

Kurs rozszerzony języka Python

Środowisko Django, cz. 2

Marcin Młotkowski

12 stycznia 2018

Plan wykładu

- 1 Administracja
- 2 Formularze
 - Trochę teorii
 - Obsługa żądań GET
 - Obiekty formularzowe
- 3 Różne
 - Zarządzanie sesjami
 - Polonizacja
- 4 Testowanie
 - Testy jednostkowe
 - Symulowanie klienta

Plan wykładu

- 1 Administracja
- 2 Formularze
 - Trochę teorii
 - Obsługa żądań GET
 - Obiekty formularzowe
- 3 Różne
 - Zarządzanie sesjami
 - Polonizacja
- 4 Testowanie
 - Testy jednostkowe
 - Symulowanie klienta

Przypomnienie

Co implementujemy

System zapisy:

- wykładowcy;
- studenci;
- wykłady.

Administrowanie aplikacją

settings.py

```
MIDDLEWARE_CLASSES = (  
    'django.middleware.common.CommonMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware', )  
INSTALLED_APPS = ( 'django.contrib.auth',  
    'django.contrib.contenttypes', 'django.contrib.sessions',  
    'django.contrib.sites', 'django.contrib.admin', 'wyklad.zapisy', )
```

Umożliwia zarządzanie aplikacją.

Administrowanie aplikacją

settings.py

```
MIDDLEWARE_CLASSES = (  
'django.middleware.common.CommonMiddleware',  
'django.contrib.sessions.middleware.SessionMiddleware',  
'django.contrib.auth.middleware.AuthenticationMiddleware', )  
INSTALLED_APPS = ( 'django.contrib.auth',  
'django.contrib.contenttypes', 'django.contrib.sessions',  
'django.contrib.sites', 'django.contrib.admin', 'wyklad.zapisy', )
```

Umożliwia zarządzanie aplikacją.

Podłączenie administracji

Wyszukanie w aplikacjach informacji, czy chcą być zarządzane przez moduł `admin`.

`zapisy/admin.py`

```
from django.contrib import admin
```

```
admin.site.register(Wykladowca)
```

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

```
admin.site.register(Wyklad)
```


Plan wykładu

- 1 Administracja
- 2 Formularze
 - Trochę teorii
 - Obsługa żądań GET
 - Obiekty formularzowe
- 3 Różne
 - Zarządzanie sesjami
 - Polonizacja
- 4 Testowanie
 - Testy jednostkowe
 - Symulowanie klienta

Podstawowe żądania HTTP

GET

GET <ścieżka do pliku> HTTP 1.1

GET <ścieżka>?query=python&subquery=django HTTP 1.1

Podstawowe żądania HTTP

GET

GET <ścieżka do pliku> HTTP 1.1

GET <ścieżka>?query=python&subquery=django HTTP 1.1

POST

POST /login HTTP 1.1

login=admin

password=qwerty

Inne typy żądań

- HEAD
- PUT
- DELETE
- TRACE
- CONNECT
- OPTIONS

Analiza żądań

Przypomnienie: widoki

```
def hello(request):
```

```
...
```

Analiza żądań

Przypomnienie: widoki

```
def hello(request):  
    ...
```

Obiekt Request, atrybuty

- path, get_full_path() (z zapytaniem)
- method: może być GET bądź POST
- GET, POST: podobne do słowników

Kiedy GET, kiedy POST

GET

Wyświetlenie danych, bez ich zmiany (np. wyszukiwanie).

POST

Efekty uboczne: zmiana danych, wysłanie maila z formularza.

Składnia

```
<form action="/search/" method="get">  
  <input type="text" name="query">  
  <input type="submit" value="Szukaj">  
</form>
```

Widok: urls.py

```
urlpatterns = patterns("",  
    # ...  
    (r"^search-form/$", views.search_form),  
    # ...  
)
```


Implementacja widoków

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from mysite.zapisy.models import Wyklad

def search_form(request):
    return render_to_response('search_form.html')

def search(request):
    if 'query' in request.GET and request.GET['query']:
        query = request.GET["query"]
        wykłady = Wyklad.objects.filter(tytul__icontains=query)
        return render_to_response('search_results.html',
                                   {'wykłady': wykłady, 'query': query})
    else:
        return render_to_response('search_form.html',
                                   {'error': True})
```

szablon search_form.html

```
{% if error %}  
    <p style="color: red;">Proszę zadać pytanie</p>  
{% endif %}  
<form action="/search/" method="get">  
    <input type="text" name="query">  
    <input type="submit" value="Szukaj">  
</form>
```

Obiekty formularzowe: co to takiego

- obiekty, z których można łatwo wyprodukować formularze
- można konfigurować różne parametry (rozmiar pola, opcjonalność pól);
- weryfikacja podawanych danych

Obiekty formularzowe

```
from django import forms
```

```
class ContactForm(forms.Form):  
    subject = forms.CharField(max_length=100)  
    email = forms.EmailField(required=False,  
                             label="Podaj email")  
    message = forms.CharField(widget=forms.Textarea)
```

Sposoby walidacji podawanych danych

```
class ContactForm(forms.Form):  
    ...  
    def clean_email(self):  
        em = self.cleaned_data['email']  
        if em niepoprawny:  
            raise forms.ValidationError('Niepoprawny email!')  
        return em
```

Użycie obiektu formularza

Metody i pola obiektu

- `.as_table`: do włożenia do tabeli
- `.as_ul`: do włożenia do wyliczenia ``
- `.as_p`: wkłada w znaczniki `<p>`

Osadzenie formularza w szablonie

Przykład użycia

```
{% if form.errors %}
<p style="color: red;">
Popraw błędy {{ form.errors|pluralize }}.
</p>
{% endif %}
<form action="" method="post">
  <table>
    {{ form.as_table }}
  </table>
  <input type="submit" value="Submit">
</form>
```

Wiązanie widoku z formularzem

```
from django.shortcuts import render_to_response
from mysite.contact.forms import ContactForm

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            operacja(cd['subject'], cd['message'],
                    cd.get('e-mail', 'noreply@example.com'))
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render_to_response('contact_form.html', {'form': form})
```


Inicjowanie wartościami początkowymi

```
form = ContactForm(initial=  
    { subject='Mam problem', message='Opisz problem' })
```

Plan wykładu

- 1 Administracja
- 2 Formularze
 - Trochę teorii
 - Obsługa żądań GET
 - Obiekty formularzowe
- 3 **Różne**
 - Zarządzanie sesjami
 - Polonizacja
- 4 Testowanie
 - Testy jednostkowe
 - Symulowanie klienta

Sesje

Sesja: ciąg akcji związanych z jednym użytkownikiem; na przykład zakupy w sklepie internetowym.

Realizacja

Ciasteczka

Obsługa sesji w Django

Co potrzebujemy:

- w `MIDDLEWARE_CLASSES` ma być `'django.contrib.sessions.middleware.SessionMiddleware'`.
- w `INSTALLED_APPS` ma być `'django.contrib.sessions'`

Jak używać sesji

Atrybut `request.session`

Jest to atrybut zachowujący się podobnie do słowników.

Przykład

Logowanie

```
def login(request):  
    if request.method != 'POST':  
        raise Http404('Tylko POSTs są dopuszczalne')  
    try:  
        m = User.objects.get(username=request.POST['username'])  
        if m.password == request.POST['password']:  
            request.session['user_id'] = m.id  
            return HttpResponseRedirect('/you-are-logged-in/')  
    except User.DoesNotExist:  
        return HttpResponse("Nieudane logowanie :-(")
```

Przykład

Logowanie

```
def login(request):  
    if request.method != 'POST':  
        raise Http404('Tylko POSTs są dopuszczalne')  
    try:  
        m = User.objects.get(username=request.POST['username'])  
        if m.password == request.POST['password']:  
            request.session['user_id'] = m.id  
            return HttpResponseRedirect('/you-are-logged-in/')  
    except User.DoesNotExist:  
        return HttpResponse("Nieudane logowanie :-(")
```

`request.session['user_id']` służy do identyfikacji, czy użytkownik jest zalogowany.

Wylogowanie

```
def logout(request):  
    try:  
        del request.session['user_id']  
    except KeyError:  
        pass  
    return HttpResponse(" Jesteś wylogowany")
```


Podstawowe pojęcia

Internacjonalizacja (I18N)

Przygotowanie programu do tworzenia różnych wersji językowych.

Podstawowe pojęcia

Internacjonalizacja (I18N)

Przygotowanie programu do tworzenia różnych wersji językowych.

Lokalizacja

Stworzenie wersji językowej.

Włączanie/wyłączanie internacjonalizacji

```
USE_I18N = True  
TEMPLATE_CONTEXT_PROCESSORS = (  
    'django.core.context_processors.i18n'  
)
```

Tłumaczenie napisów w widokach

```
from django.utils.translation import ugettext as _  
  
def widoczek(request):  
    output = _("Welcome to my site.")  
    return HttpResponse(output)
```

Tłumaczenia w szablonach

Tłumaczenie pojedynczego napisu

```
<title>{% trans "This is the title." %}</title>
```

Tłumaczenie większych fragmentów

```
{% blocktrans %}  
This string will have {{ value }} inside.  
{% endblocktrans %}
```

Tłumaczenie

`django-admin.py makemessages -l pl`

tworzy plik `locale/pl/LC_MESSAGES/django.po`. String z pliku źródłowego jest kluczem, my powinniśmy uzupełnić o tłumaczenie

Tłumaczenie

`django-admin.py makemessages -l pl`

tworzy plik `locale/pl/LC_MESSAGES/django.po`. String z pliku źródłowego jest kluczem, my powinniśmy uzupełnić o tłumaczenie

Po wpisaniu tłumaczeń

Kompilacja: `django-admin.py compilemessages`

Plan wykładu

- 1 Administracja
- 2 Formularze
 - Trochę teorii
 - Obsługa żądań GET
 - Obiekty formularzowe
- 3 Różne
 - Zarządzanie sesjami
 - Polonizacja
- 4 Testowanie
 - Testy jednostkowe
 - Symulowanie klienta

Poziomy testowania

Testy jednostkowe

Patrz: wykład 9: unittest

Poziomy testowania

Testy jednostkowe

Patrz: wykład 9: unittest

Testy klienckie

Symulacja działania przeglądarki.

Mała sugestia

Zamiast

```
import unittest
```

lepiej

```
from django.test import TestCase
```

Różnice między unittest a django.test

- automatyczne ładowanie danych testowych do bazy testowej;
- Każdy test jest odrębną transakcją;
- dodatkowe asercje do testowania HTML'a, przekierowań HTTP, etc.

Przykład

```
from django.test import TestCase
from wyklad.zapisy.models import Wyklad

class WykladTestCase(TestCase):
    def setUp(self):
        Wyklad.objects.create(nazwa='Python', ects=3)

    def test_zapisu(self):
        py = Wyklad.objects.get(nazwa='Python')
        self.assertEqual(py.ects, 3)
```

Przykład

```
from django.test import TestCase
from wyklad.zapisy.models import Wyklad

class WykladTestCase(TestCase):
    def setUp(self):
        Wyklad.objects.create(nazwa='Python', ects=3)

    def test_zapisu(self):
        py = Wyklad.objects.get(nazwa='Python')
        self.assertEqual(py.ects, 3)
```

```
./manage.py test
```

Wyszukuje wszystkie pliki test*.py i traktuje je jako elementy zestawu testów.

Testowa baza danych

Testowa baza danych (*test_baza produkcyjna*) jest czyszczona po każdym teście.

Dane testowe

Dane testowe z istniejącej bazy danych:

```
manage.py syncdb
```

tworzy plik `initial_data` z danymi

Te dane są domyślnie ładowane podczas testów.

Inne dane testowe

```
from django.test import TestCase
from wyklad.zapisy.models import Wyklad

class WykladTestCase(TestCase):
    fixtures = ['jesien.json', 'wiosna']
    ...
```

Klient testowy

`django.test.client.Client`

Symuluje wysyłanie zapytań GET/POST.

Klient testowy

`django.test.client.Client`

Symuluje wysyłanie zapytań GET/POST.

Można też używać narzędzi typu Selenium.

Przykład scenariusza

```
from django.test.client import Client
student = Client()
response = student.post('/login/',
    {'username': 'i123456', 'password': '123456'})
print response.status_code # 200
response = student.get('/student/profil/')
print response.content
```

Uwagi

Scenariusze można opakować w przypadki testowe i korzystać z dodatkowych asercji z `django.test.TestCase`

Uwagi

Scenariusze można opakować w przypadki testowe i korzystać z dodatkowych asercji z `django.test.TestCase`

django dostarcza specjalizowanych klas do implementacji różnych testów wraz z sugestiami w jakich sytuacjach ich używać.