user, title=u'Photo without image', published
=False)

    def test_empty_photos_list(self):
            """Sprawdzenie czy zdjęcia z publisjed=False nie są pokazywane"""
            client = Client()
            response = client.get('/')
            self.assertEqual(response.content.find('<article>'), -1)
```

Aby uruchomić testy, wystarczy wykonać:

```
$ python manage.py test
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.166s

OK
Destroying test database for alias 'default'...
```

Praca z repozytorium Git

Istnieje wiele systemów kontroli wersji, ale żaden nie jest aktualnie tak popularny jak `git`. Darmowy hosting repozytorów na GitHub lub Bitbucket (darmowe do 10 prywatnych repozytorów), obsługa deployowania na Heroku, Microsoft Azure oraz rozproszona architektura czynią z Gita nieodłączne narzędzie programisty.

Instalacja

Windows

Świetnym klientem Git dla Windows jest msysgit http://msysgit.github.io/ który daje zarówno tekstowy jak i graficzny interfejs.

Linux

Paczka `git` powinna być dostępna w każdym managerze pakietów.

OS X

Git jest dostępny jako część narzędzi lini poleceń Xcode.

Podstawowe użycie

W odróżnieniu od CVS i SVN, Git jest rozproszony co w praktyce oznacza, że każdy katalog roboczy jest samodzielnym repozytorium. Jeśli więc pracuje się samodzielnie, można lokalnie przechowywać całą historię kodu bez używania serwera. Aby zainicjować repozytoriu Git, należy w katalogu projektu wykonać:

```
$ git init
Initialized empty Git repository in /Users/suda/.virtualenvs/photogram/photogram/.git/
```

Od tego momentu, aktualny katalog staje się repozytorium. Można zatem sprawdzić jego stan:

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        db.sqlite3
        images/
        main/
        manage.py
        photogram/

nothing added to commit but untracked files present (use "git add" to track)
```

Żaden plik nie został jeszcze dodany do repozytorium (nie jest zatem śledzony), lecz widać jeden plik `db.sqlite3` który wygląd na plik który nie powinine znajdować się w repo. Dlatego należy go dodać do pliku `.gitignore` w katalogu głównym projektu razem z katalogiem `images` zawierającym wrzucone zdjęcia oraz zkompilowanymi plikami Pythona:

```
*.sqlite3
images/
*.pyc
```

Teraz można dodać wszystkie pozostałe pliki:

```
$ git add .
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   main/__init__.py
        new file:   main/admin.py
        new file:   main/forms.py
        new file:   main/migrations/0001_initial.py
        new file:   main/migrations/0002_auto__add_field_photo_published.py
        new file:   main/migrations/__init__.py
        new file:   main/models.py
        new file:   main/templates/base.html
        new file:   main/templates/index.html
        new file:   main/templates/upload.html
        new file:   main/urls.py
        new file:   main/views.py
        new file:   manage.py
        new file:   photogram/__init__.py
        new file:   photogram/settings.py
        new file:   photogram/urls.py
        new file:   photogram/wsgi.py
```

Jak widać pliki zostały dodane do sekcji **Changes to be committed**. Teraz należy je tylko zatwierdzić:

```
$ git commit -m "Pierwszy commit"
```

Po każdej zmianie wystarczy powtarzać cykl `git add` i `git commit`.

Aby zacząć współdzielić kod, należy dodać serwer. Można założyć konto na Githubie (darmowe dla otwartych repozytoriów) lub Bitbuckecie (darmowe dla otwartych i do 10 prywatnych), następnie stworzyć na nim repozytorium. Na stronie podsumowania będzie widoczny adres repo zaczynający się od `git@` (zwany czasami **clone URL**) który należy skopiować do terminala:

```
$ git remote add origin ADRES
```

`origin` jest przyjętą nazwą dla głównego serwera. Po dodaniu serwera, należy wysłać na niego lokalne zmiany:

```
$ git push origin master
Counting objects: 22, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (22/22), 6.19 KiB | 0 bytes/s, done.
Total 22 (delta 1), reused 0 (delta 0)
To git@github.com:suda/warsztat-django.git
 * [new branch]      master -> master
```

Jedną z ważniejszych cech Gita są gałęzie. Po utworzeniu repozytorium, automatycznie tworzona jest główna gałąź o nazwie `master`. Istnieje wiele metodologii używania gałęzi, lecz najpopularniejszą jest tzw. **feature branch** w której każda funkcjonalność posiada własną gałąź tworzoną przed rozpoczęciem nad niej pracy:

```
$ git checkout -b stronicowanie master
Switched to a new branch 'stronicowanie'
```

Po dokonaniu zmian, w tym przypadku dodania stronicowania do listy zdjęć używając `django-bootstrap-pagination` które należy zainstalować:

```
$ pip install django-bootstrap-pagination
```

dodać do `INSTALLED_APPS` w `settings.py`:

```
INSTALLED_APPS = (
    ...
    'bootstrap_pagination',
)
```

oraz dodać `django.core.context_processors.request` do `TEMPLATE_CONTEXT_PROCESSORS` w `settings.py`:

```
from django.conf import global_settings
TEMPLATE_CONTEXT_PROCESSORS = global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
    "django.core.context_processors.request",
)
```

podmienić widok `hello()` na:

```
def hello(request):
        photos = Photo.objects.filter(published=True).order_by('-date')
        paginator = Paginator(photos, 1)

        page = request.GET.get('page')
        try:
                photos = paginator.page(page)
        except PageNotAnInteger:
                photos = paginator.page(1)
        except EmptyPage:
                photos = paginator.page(paginator.num_pages)

        return render(request, 'index.html', {
        'title': u'ostatnie zdjęcia',
        'photos': photos,
        'paginator': paginator
    })
```

zaimportować w widokach:

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
```

i na końcu dodać do pliku `index.html` załadowanie biblioteki:

```
{% load bootstrap_pagination %}
```

oraz jej użycie po pętli:

```
{% bootstrap_paginate photos %}
```

Po sprawdzeniu, że stronicowanie działa, można je zatwierdzić:

```
$ git status
On branch stronicowanie
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   main/templates/index.html
        modified:   main/views.py
        modified:   photogram/settings.py

no changes added to commit (use "git add" and/or "git commit -a")
$ git add .
$ git commit -m "Stronicowanie"
[stronicowanie 9e8c347] Stronicowanie
 3 files changed, 23 insertions(+), 2 deletions(-)
```

A następnie wysłać na serwer zaktualizowaną gałąź:

```
$ git push -u origin stronicowanie
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 1.09 KiB | 0 bytes/s, done.
Total 8 (delta 5), reused 0 (delta 0)
To git@github.com:suda/warsztat-django.git
 * [new branch]      stronicowanie -> stronicowanie
Branch stronicowanie set up to track remote branch stronicowanie from origin.
```

Gdy gałąź jest już aktualna na serwerze, należy stworzyć tzw. **pull request** na GitHubie/Bitbuckecie, który da znać reszcie zespołu o zmianach, pozwoli je sprawdzić i jeśli wszystko jest ok, złączyć je z gałęzią `master`.

Tips & tricks

Werkzeug

Używając komendy `runserver_plus` "pod maską" używana jest biblioteka WSGI `Werkzeug` która pozwala m.in. bardzo wygodny sposób odpluskwiania kodu. Gdy podczas przetwarzania zapytania zostanie wywołany wyjątek wyświetli się komunikat z dokładnymi informacjami n.t. błędu. Można także specjalnie wywołać tą stronę dodając przed `return` w widoku `hello()`:

```
raise Exception()
```

Po odświeżeniu strony głównej widoczny jest **traceback** z liniami kodu kolejno odpowiadającymi za dojście do wyjątku. Kliknięcie w linię, pokazuje otaczający wyjątek kod, lecz najbardziej przydatna jest ikonka terminala z prawej strony linii. Otwiera ona interaktywną konsolę zatrzymaną w tej linii. Pozwala ona na podejrzenie zawartości zmiennych a nawet wykonywanie funkcji:

```
[console ready]
>>> page
u'1'
>>> photos
<Page 1 of 2>
>>> Photo.objects.filter(published=True).order_by('-date')
[<Photo: Joffrey>, <Photo: Tommen>]
```

Jest to bardzo wygodny sposób na odnalezienie błędów bez cyklu zmień/odśwież.