Wykład 6: Formularze w Django i obsługa sesji w HTTP

Date: 2015-11-16
tags: zaj6, wykład, materiały
category: materiały

> **Note**
> Wykład do pobrania również w wersji PDF.

> **Note**
> Niektóre przykłady zapożyczone z dokumentacji django (tak w ogóle dokumentacja Django jest wyjątkowo dobra, więc polecam).

Spis treści

Formularze w Django

Jak zapewne zauważyliście, obrabianie formularzy HTML z obsługą błędów jest dość żmudne (nie jest trudne, ale kodu do obsługi robi się sporo).

Django dostarcza abstrakcję, pozwalającą zautomatyzować:

- Tworzenie formularzy
- Ich walidację po stronie aplikacji
- Konwersję danych

Abstrakcją tą są formularze Django.

Tworzenie formularzy Django

Najprostszy formularz Django:

```
from django import forms

class NameForm(forms.Form):
    your_name = forms.CharField(label='Your name', max_length=100)
```

Składnia jest **prawie taka sama**, jak modeli (potem pokażę jak automatycznie tworzyć formularze z modeli).

> **Note**
> Uwaga! Mimo podobieństw formularze i modele korzystają z zupełnie innej paczki Django (modele z `django.db.models`, formularze z `django.forms`).

Praca z formularzami w widokach

Typowy widok w Django wygląda tak (przeczytajcie komentarze!):

```
def view(request):
    if request.method == 'POST': # Jeśli zapytanie jest post to
                                 # próbujemy przetworzyć dane z od użytkownika
        form = Form(request.POST) # Tworzymy formularz
        if form.is_valid(): # Jeśli jest poprawny
            # Tu jest kod, który coś robi z formularzem
            return HttpResponseRedirect('/thanks/') # Przekierowanie
    elif request.method == 'GET':
        form = NameForm() # Zapytanie jest GET więc tworzymy formularz
    else:
      return HttpResponse(status=403)
    # Tutaj możemy dość jeśli:
    # * Zapytanie jest GET (wtedy tworzymy nowy formularz)
    # * Zapytanie jest POST i formularz jest **niepoprawny**,
    #   wtedy pojawia się formularz z zaznaczonymi błędami
    return render(request, 'name.html', {'form': form})
```

Note

Django nie lubi duplikacji kodu (zasada DRY), a taki schemat ma 90% widoków w Django, dostarcza się więc Class Based Views oraz Class Based Generic Views, które pozwalają oprogramować widok nie w postaci funkcji, a w postaci **typu**, dodatowo dostarczane są generyczne widoki, które zawierają wzorce jak ten powyżej.
Na razie nie będziemy ich używać (nie wiem czy w ogóle), bo użycie widoków klasowych *na początku* jest trudniejsze niż funkcyjnych. Jednak jeśli kiedyś zajmiecie się Django na poważnie, bardzo polecam ich używanie.

Umieszczanie formularzy w szablonie

Formularz potrafi wyświetlić pola i ew. błędy walidacji. Nie wyświetla natomiast samego tagu `<form>` oraz guzika submit.

```
<form action="/your-name/" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit" />
</form>
```

Instrukcja `{{ form }}` wyświetla zawartość formularza, do tego musicie stworzyć tag `<form>` oraz guzik. Pojawia się również "magiczny" tag `{% csrf_token %}`, jest to tag który implementuje zabezpieczenie przed (bardzo poważnym) atakiem `Cross Site Request Forgery`, który zostanie wyjaśniony pod koniec zajęć.

Pobieranie danych z formularza

Po wykonaniu walidacji (czyli po wywołaniu funkcji `is_valid` albo `full_clean`) dane przesłane przez użytkownika są dostępne jako słownik `cleaned_data`.

Przyjrzyjmy się jeszcze raz fukcji widoku:

```
def view(request):
    if request.method == 'POST':
        form = Form(request.POST)
        if form.is_valid():
            name = form.cleaned_data['your_name']
            # Coś robimy z name
            return HttpResponseRedirect('/thanks/') # Przekierowanie
    elif request.method == 'GET':
        form = NameForm() # Zapytanie jest GET więc tworzymy formularz
        # Tutaj form.cleaned_data jest puste a form.is_valid zwraca False
    else:
      return HttpResponse(status=403)
    return render(request, 'name.html', {'form': form})
```

Obsługa walidacji w formularzach

Formularze Django posiadają funkcjonalność walidacji danych przychodzących od użytkownika. Błędy są automatycznie wyświetlane pod polem (walidacja ma miejsce po stronie serwera, więc błędy pojawią się dopiero po wysłaniu danych).

Podstawowa walidacja

Walidację pól w formularzu można zmieniać podając odpowiednie parametry konstruktora, np. do `IntegerField` można przekazać parametry `max_value` oraz `min_value`.

Dokładne opcje dla każdego pola są opisane w dokumentacji.

Użycie funkcji validatorów

Funkcje validatorów to funkcje które weryfikują, pojedyńcze pola.

```
def validate_even(value):
    if value % 2 != 0:
        raise ValidationError('%s is not an even number' % value)
```

Podpinanie walidatorow:

```
from django.db import models

class MyModel(models.Model):
    even_field = models.IntegerField(validators=[validate_even])
```

Dwie ważne uwagi:

- W przypadku poprawnego pola walidator nie zwraca nic. W przypadku błędu rzuca wyjątek `ValidationError`.
- Walidator otrzymuje wartość już wstępnie przetworzoną przez pole. Walidator na polu Integer field dostanie argument będący już intem.

Sprawdzanie zależności między polami

Powiedzmy, że piszemy widok, w którym (pośród innych pól) są dwa pola: jedno jest chekboxem: `Proszę o przesłanie potwierdzenia`, drugie zawiera adres `e-mail` do przesłania potwierdzenia. Adres `e-mail` nie jest wymagany, chyba, że użytkownik chce dostać informacje o potwierdzeniu.

Takiej relacji nie da się zaprogramować stosując walidatory. W takim przypadku należy nadpisać funkcję `clean` w modelu:

```
from django import forms

class ContactForm(forms.Form):
  notify = forms.BooleanField(label="...", "...")
  e_mail = forms.EmailField(label="...", "...")

  def clean(self):
      cleaned_data = super(ContactForm, self).clean()
      if cleaned_data['notify'] and not cleaned_data['email']:
            raise forms.ValidationError(
                "Please specify e-mail if you want to get notified"
            )
```

> **Note**
> Funkcja clean może modyfikować dane w formularzu, do dango `1.7` musiała ona zwrócić słownik z danymi, teraz może go zwrócić, ale nie jest to wymagane.

Konfiguracja adresów URL w Django

Jest to temat poboczny, ale dość ważny.

Wyrażenia regularne

Wyrażenia regularne, są bardzo prostym narzędziem pozwalającym na przetwarzanie języków regularnych (języki regularne to najbardziej prymitywne języki w hierarchii Chomskyego).

Note

Ogólnorozwojowo polecam Państwu poczytanie o Chomskym i jego poglądach, twierdzi on np., że język jest (a dokładnie umiejętność tworzenia i poznawania języków) jedną z naturalnych funkcji mózgu człowieka --- inaczej mówiąc, język jest organem, podobnie jak wątroba (tylko zapewnia inne funkcje zwiększające szanse przeżycia).

Języki regularne są bardzo prymitywne, następujące języki nie mogą być opisane wyrażeniem regularnym

- Wszystkie nieegzotyczne języki programowania: C, C++, Java, Python
- Języki składu tekstu: Latex, HTML, XML
- Adresu e-mail (bo poprawnym adresem e-mial jest np.: `"foo@bar"+"tag foo bar"@gmail.com`)

Nadają się natomiast do:

- Parsowania wszystikich tych języków w zastoswaniach, w których nie zależy nam na wydajności, ani na 100% poprawności. Można parsować "znany podzbiór" HTML za pomocą wyrażeń regularnych (lepiej użyć parsera HTML wbudowanego w bibliotekę Pythona).
- Parsowania URL na stronie.

Wprowadzenie do wyrażeń regularnych

- Wyrażenie regularne `foo` opisuje ciąg znaków foo.
- Wyrażenie regularne `foo+` opisuje ciąg znaków `foo`, `fooo`, `foooo`` (plus oznacza: "ostatnie wyrażenie powielamy raz lub więcej). Gwiazdka działa jak plus ale dopuszcza zero powtórzeń. Znak zapytania oznacza jedno lub zero powtórzeń.
- Wyrażenie regularne `[abc]` opisuje ciąg znaków `a`, `b` oraz `c`. Nawiasy `[]` oznaczają "grupę znaków", opisują dowolny znak z grupy.
- `[abc]+` opisuje ciągi takie jak: `a`, `aa`, `abc`, `cab` itp...
- `\w` jest predefinowaną grupą oznaczającą litery, `\d` --- cyfry.

Grupy w wyrażeniach regularnych

Wyrażenia regularne mogą zawierać też grupy, np. `/login/(\w+)/?`.

Przykładowo:

```
import re

match = re.match(r'/login/(\w+)/?', '/login/jbzdak')

match.group(1) # Zwraca jbzdak
```

By zgrupować elementy bez tworzenia grupy można skorzystać z `(?: )`, które tworzy grupę która nie ma numeru.

Grupy mogą też mieć nazwę:

```
import re

match = re.match(r'/login/(?P<username>\w+)/?', '/login/jbzdak')

match.group("username") # Zwraca jbzdak
```

Wyrażenia regularne w konfiguracji url w django

Zasady przetwarzania adresów url są proste.

- Django po kolei próbuje dopasować adres URL do wszystkich wzorców
- Jeśli nie uda się to zwraca błąd: `404`
- Jeśli się uda to zwraca wynik wywołania odpowiedniego widoku

Nazwane grupy w konfiguracji URL

Mamy następujący widok

```
def login(request, username):
  pass
  # ....
```

I taką konfigurację URL:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
  url('^/login/(?P<username>\w+)/?$', views.login, name="login")
]
```

Jeśli w takiej konfiguracji użytkownik wejdzie na adres `/login/jb`, to jako parametr `username` widoku zostanie przesłana grupa o nazwie `username` czyli wartość `jb`.

Odwracanie adresów url

Django stosuje zasadę DRY i adres danej strony winien być zdefiniowany dokładnie w jednym miejscu: w konfiguracji url. Jeśli adres strony potrzebny jest w innym miejscu, można go uzyskać za pomocą **odwracania** adresów url.

By dokonać tego w Pythonie należy:

```
from django.core.urlresolvers import reverse

reverse('login', kwargs={"username": "jb"})
```

Pierwszym argumentem tej funkcji jest **nazwa** urla, który odwracamy. Za pomocą kwargs przekazujemy słownik zawierający wartości wszystkich zdefiniowanych grup.

By odwrócić url w szablonie należy użyć tagu `url`:

```
<a href="{% url 'login' username="jb" %}">Login as JB</a>
```

Sesje HTTP

Protokół HTTP nie ma mechanizmu sesji, tj. nie istnieje możliwość by pogrupować zapytania w konwersacje, zasadniczo każde zapytanie jest od siebie niezależne.

HTTP Basic Auth

Funkcjonalność sesji zasadniczo nie jest konieczna do zapewnienia możliwości zalogowania się użytkownika, najprostszym standardem, który umożliwiał logowanie do usług był standard HTTP Basic authentication.

Działanie tego protokołu jest bardzo proste:

- Użykownik próbuje wykonać akcję, do której nie ma uprawnień.
- Serwer odpowiada ze stanem `401` (który oznacza brak autoryzacji), oraz załącza do odpowiedzi nagłowek o treści: `WWW-Authenticate: Basic realm="domena"`.

  Realm oznacza "domenę", do której użytkownik powinien się zautoryzować.
- Użytkownik przesyła kolejne zapytania z nagłówkiem: `Authorization: Basic <<auth>>`, gdzie `<auth>>` zawiera ciąg znaków `użytkownik:hasło` zakodowany za pomocą kodowania Base64.
- Serwer sprawdza hasło i ew. umożliwia danej akcji.

W HTTP-Basic auth nie ma możliwości "wylogowania się" (o ile nie wspiera tego bezpośrednio przeglądarka).

> **Note**
> HTTP Basic Auth **nie zapewnia żadnego bezpieczeństwa**, hasło jest załączane w prostej do odzyskania formie do każdego zapytania.
> Czasem standard ten stosowany jest w sieciach wewnętrznych (zakładamy, że tam sieć jest zaufana). Nie powinno się go stosować bez równoległego szyfrowania całej komunikacji (protokół TLS).

Sesje HTTP

Ciastka

Sesje HTTP mogą być zaimplementowane za pomocą rozszerzenia, czyli `ciastek`. Ciastka to specjalne nagłowki HTTP, które wysyła serwer i które oznaczają: "Droga

przeglądarko tutaj masz ciąg znaków który powinnaś odsyłać do każdego kolejnego zapytania, które dodatkowo spełnia pełne warunki".

By poprosić przeglądarkę o ustawienie ciastka serwer wysyła nagłowek o treści:

```
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

Przeglądarka powinna odsyłać nagłowek o treści:

```
Cookie: sessionToken=abc123
```

Aż do dnia: Wed, 09 Jun 2021 10:18:14 GMT.

Jedno ciastko jest odwzorowaniem `klucz=wartość`, tj. przeglądarka, która otrzyma:

```
Set-Cookie: sessionToken=abc123; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

zapamiętuje że kluczowi `sessionToken`, przypisano wartość `abc123`.

Ograniczenia ciastek

Serwer może określić:

- Maksymalny wiek ciastka
- Domenę, dla której ciastko jest ustawione (Domena `foo.com` może ustawić ciastko dla domeny `*.foo.com`, nie może dla `com` oraz `bar.com`).
- Ścieżkę dla której ciastko jest ustawione (jeśli ciastko jest ustawione na ścieżkę: `/path` to tylko zapytania na ścieżki zaczynające się od `/path` będą zawierać dane ciastko.
- To, że ciastko może być wysyłane tylko dla połączeń HTTPS.
- To, że ciastko może nie jest widoczne dla kodu Javascript.

Implementacja sesji za pomocą ciastek

Sesja trzymana po stronie serwera

> **Note**
> W 99% przypadków prawidłowym rozwiązaniem jest trzymanie sesji po stronie serwera. Proszę używać tej metody, o ile nie macie ważnego powodu.

Użytkownik wchodzi na stronę logowania i wprowadza poprawne dane logowania. Serwer generuje dla niego **losowy** "identyfikator sesji". Następnie zapisuje (w bazie danych, lub na dysku, lub w inny sposób) informacje o tym, że z danym identyfikatorem sesji posługuje się użytkownik o danej nazwie.

Po wykonaniu tych czynności serwer odsyła identyfikator sesji w ciasteczku do użytkownika.

Autoryzacja użykownika następuje za pomocą identyfikatora sesji.

Note

Identyfikator sesji stanowi informację za pomocą, której można *podszyć się za danego użytkownika*. Kiedy jestem w stanie **zgadnąć** czyjś identyfikator sesji mogę wykonywać zapytania tak jakbym był tą osobą.
Numery sesji **nie mogą być przewidywalne**, w szczególności **nie mogą one być przydzielane po kolei**, nie mogą one też być generowane za pomocą **generatorów pseudolosowych ogólnego przeznaczenia**.
Muszą być generowane za pomocą **kryptograficznych generatorów pseudolosowych**.

Sesja trzymana po stronie klienta

Czasem można trzymać sesję po stronie klienta. Wtedy działa to tak:

Użytkownik wchodzi na stronę logowania i wprowadza poprawne dane logowania. Serwer generuje dla niego ciastko np. o treści `session=user:jb,auth:bewqe23321`, i wysyła takie ciastko.

Autoryzacja użykownika do dalszych zapytań polega na odczytaniu ciastka i sprawdzeniu nazwy użytkownika.

Oczywiście użytkownik może podmienić w ciastku nazwę użytkownika na inną, np. `session=user:admin,auth:bewqe23321`, a serwer powinienen wtedy odrzucić to ciastko jako niepoprawne. By tak się stało ciastko musi być **kryptograficznie podpisane**.

Zalety sesji trzymanej po stronie klienta:

- Jeśli sesja zawiera mało danych, to by sprawdzić dane z sesji nie trzeba wykonywać zapytania bazodanowego do pobrania danych z sesi, co może czasem znacznie przyśpieszyć działanie niektórych aplikacji.

Wady sesji trzymanej po stronie użytkownika:

- Ciastko musi być kryptograficznie podpisane. Bardzo łatwo jest stworzyć system kryptograficzny który **nie działa**. Np. StackOverflow swojego czasu miał dziurę pozwalającą każdemu zalogować się jako admin
- Ciastko jest przesyłane z każdym zapytaniem. Jeśli jest ono duże może powodować to spowolnienie działania strony dla innych użytkowników.
- Nie ma możliwości **wygaśnięcia** sesji przed czasem. Ciastko to musi jeszcze zawierać (w części, która jest podpisana!) datę wygaśnięcia, niestety nie ma możliwości spowodowania, by taka sesja szybciej wygasła.

Implementacja sesji trzymanej w bazie danych

Serializacja danych

Serializacja to proces odwracalnej zamiany drzewa obiektów w ciąg bajtów, deserializacja to proces zamiany ciągu bajtów na drzewo obiektów.

Jednym z narzędzi do serializacji danych w Pythonie jest moduł `pickle`, by zapisać dowolny obiekt Pythona należy:

```
import pickle
serialized = pickle.dumps({'a':1, 'b': 'foo'})
```

By go deserializować:

```
object = pickle.loads(serialized)
```

Note

Pickle jest protokołem, który pozwala na serializowanie dowolnych obiektów Pythona i jest rozsądnie szybki. Jednak **nie wolno odczytywać danych pochodzących z niezaufanego źródła**. Odpowiednio stworzony strumień danych pickle może podczas deserializacji wykonać **dowolne polecenia z uprawnieniami danego użytkownika**, w szczególności **może wywołać** `rm -rf`.
Pickle nie wspiera również szyfrowania danych! Dane są bezpośrednio czytelne w zapisanym strumieniu:

```
print(pickle.dumps({'a':1, 'b': '       ********** TU JESTEM **********
```

wyświetli:

```
b'\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02X.\x
```

Model sesji

Sesja jest implementowana jako model podobny do:

```
class Session(models.Model):

  session_id = models.CharField(max_length=32, unique=True, null=False)
  user = models.ForeignKey('auth.User', null=True)
  data = models.BinaryField(null=False)
  expiry = models.DateTimeField(null=False)
```

Algortmy powiązane z sesją

Nadanie sesji:

1. Jeśli użytkownik nie wysłał nam identyfikatora sesji
2. Tworzymy w nowy identyfikator sesji
3. Zapisujemy w bazie danych nową sesję z ustawionym czasem wygaśnięcia
4. Odsyłamy w ciastku identyfikator sesji.

Zalogowanie użytkownika

1. Sprawdzamy czy użytkownik ma już istniejącą sesję, jeśli nie tworzymy ją.
2. Dodajemy do sesji informację o tym, że użytkownik się zalogował

Wylogowanie użytkownika

1. Usuwamy daną sesję z bazy danych.

Pobranie aktualnej sesji

1. Jeśli użytkownik nie wysłał identyfikatora sesji tworzymy nową.
2. Pobieramy sesję z bazy danych. Jeśli nie istnieje tworzymy nową.
3. Sprawdzamy czy sesja nie wyglasła. Jeśli tak tworzymy nową.

> **Note**
> Proszę się zastanowić czemu czas wygaśnięcia sesji jest zapisywany również po stronie serwera.

Podstawy kryptografii

Przechowywanie haseł

Haseł nigdy nie przechowujemy w postaci jawnej w bazie danych. Zamiast zapsania hasła przechowujemy takie informacje:

- nazwę algorytmu funkcji skrótu
- znany (ale różny dla każdego hasła) ciąg znaków zwany solą
- wynik działania funkcji skrótu na ciągu znaków `sól:hasło`.

By sprawdzić czy użytkownik podał poprawne hasło należy: wykonać funkcję skrótu na ciągu znaków `sól:hasło dane przez użytkownika` i sprawdzić czy równa się tej zapisanej w bazie danych.

> **Note**
> Funkcja skrótu to funkcja, która przekształca ciąg N bitów w ciąg M bitów, gdzie M jest stałe a N dowolne. Ma dodatkowo nastepujące cechy:
> - Statystycznie zmiana jednego bitu w ciągu wejściowym powoduje zmianę M/2 bajtów w ciągu wyjściowum.
> - Mając ciąg wyjściowy nie da się odzyskać wejściowego, szybciej niż próbując wykonując tą funkcję dla wszystkich możliwych ciągów wejściowe.

Losowanie liczb

Funkcje:

- `random.random()` z Pythona
- `rand` z biblioteki standardowej `C`
- `java.Util.Random`
- Generator `Mersenne Twister` (często używany do losowania liczb w Fizyce Wysokich Energii)

Absolutnie nie nadają się do generowania liczb losowych do krytograficznego zastosowania.

W Pythonie poleca się albo: `os.urandom` albo `random.SystemRandom`, które korzystają z kryptograficznego generatora wbudowanego w system operacyjny.

> **Note**
>
> Do poważnych zastosowań lepiej jest użyć nie używać generatora wbudowanego w system, ponieważ może on posiadać problem z brakiem dostatecznej losowości na starcie systemu.
> W skrócie: dwie takie same maszyny wirtualne mają też bardzo podobne ziarna losowe dla generatora wbudowanego w system, co powoduje, że da się przewidzieć wygenerowane tak liczby losowe.

Same Origin Policy

Single Origin Policy to podstawa modelu bezpieczeństwa współczesnych przeglądarek internetowych.

Oznacza ona, że serwer skojarzony z domeną A widzi ciastka przesłane przez serwer skojrzaony z domeną B. Tylko wtedy gdy B==A lub B jest subdomeną A.

Prostymi słowy oznacza to, ciastko sesji z Waszego banku jest przesyłane tylko na strony znajdujące się w domenie banku.

To co tutaj przedstawiam jest bardzo uproszczoną wersją SOP, tutaj więcej informacji.

Cross Site Request Forgery

Podatność Cross Site Request Forgery

Powiedzmy, by wykonać przelew w Waszym banku należy wyjkonać zapytanie POST na stronę `/przelew`, które zawiera:

1. Numer konta docelowego
2. Kwotę

Umieszczam teraz na mojej stronie(!) następujący formularz:

```
<input type="text" name="treść">

<form action="wasz.bank.com/przelew" method="Post">
  <input type="hidden" name="account-from" value="Mój numer konta">
  <input type="hidden" name="amount" value="100zł">
  <submit> Wyślij Komentarz </submit>
</form>
```

Użytkownik widzi pole tekstowe oraz guzik z napisem "Wyślij komentarz".

Użytkownik dodatkowo jest w innym oknie przeglądarki zalogowany do swojego banku, wpisuje komentarz i klika wyślij.

Przeglądarka wysyła zapytanie POST na serwer `wasz.bank.com`, załącza do niego również wszystkie ciastka z tej domeny (co jest zgodne z SOP). W wyniku tego dokonywany jest przelew.

Mechanizm ochrony przed CSRF

By zabezpieczyć się przed CSRF należy:

1. Do każdego formularza POST dodać ukryte pole z losową wartością.
2. Przy przetwarzaniu formularza POST sprawdzać wartośc w tym polu.

W Django takie pole dodaje tag `{% csrf_token %}`, a sprawdzanie jest automatyczne.