



Technologia i rozwiązania

Aplikacje internetowe z Django

Najlepsze receptury

Wykorzystaj potencjał Django!



Aidas Bendoraitis

[PACKT] open source*
PUBLISHING community experience distilled

Tytuł oryginału: Web Development with Django Cookbook

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-1107-7

Copyright © Packt Publishing 2014.

First published in the English language under the title 'Web Development with Django Cookbook - 9781783286898'

Polish edition copyright © 2015 by Helion S.A.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/apindj.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/apindj>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

O autorze	7
O recenzentach	9
Wstęp	11
Rozdział 1. Wprowadzenie do systemu Django 1.6	15
Wprowadzenie	16
Praca w środowisku wirtualnym	16
Tworzenie struktury plików projektu	18
Zarządzanie zależnościami projektu za pomocą narzędzia pip	20
Dołączanie zewnętrznych zależności do projektu	21
Definiowanie ścieżek względnych w ustawieniach	23
Dynamiczne ustawianie wartości zmiennej STATIC_URL dla użytkowników systemu Subversion	25
Dynamiczne ustawianie wartości zmiennej STATIC_URL dla użytkowników systemu Git	26
Tworzenie i dołączanie ustawień lokalnych	28
Ustawianie domyślnego kodowania UTF-8 w konfiguracji MySQL	29
Ustawianie własności ignore systemu Subversion	30
Tworzenie pliku z informacjami o ignorowanych zasobach w systemie Git	32
Usuwanie skompilowanych plików Pythona	32
Ustalanie kolejności importowania plików Pythona	33
Definiowanie możliwych do zmienienia ustawień aplikacji	35
Rozdział 2. Struktury bazy danych	37
Wprowadzenie	37
Stosowanie domieszek do modeli	38
Tworzenie domieszek do modeli przy użyciu metod związanych z adresami URL	39
Tworzenie domieszek do modeli do tworzenia i modyfikowania dat	42

Tworzenie domieszek do modeli do obsługi metaznaczników	43
Tworzenie domieszek do modeli do obsługi relacji generycznych	46
Obsługa pól wielojęzycznych	50
Sposób użycia migracji South	55
Zmianie klucza obcego na pole wiele do wielu przy użyciu narzędzia South	58
Rozdział 3. Formularze i widoki	61
Wprowadzenie	61
Przekazywanie obiektu HttpRequest do formularza	61
Sposób użycia metody save formularza	64
Wysyłanie obrazów na serwer	66
Tworzenie układu formularza przy użyciu aplikacji django-crispy-forms	70
Filtrowanie list obiektów	74
Zarządzanie listami stronicowanymi	80
Komponowanie widoków na bazie klas	83
Rozdział 4. Szablony i JavaScript	87
Wprowadzenie	87
Konfiguracja pliku szablonowego base.html	88
Dołączanie ustawień JavaScript	90
Wykorzystywanie atrybutów danych HTML5	93
Otwieranie szczegółów obiektu w wyskakującym okienku	97
Implementacja ciągłego przewijania	100
Implementacja widżetu polubień	102
Wysyłanie obrazów przy użyciu technologii Ajax	108
Rozdział 5. Tworzenie własnych filtrów i znaczników szablonowych	117
Wprowadzenie	117
Stosowanie konwencjonalnych rozwiązań przy tworzeniu filtrów i znaczników szablonowych	118
Tworzenie filtra szablonowego do pokazywania, ile dni upłynęło	119
Tworzenie filtra szablonowego do pobierania pierwszego obiektu mediów	121
Tworzenie filtra szablonowego dostosowującego adresy URL do potrzeb użytkownika	123
Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje	124
Tworzenie znacznika szablonowego do ładowania zestawu obiektów do szablonu	127
Tworzenie znacznika szablonowego do przetwarzania treści jako szablonu	131
Tworzenie znacznika szablonowego do modyfikowania parametrów zapytań	133
Rozdział 6. Modelowanie panelu administracyjnego	137
Wprowadzenie	137
Dostosowywanie kolumn na stronie listy zmian	138
Tworzenie czynności administracyjnych	141
Tworzenie filtrów listy zmian	144
Zamienianie ustawień administracyjnych na zewnętrzne aplikacje	147
Wstawianie map do formularza zmian	149

Rozdział 7. Django CMS	157
Wprowadzenie	157
Tworzenie szablonów dla systemu Django CMS	158
Konfigurowanie menu stron	161
Konwertowanie aplikacji na aplikację CMS	164
Dodawanie własnej nawigacji	166
Pisanie własnej wtyczki	168
Dodawanie nowych pól do strony CMS-a	173
Rozdział 8. Struktury hierarchiczne	179
Wprowadzenie	179
Tworzenie kategorii hierarchicznych	181
Tworzenie interfejsu do administracji kategoriami przy użyciu aplikacji django-mptt-admin	184
Tworzenie interfejsu do administracji kategoriami przy użyciu aplikacji django-mptt-tree-editor	186
Generowanie kategorii w szablonie	188
Wybieranie kategorii w formularzach przy użyciu pola pojedynczego wyboru	190
Wybieranie wielu kategorii w formularzach przy użyciu listy pól wyboru	191
Rozdział 9. Importowanie i eksportowanie danych	197
Wprowadzenie	197
Importowanie danych z lokalnego pliku CSV	197
Importowanie danych z lokalnego pliku Excela	199
Importowanie danych z zewnętrznego pliku w formacie JSON	201
Importowanie danych z zewnętrznego pliku XML	206
Tworzenie kanałów RSS z możliwością filtrowania	209
Dostarczanie danych do użytkowników zewnętrznych za pomocą usługi Tastypie	214
Rozdział 10. Sztuczki i bajery	217
Wprowadzenie	217
Posługiwanie się powłoką Django	218
Tworzenie przyjaznych adresów	220
Zastępowanie modelu administracji	222
Włączanie i wyłączanie paska narzędzi debugowania	226
Sposób użycia programu pośredniczącego ThreadLocalMiddleware	229
Buforowanie wartości metody	231
Wysyłanie raportów o błędach na adres e-mail	232
Wdrażanie aplikacji na serwerze Apache przy użyciu modułu mod_wsgi	234
Tworzenie i używanie skryptu wdrażania Fabric	241
Skorowidz	251

Tworzenie własnych filtrów i znaczników szablonowych

W rozdziale:

- Stosowanie konwencjonalnych rozwiązań przy tworzeniu filtrów i znaczników szablonowych
- Tworzenie filtru szablonowego do pokazywania, ile dni upłynęło
- Tworzenie filtru szablonowego do pobierania pierwszego obiektu mediów
- Tworzenie filtru szablonowego dostosowującego adresy URL do potrzeb użytkownika
- Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje
- Tworzenie znacznika szablonowego do ładowania zestawu obiektów do szablonu
- Tworzenie znacznika szablonowego do przetwarzania treści jako szablonu
- Tworzenie znacznika szablonowego do modyfikowania parametrów zapytań

Wprowadzenie

Jak wiadomo, Django zawiera dość rozbudowany system szablonowy obsługujący między innymi dziedziczenie szablonów, filtry do zmieniania reprezentacji wartości oraz znaczniki obsługujące logikę prezentacyjną. Ponadto Django umożliwia dodawanie filtrów i znaczników szablonowych we własnych aplikacjach. Niestandardowe filtry i znaczniki powinny znajdować się

w pliku biblioteki znaczników szablonowych w pakiecie Pythona `templatetags` w aplikacji. Bibliotekę taką można załadować w wybranym szablonie za pomocą znacznika szablonowego `{% load %}`. W tym rozdziale utworzymy kilka przydatnych filtrów i znaczników dających edytorom szablonów dodatkowe możliwości działania.

Stosowanie konwencjonalnych rozwiązań przy tworzeniu filtrów i znaczników szablonowych

Jeśli programista nie zastosuje spójnych reguł działania, w jego niestandardowych filtrach i znacznikach szablonowych szybko może powstać straszny bałagan. Składniki te powinny być jak najbardziej pomocne dla edytorów szablonów. Powinny być zarówno wygodne w użyciu, jak i elastyczne. W tej recepturze przedstawiam pewne konwencje, które można stosować przy rozszerzaniu systemu szablonowego Django.

Jak to zrobić

Rozszerzając system szablonów Django, stosuj się do następujących zaleceń:

1. Nie twórz i nie używaj niestandardowych filtrów ani znaczników szablonowych, jeśli logika strony lepiej pasuje do widoku, procesora kontekstu lub metod modelu. Jeżeli strona jest specyficzna dla kontekstu, na przykład zawiera listę obiektów albo widok szczegółów obiektów, załaduj obiekty w widoku. Jeśli chcesz wyświetlić pewną treść na każdej stronie, utwórz procesor kontekstu. Jeżeli trzeba pobrać niektóre własności obiektu niezwiązane z kontekstem szablonu, lepiej jest wykorzystać niestandardowe metody modelu zamiast filtrów szablonowych.
2. Do nazwy biblioteki znaczników szablonowych dodawaj przyrostek `_tags`. Dzięki nadaniu bibliotece innej nazwy niż aplikacji unikniesz problemów z niejednoznacznością przy importowaniu.
3. W nowo utworzonej bibliotece oddziel filtry od znaczników, na przykład za pomocą komentarzy, jak poniżej:

```
# -*- coding: UTF-8 -*-
from django import template
register = template.Library()

### FILTRY ###
# ...kod źródłowy filtrów...

### ZNACZNIKI ###
# ...kod źródłowy znaczników...
```

4. Twórz takie znaczniki szablonowe, które jest łatwo zapamiętać; stosuj do tego poniższe konstrukcje:
 - `for [nazwa_aplikacji.nazwa_modelu]` — konstrukcja umożliwiająca wykorzystanie określonego modelu.
 - `using [nazwa_szablonu]` — konstrukcja umożliwiająca wykorzystanie szablonu dla wyniku znacznika szablonowego.
 - `limit [liczba]` — konstrukcja umożliwiająca ograniczenie wyników do określonej liczby.
 - `as [zmienna_kontekstowa]` — konstrukcja umożliwiająca zapisanie wyników w zmiennej kontekstowej, której później można używać wielokrotnie.
5. Staraj się nie używać w znacznikach szablonowych zbyt wielu wartości zdefiniowanych pozycyjnie, chyba że ich przeznaczenie jest oczywiste. W przeciwnym razie programiści szablonów mogą mieć z nimi problemy.
6. Twórz jak najwięcej rozwiązywalnych argumentów. Łańcuchy bez cudzysłowów powinno się traktować jak zmienne kontekstowe, które trzeba rozwiązać, lub jako krótkie słowa przypominające o strukturze składników znacznika szablonowego.

Zobacz również

- Receptura „Tworzenie filtru szablonowego do pokazywania, ile dni upłynęło”
- Receptura „Tworzenie filtru szablonowego do pobierania pierwszego obiektu mediów”
- Receptura „Tworzenie filtru szablonowego dostosowującego adresy URL do potrzeb użytkownika”
- Receptura „Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje”
- Receptura „Tworzenie znacznika szablonowego do ładowania zestawu obiektów do szablonu”
- Receptura „Tworzenie znacznika szablonowego do przetwarzania treści jako szablonu”
- Receptura „Tworzenie znacznika szablonowego do modyfikowania parametrów zapytań”

Tworzenie filtru szablonowego do pokazywania, ile dni upłynęło

Nie każdy rejestruje daty, a jeśli chodzi o tworzenie i modyfikowanie najświeższych informacji, to w wielu przypadkach wygodniejsze jest rejestrowanie różnic w czasie, na przykład wpis na blogu został opublikowany trzy dni temu albo artykuł opublikowano dziś i użytkownik logował się po raz ostatni wczoraj. W tej recepturze pokazuję, jak utworzyć filtr szablonowy o nazwie `days_since`, konwertujący daty na czytelne różnice czasu.

Przygotowanie

Jeśli jeszcze tego nie zrobiłeś, utwórz aplikację `utils` i dodaj ją do sekcji `INSTALLED_APPS`. Następnie w tej aplikacji utwórz pakiet Pythona o nazwie `templatetags` (pakiety Pythona to katalogi zawierające pusty plik `__init__.py`).

Jak to zrobić

Utwórz plik o nazwie `utility_tags.py` z następującą zawartością:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from datetime import datetime
from django import template
from django.utils.translation import ugettext_lazy as _
from django.utils.timezone import now as tz_now
register = template.Library()

### FILTRY ###

@register.filter
def days_since(value):
    """ Zwraca liczbę dni między aktualną datą a podaną wartością. """

    today = tz_now().date()
    if isinstance(value, datetime.datetime):
        value = value.date()
    diff = today - value
    if diff.days > 1:
        return _("%s dni temu") % diff.days
    elif diff.days == 1:
        return _("wczoraj")
    elif diff.days == 0:
        return _("dzisiaj")
    else:
        # Podano przyszłą datę, zwraca sformatowaną datę.
        return value.strftime("%B %d, %Y")
```

Jak to działa

Jeśli zastosuje się ten filtr w takim szablonie jak poniższy, to wyrenderuje on napis typu wczoraj albo 5 dni temu:

```
{% load utility_tags %}
{{ object.created|days_since }}
```

Filtr ten można stosować do wartości typów `date` i `datetime`.

Każda biblioteka znaczników szablonowych ma rejestr, w którym zebrane są filtry i znaczniki. Filtry Django są funkcjami zarejestrowanymi przez dekorator `register.filter`. Domyślnie filtrowi w systemie szablonów nadawana jest taka sama nazwa jak nazwa funkcji lub innego wywołывalnego obiektu. Jednak w razie potrzeby można ustawić inną nazwę, przekazując opcję `name` do dekoratora:

```
@register.filter(name="humanized_days_since")
def days_since(value):
    ...
```

Sposób działania filtra nie wymaga obszernych wyjaśnień. Najpierw odczytuje aktualną datę. Jeśli podana wartość jest typu `datetime`, to zostaje pobrana data. Następnie obliczana jest różnica między bieżącą a pobraną datą. Zwrot wartości zależy od liczby dni różnicy.

To nie wszystko

Ten filtr można łatwo rozszerzyć tak, aby pokazywał też różnicę w minutach i godzinach, na przykład: właśnie teraz, 7 minut temu, 3 godziny temu itd. Trzeba tylko posługiwać się wartościami typu `datetime` zamiast `date`.

Zobacz również

- Receptura „Tworzenie filtra szablonowego do pobierania pierwszego obiektu mediów”
- Receptura „Tworzenie filtra szablonowego dostosowującego adresy URL do potrzeb użytkownika”

Tworzenie filtra szablonowego do pobierania pierwszego obiektu mediów

Wyobraź sobie, że tworzysz stronę przeglądu zawartości bloga i dla każdego wpisu chcesz pokazać, jakie obrazy, klipy muzyczne lub filmy wideo znajdują się w jego treści. Aby to zrobić, musisz pobrać elementy `<img>`, `<object>` i `<embed>` z kodu HTML. W tej recepturze pokazuję, jak to zrobić przy użyciu wyrażeń regularnych w filtrze `get_first_media`.

Przygotowanie

Najpierw utwórz aplikację `utils`, którą trzeba dodać do sekcji `INSTALLED_APPS` w ustawieniach, a w aplikacji tej utwórz pakiet `templatetags`.

Jak to zrobić

Dodaj do pliku `utility_tags.py` poniższy kod:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
import re
from django import template
from django.utils.safestring import mark_safe
register = template.Library()

### FILTRY ###

media_file_regex = re.compile(r'<object .+?</object>|<(img|embed) [^>]+>')

@register.filter
def get_first_media(content):
    """Zwraca pierwszy obraz lub plik Flash z treści HTML. """
    m = media_file_regex.search(content)
    media_tag = ""
    if m:
        media_tag = m.group()
    return mark_safe(media_tag)
```

Jak to działa

Jeśli znajdująca się w bazie danych treść HTML jest poprawna, to poniższy kod umieszczony w szablonie spowoduje pobranie elementów `<object>`, `<img>` i `<embed>` z treści pola obiektu lub pobranie pustego łańcucha, jeśli nie zostaną znalezione żadne media:

```
{% load utility_tags %}
{{ object.content|get_first_media }}
```

Najpierw definiujemy skompilowane wyrażenie regularne jako `media_file_regex`, następnie w filtrze szukamy fragmentów tekstu pasujących do tego wzorca. Domyślnie w wyniku znaki `<`, `>` i `&` są zastępowane encjami `<`, `>` oraz `&`. Jednak zastosowaliśmy funkcję `mark_safe` oznaczającą wynik jako bezpieczny kod HTML, który można wstawić do szablonu bez stosowania symboli zastępczych.

To nie wszystko

Ten filtr można łatwo rozszerzyć tak, aby rozpoznawał też elementy `<iframe>` (które od pewnego czasu są wykorzystywane przez serwisy Vimeo i YouTube) oraz nowe elementy HTML5 `<audio>` i `<video>`. Wystarczy tylko zmienić wyrażenie regularne w następujący sposób:

```
media_file_regex = re.compile(r"<iframe .+?</iframe>|"
    r"<audio .+?</audio>|<video .+?</video>|"
    r"<object .+?</object>|<(img|embed) [^>]+>")
```

Zobacz również

- Receptura „Tworzenie filtru szablonowego do pokazywania, ile dni upłynęło”
- Receptura „Tworzenie filtru szablonowego dostosowującego adresy URL do potrzeb użytkownika”

Tworzenie filtru szablonowego dostosowującego adresy URL do potrzeb użytkownika

Typowy użytkownik internetu w przeglądarce wpisuje adresy URL bez przedrostka określającego protokół i bez końcowego ukośnika. W tej recepturze utworzymy filtr `humanize_url` prezentujący adresy w skróconym formacie, tzn. skracający długie adresy podobnie, jak robi to portal Twitter z długimi odnośnikami we wpisach.

Przygotowanie

Podobnie jak w poprzedniej recepturze musisz mieć aplikację `utils` dodaną do sekcji `INSTALLED_APPS` w ustawieniach, a w niej pakiet `templatetags`.

Jak to zrobić

W sekcji `FILTRY` biblioteki szablonowej `utility_tags.py` w aplikacji `utils` dodamy filtr o nazwie `humanize_url`, który od razu zarejestrujemy:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
import re
from django import template
register = template.Library()

### FILTRY ###

@register.filter
def humanize_url(url, letter_count):
    """Zwraca skrócony czytelny adres URL. """
    letter_count = int(letter_count)
    re_start = re.compile(r'^https?://')
    re_end = re.compile(r'/$')
    url = re_end.sub("", re_start.sub("", url))
    if len(url) > letter_count:
        url = u"%s..." % url[:letter_count - 1]
    return url
```

Jak to działa

Filtru `humanize_url` można używać we wszystkich szablonach w następujący sposób:

```
{% load utility_tags %}
<a href="{{ object.website }}" target="_blank">
    {{ object.website|humanize_url:30 }}
</a>
```

Filtr ten za pomocą wyrażenia regularnego usuwa nazwę protokołu z początku i ukośnik z końca, a następnie skraca adres URL do określonej liczby znaków, dodając na końcu wielokropek, jeśli adres jest dłuższy.

Zobacz również

- Receptura „Tworzenie filtru szablonowego do pokazywania, ile dni upłynęło”
- Receptura „Tworzenie filtru szablonowego do pobierania pierwszego obiektu mediów”
- Receptura „Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje”

Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje

W systemie Django dostępny jest znacznik szablonowy `{% include %}` renderujący i dołączający szablon. Jednak czasami, jeśli szablon, który ma zostać dołączony, nie istnieje, zostaje zgłoszony błąd. W tej recepturze pokazuję, jak utworzyć znacznik szablonowy `{% try_to_include %}`, dołączający do szablonu inny szablon, ale nie zgłaszający żadnych błędów, jeśli nie znajdzie składnika do dołączenia.

Przygotowanie

Pracę rozpoczniemy po raz kolejny od aplikacji `utils`, która powinna być już zainstalowana i gotowa do dodawania niestandardowych znaczników szablonowych.

Jak to zrobić

Znaczniki szablonowe składają się z dwóch części: funkcji przetwarzającej argumenty oraz klasy węzłowej zawierającej logikę działania znacznika i zwracania przez niego treści. Wykonaj następujące czynności:

1. Najpierw napiszemy funkcję przetwarzającą argumenty znacznika szablonowego:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from django import template
from django.template.loader import get_template
register = template.Library()

### ZNACZNIKI ###

@register.tag
def try_to_include(parser, token):
    """ Sposób użycia: {% try_to_include "szablon.html" %}
    Jeśli szablon nie istnieje, nastąpi cicha awaria. Jeśli szablon istnieje,
    zostanie wygenerowany przy użyciu bieżącego kontekstu. """
    try:
        tag_name, template_name = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError, \
            "Znacznik %r wymaga pojedynczego argumentu." % token.contents.split()[0]

    return IncludeNode(template_name)
```

2. W następnej kolejności potrzebujemy klasy węzłowej, która powinna znaleźć się w tym samym pliku:

```
class IncludeNode(template.Node):
    def __init__(self, template_name):
        self.template_name = template_name

    def render(self, context):
        try:
            # Ładowanie i renderowanie szablonu.
            template_name = template.resolve_variable(self.template_name, context)
            included_template = get_template(template_name).render(context)
        except template.TemplateDoesNotExist:
            included_template = ""
        return included_template
```

Jak to działa

Znacznik szablonowy `{% try_to_include %}` pobiera jeden argument określający nazwę szablonu — `template_name`. Zatem w funkcji `try_to_include` próbujemy przypisać podzieloną treść argumentu `token` tylko do zmiennych `tag_name` (którą jest `try_to_include`) i `template_name`. Jeśli to się nie uda, zostaje zgłoszony błąd składni szablonu. Funkcja zwraca obiekt `IncludeNode`, któremu nadaje pole `template_name` do użytku w późniejszym czasie.

W metodzie `render` obiektu `IncludeNode` rozwiązujemy zmienną `template_name`. Jeżeli do znacznika szablonowego została przekazana zmienna kontekstowa, to zostanie ona użyta jako wartość zmiennej `template_name`. Jeśli do znacznika szablonowego został przekazany łańcuch

w cudzysłowie, to jako wartość zmiennej `template_name` zostanie wykorzystany tekst z tego cudzysłowu.

Na koniec próbujemy załadować i wyrenderować szablon przy użyciu bieżącego kontekstu szablonowego. Jeśli się to nie uda, zwracamy pusty łańcuch.

Znacznik ten może być przydatny w przynajmniej dwóch sytuacjach:

- przy dołączaniu szablonu, którego ścieżka jest zdefiniowana w modelu, na przykład:


```
{% load utility_tags %}
{% try_to_include object.template_path %}
```
- przy dołączaniu szablonu, którego ścieżka jest zdefiniowana przy użyciu znacznika szablonowego `{% with %}` gdzieś wysoko w zakresie zmiennej kontekstowej szablonu. Jest to szczególnie przydatne, gdy trzeba utworzyć własne układy dla wtyczek w tekście zastępczym szablonu w Django CMS:

```
#templates/cms/start_page.html
{% with editorial_content_template_path=
"cms/plugins/editorial_content/start_page.html" %}
    {% placeholder "main_content" %}
{% endwith %}

#templates/cms/plugins/editorial_content.html
{% load utility_tags %}

{% if editorial_content_template_path %}
    {% try_to_include editorial_content_template_path %}
{% else %}
    <div>
        <!-- Domyślna prezentacja wtyczki
              dotyczącej treści redakcyjnej. -->
    </div>
{% endif %}
```

To nie wszystko

Za pomocą znaczników szablonowych `{% try_to_include %}` i `{% include %}` można dołączać szablony rozszerzające inne szablony. Jest to szczególnie przydatne w dużych portalach zawierających różnego rodzaju listy, w których skomplikowane elementy, takie jak widżety, mają taką samą strukturę, ale korzystają z różnych źródeł danych.

Na przykład w szablonie listy artystów można dołączyć szablon elementu reprezentującego jednego artystę w następujący sposób:

```
{% load utility_tags %}
{% for object in object_list %}
    {% try_to_include "artists/includes/artist_item.html" %}
{% endfor %}
```

Szablon ten rozszerzałby szablon bazowy elementów w następujący sposób:

```
{# templates/artists/includes/artist_item.html #}
{% extends "utils/includes/item_base.html" %}

{% block item_title %}
    {{ object.first_name }} {{ object.last_name }}
{% endblock %}
```

W szablonie bazowym znajdowałyby się definicje kodu HTML reprezentującego elementy i dołączany byłby widżet Like:

```
{# templates/utils/includes/item_base.html #}
{% load likes_tags %}

<h3>{% block item_title %}{% endblock %}</h3>
{% if request.user.is_authenticated %}
    {% like_widget for object %}
{% endif %}
```

Zobacz również

- Receptura „Tworzenie szablonów dla systemu Django CMS” z rozdziału 7. „Django CMS”
- Receptura „Pisanie własnej wtyczki” z rozdziału 7. „Django CMS”
- Receptura „Implementacja widżetu polubień” z rozdziału 4. „Szablony i JavaScript”
- Receptura „Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje”
- Receptura „Tworzenie znacznika szablonowego do ładowania zestawu obiektów do szablonu”
- Receptura „Tworzenie znacznika szablonowego do modyfikowania parametrów zapytań”

Tworzenie znacznika szablonowego do ładowania zestawu obiektów do szablonu

Treść przeznaczona do wyświetlenia na stronie internetowej najczęściej musi być zdefiniowana w widoku. Jeśli powinna ona być widoczna na każdej stronie, to najlepszym rozwiązaniem jest utworzenie procesora kontekstu. Czasami chcemy też pokazać dodatkową treść, taką jak najświeższe wiadomości albo losowy cytat na pewnych wybranych stronach, na przykład stronie startowej lub stronie szczegółowych informacji o obiekcie. W takim przypadku można załadować potrzebną treść za pomocą znacznika szablonowego `{% get_objects %}`, który zaimplementujemy w tej recepturze.

Przygotowanie

Po raz kolejny zaczniemy pracę od aplikacji `utils`, która powinna być już zainstalowana i gotowa do definiowania znaczników szablonowych.

Jak to zrobić

Znaczniki szablonowe składają się z funkcji przetwarzającej argumenty przekazane do znacznika i klasy węzłowej renderującej wynik lub modyfikującej kontekst szablonu. Wykonaj zatem następujące czynności:

1. Utwórz następującą funkcję przetwarzającą argumenty znacznika szablonowego:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from django.db import models
from django import template
register = template.Library()

### ZNACZNIKI ###

@register.tag
def get_objects(parser, token):
    """
    Pobiera zbiór obiektów modelu określonego przez aplikację i nazwy modeli.
    Sposób użycia:
    {% get_objects [<manager>.]<method> from <app_name>.<model_name>
    ↳[limit <amount>] as <var_name> %}

    Przykład:
    {% get_objects latest_published from people.Person limit 3 as people %}
    {% get_objects site_objects.all from articles.Article limit 3 as articles %}
    {% get_objects site_objects.all from articles.Article as articles %}
    """
    amount = None
    try:
        tag_name, manager_method, str_from, appmodel, str_limit, amount, str_as,
        ↳var_name = token.split_contents()
    except ValueError:
        try:
            tag_name, manager_method, str_from, appmodel, str_as, var_name =
            ↳token.split_contents()
        except ValueError:
            raise template.TemplateSyntaxError, "Znacznik get_objects ma
            ↳następującą składnię: {% get_objects [<manager>.]<method> from
            ↳<app_name>.<model_name> [limit <amount>] as <var_name> %}"
    try:
        app_name, model_name = appmodel.split(".")
    except ValueError:
        raise template.TemplateSyntaxError, "Znacznik get_objects wymaga nazwy
        ↳aplikacji i modelu rozdzielonych kropką."
    model = models.get_model(app_name, model_name)
    return ObjectsNode(model, manager_method, amount, var_name)
```

2. Następnie w tym samym pliku utwórz klasę węzłową:

```
class ObjectsNode(template.Node):
    def __init__(self, model, manager_method, amount, var_name):
        self.model = model
        self.manager_method = manager_method
        self.amount = amount
        self.var_name = var_name

    def render(self, context):
        if "." in self.manager_method:
            manager, method = self.manager_method.split(".")
        else:
            manager = "_default_manager"
            method = self.manager_method

        qs = getattr(
            getattr(self.model, manager),
            method,
            self.model._default_manager.none,
        )()
        if self.amount:
            amount = template.resolve_variable(self.amount, context)
            context[self.var_name] = qs[:amount]
        else:
            context[self.var_name] = qs
        return ""
```

Jak to działa

Znacznik szablonowy `{% get_objects %}` ładuje obiekt QuerySet zdefiniowany przez metodę `manager` z określonych aplikacji i modelu, ogranicza wyniki do podanej ilości oraz zapisuje wynik w zmiennej kontekstowej.

Poniżej znajduje się najprostszy przykład użycia utworzonego przez nas znacznika szablonowego. Ładuje on pięć artykułów do dowolnego szablonu:

```
{% load utility_tags %}
{% get_objects all from news.Article limit 5 as latest_articles %}
{% for article in latest_articles %}
    <a href="{{ article.get_url_path }}">{{ article.title }}</a>
{% endfor %}
```

Wykorzystujemy tu metodę `all` domyślnego menedżera `objects` modelu `Article`. Artykuły zostaną posortowane przez atrybut `ordering` zdefiniowany w klasie `Meta`.

W bardziej zaawansowanym przypadku można by było utworzyć własnego menedżera z własną metodą do odpytywania obiektów z bazy danych. Menedżer jest interfejsem dostarczającym operacje zapytaniowe na bazie danych do modeli. Każdy model ma domyślnie przynajmniej

jednego menedżera o nazwie `objects`. W ramach przykładu utworzymy model `Artist`, który może mieć status `draft` lub `published`, i nowego menedżera `custom_manager` umożliwiającego losowe wybieranie opublikowanych artystów:

```
#artists/models.py
# -*- coding: UTF-8 -*-
from django.db import models
from django.utils.translation import ugettext_lazy as _

STATUS_CHOICES = (
    ('draft', _("Szkiec")),
    ('published', _("Opublikowany")),
)

class ArtistManager(models.Manager):
    def random_published(self):
        return self.filter(status="published").order_by('?')

class Artist(models.Model):
    # ...
    status = models.CharField(_("Status"), max_length=20,
                               choices=STATUS_CHOICES)
    custom_manager = ArtistManager()
```

Aby załadować losowego opublikowanego artystę, należy dodać poniższy kod do dowolnie wybranego szablonu:

```
{% load utility_tags %}
{% get_objects custom_manager.random_published from artists.Artist
limit 1 as random_artists %}
{% for artist in random_artists %}
    {{ artist.first_name }} {{ artist.last_name }}
{% endfor %}
```

Przyjrzyjmy się kodowi znacznika szablonowego. W funkcji przetwarzającej spodziewany jest jeden z dwóch formatów: z limitem lub bez limitu. Łańcuch jest przetwarzany, model zostaje rozpoznany i składniki znacznika szablonowego zostają przekazane do klasy `ObjectNode`.

W metodzie `render` klasy węzłowej sprawdzamy nazwę menedżera i nazwę jego metody. Jeśli nie jest on zdefiniowany, zostanie użyty domyślny `_default_manager`, który w większości przypadków jest taki sam jak `objects`. Potem wywołujemy metodę menedżera i jeśli metoda ta nie istnieje, awaryjnie używamy pustego obiektu `QuerySet`. Jeżeli podany jest limit, określamy jego wartość i ograniczamy `QuerySet`. Na koniec zapisujemy obiekt `QuerySet` w zmiennej kontekstowej.

Zobacz również

- Receptura „Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje”
- Receptura „Tworzenie znacznika szablonowego do przetwarzania treści jako szablonu”
- Receptura „Tworzenie znacznika szablonowego do modyfikowania parametrów zapytań”

Tworzenie znacznika szablonowego do przetwarzania treści jako szablonu

W tej recepturze pokazuję, jak utworzyć znacznik szablonowy o nazwie {% parse %}, który umożliwia wstawianie fragmentów szablonów do bazy danych. Jest to przydatne, gdy trzeba dostarczać inną treść użytkownikom uwierzytelnionym niż niewierzytelnionym, gdy chce się wyświetlać spersonalizowane powitania albo gdy nie chce się wpisywać na sztywno ścieżek do mediów do bazy danych.

Przygotowanie

Jak się pewnie spodziewasz, pracę zaczniemy od aplikacji `utils`, która powinna być już zainstalowana i gotowa do definiowania znaczników szablonowych.

Jak to zrobić

Znaczniki szablonowe składają się z funkcji przetwarzającej argumenty przekazane do znacznika i klasy węzłowej implementującej logikę oraz wynik zwracany przez znacznik:

1. Najpierw utwórz następującą funkcję przetwarzającą argumenty znacznika szablonowego:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
from django import template
register = template.Library()

### ZNACZNIKI ###

@register.tag
def parse(parser, token):
    """
    Przetwarza wartość jako szablon i ją drukuje lub zapisuje w zmiennej.
    Sposób użycia:
        {% parse <template_value> [as <variable>] %}
    Przykłady:
        {% parse object.description %}
        {% parse header as header %}
        {% parse "{{ MEDIA_URL }}" as js_url %}
    """
    bits = token.split_contents()
    tag_name = bits.pop(0)
    try:
        template_value = bits.pop(0)
        var_name = None
        if len(bits) == 2:
```

```

        bits.pop(0) #usuń słowo „as”
        var_name = bits.pop(0)
    except ValueError:
        raise template.TemplateSyntaxError, "Znacznik parse ma następującą
        ↳składnię: {% parse <template_value> [as <variable>] %}"
    return ParseNode(template_value, var_name)

```

2. Następnie w tym samym pliku utwórz klasę węzłową:

```

class ParseNode(template.Node):
    def __init__(self, template_value, var_name):
        self.template_value = template_value
        self.var_name = var_name

    def render(self, context):
        template_value = template.resolve_variable(self.template_value, context)
        t = template.Template(template_value)
        context_vars = {}
        for d in list(context):
            for var, val in d.items():
                context_vars[var] = val
        result = t.render(template.RequestContext(context['request'], context_vars))
        if self.var_name:
            context[self.var_name] = result
        return ""
    return result

```

Jak to działa

Znacznik szablonowy `{% parse %}` umożliwia przetwarzanie wartości jako szablonu i natychmiastowe jego renderowanie lub zapisanie go jako zmiennej kontekstowej.

Jeśli mamy obiekt z polem `description`, które może zawierać zmienne szablonowe lub logikę, to możemy go przetwarzać i renderować przy użyciu poniższego kodu:

```

{% load utility_tags %}
{% parse object.description %}

```

Można też zdefiniować wartość do przetworzenia przy użyciu łańcucha w cudzysłowie:

```

{% load utility_tags %}
{% parse "{{ STATIC_URL }}"|site/img/" as img_path %}


```

Spójrzmy na kod naszego znacznika szablonowego. Funkcja przetwarzająca sprawdza po kolei jego argumenty. Najpierw oczekujemy nazwy `parse`, następnie wartości szablonu, później opcjonalnego słowa `as` i na końcu nazwy zmiennej kontekstowej. Wartość szablonu i nazwa zmiennej są przekazywane do klasy `ParseNode`. Metoda `render` tej klasy rozwiązuje wartość zmiennej szablonowej i tworzy z niej obiekt szablonu. Później renderuje ten szablon ze wszystkimi zmiennymi kontekstowymi. Jeśli podana jest nazwa zmiennej, to wynik zostaje zapisany w tej zmiennej. W przeciwnym razie wynik zostaje od razu wyświetlony.

Zobacz również

- Receptura „Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje”
- Receptura „Tworzenie znacznika szablonowego do ładowania zestawu obiektów do szablonu”
- Receptura „Tworzenie znacznika szablonowego do modyfikowania parametrów zapytań”

Tworzenie znacznika szablonowego do modyfikowania parametrów zapytań

System Django ma wygodny w użyciu i elastyczny system do tworzenia kanonicznych i czystych adresów URL za pomocą reguł w postaci wyrażeń regularnych, które dodaje się do plików konfiguracji URL. Brakuje w nim jednak wbudowanego mechanizmu do zarządzania parametrami zapytań. Widoki takie jak wyniki wyszukiwania czy filtrowalne listy obiektów muszą przyjmować parametry pozwalające przeglądać przefiltrowane wyniki przy użyciu innego parametru lub przejść do innej strony. W tej recepturze utworzymy znacznik szablonowy o nazwie `{% append_to_query %}`, umożliwiający dodawanie, zmienianie i usuwanie parametrów bieżącego zapytania.

Przygotowanie

Ponownie pracę zaczniemy od aplikacji `utils`, która powinna być dodana do sekcji `INSTALLED_APPS` i zawierać pakiet `templatetags`.

Ponadto w ustawieniu `TEMPLATE_CONTEXT_PROCESSORS` powinien być ustawiony procesor kontekstu `request`:

```
#settings.py
TEMPLATE_CONTEXT_PROCESSORS = (
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.core.context_processors.tz",
    "django.contrib.messages.context_processors.messages",
    "django.core.context_processors.request",
)
```

Jak to zrobić

W tym znaczniku wykorzystamy dekorator `simple_tag`, przetwarzający składniki i wymagający od programisty zdefiniowania tylko funkcji renderującej:

```
#utils/templatetags/utility_tags.py
# -*- coding: UTF-8 -*-
import urllib
from django import template
from django.utils.encoding import force_str
register = template.Library()

### ZNACZNIKI ###

@register.simple_tag(takes_context=True)
def append_to_query(context, **kwargs):
    """ Renderuje odnośnik ze zmodyfikowanymi zapytaniami bieżącego zapytania. """
    query_params = context['request'].GET.copy()
    for key, value in kwargs.items():
        query_params[key] = value
    query_string = u""
    if len(query_params):
        query_string += u"?%s" % urllib.urlencode([
            (key, force_str(value)) for (key, value) in query_params.iteritems()
        ])
    return query_string
```

Jak to działa

Znacznik szablonowy `{% append_to_query %}` wczytuje bieżące parametry zapytania z podobnego do słownika obiektu `request.GET` typu `QueryDict` do nowego słownika o nazwie `query_params` i za pomocą pętli przegląda parametry przekazane do znacznika szablonowego, aktualizując wartości. Następnie tworzony jest nowy łańcuch zapytania, wszystkie spacje i znaki specjalne są kodowane do użytku w adresie URL, a znaki & łączące parametry zapytań zostają zastąpione encjami. Nowo powstały łańcuch zapytania zostaje zwrócony do szablonu.

Więcej informacji o obiektach `QueryDict` można znaleźć w oficjalnej dokumentacji Django na stronie <https://docs.djangoproject.com/en/1.6/ref/request-response/#querydict-objects>.

Spójrzmy na przykład użycia znacznika `{% append_to_query %}`. Jeśli bieżący adres to `http://127.0.0.1:8000/artists/?category=fine-art&page=1`, to za pomocą poniższego znacznika szablonowego możemy wygenerować odnośnik do następnej strony:

```
{% load utility_tags %}
<a href="{% append_to_query page=2 %}">2</a>
```

Oto wynik wyrenderowany przez ten znacznik:

```
<a href="?category=fine-art&page=2">2</a>
```

A za pomocą poniższego znacznika szablonowego wyrenderujemy odnośnik zerujący stronicowanie i przechodzący do innej kategorii:

```
{% load utility_tags i18n %}
```

```
<a href="{% append_to_query category='sculpture' page='' %}">{% trans "Rzeźba" %}</a>
```

Oto wynik wyrenderowany przez ten znacznik:

```
<a href="?category=sculpture">Rzeźba</a>
```

Zobacz również

- Receptura „Filtrowanie list obiektów” z rozdziału 3. „Formularze i widoki”
- Receptura „Tworzenie znacznika szablonowego do dołączania szablonu, jeśli istnieje”
- Receptura „Tworzenie znacznika szablonowego do ładowania zestawu obiektów do szablonu”
- Receptura „Tworzenie znacznika szablonowego do przetwarzania treści jako szablonu”

Skorowidz

A

- administracja kategoriami, 184, 186, 187
- adres e-mail, 232
- Ajax, 107, 108
- algorytm MPIT, 179
- Apache, 234
- aplikacja
 - books, 223
 - bulletin_board, 214
 - BulletinFeed, 209
 - cms_extensions, 173
 - custom_admin, 147
 - django-ajax-uploader, 108
 - django-crispy-forms, 70
 - django-mptt-admin, 184
 - django-mptt-tree-editor, 186
 - editorial, 168
 - email_messages, 62
 - guerrilla_patches, 221, 222
 - likes, 103
 - locations, 93, 149
 - movies, 74, 100
 - products, 141
 - quotes, 66, 111
 - utils, 85, 91, 120, 229
- aplikacje CMS, 164
- apphooks, 166
- arkusz Excel, 144
- arkusze stylów, 95
- atributy danych HTML5, 93, 94

B

- biblioteka
 - Fabric, 242
 - jQuery jqTree, 185
- buforowanie wartości metody, 231

C

- CMS, 157
- CSS, 169
- CSV, comma-separated values, 197

D

- data, 42
- debugowanie, 226
- definiowanie
 - ścieżek względnych, 23
 - ustawień aplikacji, 35
- dekorator
 - register.filter, 121
 - simple_tag, 134
- diagnozowania błędów, 199
- Django 1.6, 15
- Django CMS, 157
- dodatek
 - bpython, 218
 - IPython, 218
- dodawanie
 - nawigacji, 166
 - pól, 173

dołączanie
 szablonu, 124, 126
 ustawień JavaScript, 90
 ustawień lokalnych, 28
 zewnętrznych zależności, 21
 domieszka
 CreatorMixin, 230
 MPTTModel, 182
 domieszki do modeli
 adresy URL, 39
 modyfikowanie dat, 42
 obsługa metaznaczników, 43
 obsługa relacji generycznych, 46
 stosowanie, 38
 tworzenie dat, 42
 domieszki do wielokrotnego użytku, 38
 dostarczanie danych, 214
 dostosowywanie kolumn, 138

E

efektywność instrukcji SQL, 227
 eksportowanie danych, 197
 element <div>, 96
 encja, 134

F

filtr, 79, 234
 humanize_url, 123, 124
 szablonowy, 117
 adresy URL, 123
 pobieranie obiektu, 121
 pokazywanie dni, 119
 filtrowanie listy
 zmian, 144
 obiektów, 74
 format
 CSV, 201
 JSON, 201
 XML, 207
 formaterzy, 234
 formularz MovieFilterForm, 75
 formularze, 61
 metoda save, 64
 obiekt HttpRequest, 61
 układ, 71
 wstawianie map, 149
 wybieranie kategorii, 190
 wybieranie wielu kategorii, 191

funkcja
 autocompleteAddress, 152
 csv.reader, 199
 export_xls, 142
 get_git_changeset, 27
 get_media_svn_revision, 26
 getAddress4search, 151
 getattr, 36
 mark_safe, 191
 slugify, 220, 221
 updateAddressFields, 153
 updateLatitudeAndLongitude, 152
 updateMarker, 152

G

generowanie
 kategorii w szablonie, 188
 pól, 194
 guerrilla patching, 217, 220

H

hak aplikacji, 165, 166
 HTML5, 93

I

implementacja
 ciągłego przewijania, 100
 widżetu polubień, 102
 importowanie
 danych, 197
 z pliku CSV, 197
 z pliku Excela, 199
 z pliku JSON, 201
 z pliku XML, 206
 plików, 33
 indykatory, 92
 informacje o
 filmach, 198
 ignorowanych zasobach, 32
 interfejs administracyjny, 184, 186

J

JavaScript, 87
 języki serwisu, 161

K

- kanal RSS, 209
- kasowanie sesji, 240
- kategorie
 - hierarchiczne, 181
 - importowe, 34
- klasa
 - BulletinFeed, 213
 - BulletinFilterForm, 211
 - ChangeList, 224
 - CMSAttachMenu, 168
 - CMSPluginBase, 169
 - Command, 199
 - CreationModificationDateMixin, 43
 - CSS, 169
 - CSSExtension, 173
 - CSSExtensionToolbar, 176
 - domieszkowa, 38
 - FilterableListView, 85
 - GroupAdmin, 148
 - img-full-width, 96
 - Meta, 216
 - ModelResource, 216
 - MoviesApphook, 165
 - MoviesMenu, 167
 - NavigationNode, 168
 - PageExtension, 176
 - PhotoFilter, 146
 - ProductAdmin, 143
 - SimpleListFilter, 145
 - TreeForeignKey, 183
 - TreeManyToManyField, 183
 - UserAdmin, 148
 - View, 83
- klasy węzłowe, 129
- klucz
 - Api, 215
 - obcy, 58, 216
- kodowanie
 - LATIN1, 29
 - UTF-8, 29
- kolejność importowania plików, 33
- komponowanie widoków, 83
- konfiguracja
 - menu stron, 161
 - MySQL, 29
 - rejestratora dziennikowego, 233

- konstrukcja `{{ child.cssextension }}`, 178
- kontrolki stronicowania, 82
- konwertowanie aplikacji, 164
- krotka, 143

L

- LATIN1, 29
- layout, 71
- lista
 - kategorii, 79
 - krotek, 143
 - obiektów, 74
 - pól wyboru, 191
 - zmian, 138, 143, 144
- listy stronicowane, 80

Ł

- ładowanie zestawu obiektów, 127

M

- mapa, 149, 155
- menu stron, 161
- metaznacznik, 43, 166
- metoda
 - as_view(), 83
 - feed_url, 213
 - GET, 83
 - get_absolute_url(), 39
 - get_object, 213
 - get_page, 83get_queryset_and_facets, 83, 85
 - get_thumbnail_picture_url, 69
 - get_translated_list, 225
 - get_url(), 41
 - get_url_path(), 41
 - handle, 204
 - image.save, 205
 - POST, 83
 - render, 130
 - request.get, 205
 - response.get, 208
 - save, 64, 110
 - save(), 43
 - save_page, 208
 - title, 213
- migracja South, 55, 57

model

- administracji Django, 223
- Book, 223
- Bulletin, 210
- Category, 181
- CSSExtension, 174
- Movie, 181, 182
- Track, 203
- modelowanie panelu administracyjnego, 137
- moduł
 - mod_wsgi, 234
 - utils, 227
- modułowa budowa klasy, 83
- modyfikator menu, 175
- modyfikowanie parametrów zapytań, 133
- monkey patching, 217, 220, 224

N

- narzędzia diagnostyczne, 227
- narzędzie
 - easy_install, 17
 - pip, 16, 20
 - South, 58
 - ThreadLocalMiddleware, 229
 - virtualenv, 16
 - Virtualmin, 234, 235
- nawigacja, 166

O

- obiekt
 - HttpRequest, 61, 63, 229, 230
 - HttpResponse, 143
 - jScroll, 102
 - QueryDict, 134
 - QuerySet, 130, 146
 - request, 62, 63
- obiekty śródliniowe, 139
- obsługa
 - metaznaczników, 43
 - pól wielojęzycznych, 50
 - rejestracji danych, 232
 - relacji generycznych, 46
- opcje administracyjne, 174

P

- pakiet
 - commands, 200
 - utils, 46, 50
 - xlr, 200
- panel
 - administracyjny, 137
 - Virtualmin, 235
- pasek narzędzi debugowania, 226
- plik
 - .bash_profile, 33
 - .gitignore, 32
 - .htaccess, 241
 - .htaccess_live, 241
 - .htaccess_under_construction, 241
 - __init__.py, 104
 - admin.py, 139, 142, 145, 147
 - app_settings.py, 35
 - base.html, 88, 161
 - base.py, 39
 - base_simple.html, 89
 - cms_app.py, 165
 - cms_plugins.py, 169
 - cms_toolbar.py, 174, 176
 - fabfile.py, 242
 - feeds.py, 211
 - fields.py, 50, 192
 - forms.py, 62, 190
 - likes_tags.py, 104
 - local_settings.py, 28, 31
 - locating.js, 151
 - log.py, 232
 - manage.py, 220
 - menu.py, 167
 - middleware.py, 229
 - misc.py, 27, 227
 - models.py, 35, 38, 42, 46, 181
 - my.conf, 29
 - myproject.wsgi, 241
 - requirements.txt, 56
 - settings.py, 23, 36, 226
 - urls.py, 97
 - utility_tags.py, 120, 122
 - views.py, 63, 67
- pliki
 - CSV, 197
 - Excela, 201
 - JSON, 201

- projektu, 18
- skompilowane, 32
- szablonowe, 88
- XML, 206
- poła wielojęzyczne, 50
- pole
 - pojedynczego wyboru, 190
 - wyboru kategorii, 191
- powłoka
 - Django, 218
 - Pythona, 220
- procedury obsługowe, 234
- program, *Patrz* narzędzie
- przeglądanie węzłów, 208
- przetwarzanie treści, 131
- przewijanie ciągle, 100

R

- raport o błędach, 232
- rejestratory, 234
- relacje generyczne, 46
- repozytorium, 225
- responsywny projekt, 70

S

- sekcja
 - <body>, 90
 - <head>, 90
- serwer Apache, 234
- skrypt, 98, 101, *Patrz także* plik
- skrypt Fabric, 242, 247, 248
- skryptyzacja, 226
- słownik, 134
- stos, 225
- stronicowanie, 80, 81
- strony CMS-a, 173
- struktura
 - bazy danych, 37
 - plików projektu, 18
- struktury hierarchiczne, 179
- system
 - Git, 26, 32
 - Subversion, 25, 30
 - szkieletowy Bootstrap 3, 79
 - własny migracji, 57
- szablon, 55, 68, 87, 193, 212
 - base.html, 158

- default.html, 160
- editorial_content.html, 171
- magazine.html, 170, 172
- settings.js, 92
- start.html, 160
- szablony aplikacji django-crispy-forms, 74
- szczegóły obiektu, 97

Ś

- ścieżka
 - picture_path, 115
 - względna, 23
- środowisko wirtualne, 16

T

- technologia Ajax, 107, 108
- tryb DEBUG, 225
- tworzenie
 - czynności administracyjnych, 141
 - domieszek do modeli, 39, 42, 46
 - filtrów, 117
 - filtrów listy zmian, 144
 - interfejsu administracyjnego, 184
 - kanałów RSS, 209
 - kategorii hierarchicznych, 181
 - klasy węzłowej, 132
 - klucza, 215
 - przyjaznych adresów, 220
 - responsywnych projektów, 70
 - skryptu wdrażania, 241
 - struktury plików, 18
 - ustawień lokalnych, 28
 - wtyczki, 168
 - znaczników, 117

U

- układ formularza, 71
- usługa Tastypie, 214
- ustawienia
 - administracyjne, 147
 - JavaScript, 90
 - lokalne, 28
- usuwanie skompilowanych plików, 32
- UTF-8, 29
- użycie
 - metody save, 64
 - migracji South, 55

W

wartość zmiennej `STATIC_URL`, 25, 26
 wdrażanie aplikacji, 234
 w środowisku produkcyjnym, 246
 w środowisku rozwojowym, 243
 w środowisku testowym, 244
 widok, 61, 83
 `MovieListView`, 83
 `render_js`, 91
 widżet
 `CheckboxSelectMultiple`, 195
 `Like`, 127
 polubień, 102
 wysyłania plików, 109, 112
 wielojęzyczne pole
 tekstowe, 50
 znakowe, 50
 własność
 `ignore`, 30
 `layout`, 71
 `list_display`, 141
 `short_description`, 141
 współrzędne geograficzne, 149
 wstawianie map, 149
 wtyczka, 168
 `EditorialContent`, 169
 `jScroll`, 100
 wybieranie kategorii, 190, 191
 wyskakujące okienko, 97
 wysyłanie
 obrazów, 66
 plików, 113
 raportów o błędach, 232

X

XPath, 209

Z

zależności
 projektu, 21
 zewnętrzne, 21
 zamienianie ustawień administracyjnych, 147
 zarządzanie
 listami stronicowanymi, 80
 treścią, 157
 zależnościami projektu, 20
 zastępowanie modelu administracji, 222
 zmienianie klucza obcego, 58
 zmienna
 `DJANGO_SETTINGS_MODULE`, 220
 `EXTERNAL_APPS_PATH`, 23
 `EXTERNAL_LIBS_PATH`, 23
 `PROJECT_PATH`, 23
 `STATIC_URL`, 25, 26
 zmienne
 kontekstowe szablonu, 228
 lokalne, 225
 znacznik
 `{% append_to_query %}`, 82, 133, 134
 `{% get_objects %}`, 129
 `{% include %}`, 126
 `{% like_widget %}`, 104
 `{% like_widget for object %}`, 107
 `{% parse %}`, 132
 `{% placeholder %}`, 161
 `{% recursetree %}`, 188
 `{% show_menu_below_id %}`, 161, 163
 `{% static_placeholder %}`, 161
 znacznik `{% try_to_include %}`, 125, 126
 znacznik szablonowy, 77, 117
 dołączanie szablonu, 124
 ładowanie zestawu obiektów, 127
 modyfikowanie parametrów zapytań, 133
 przetwarzanie treści, 131
 znak `@`, 73

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



Helion SA

Aplikacje internetowe z Django

Najlepsze receptury

Django to szkielet napisany w języku Python, pozwalający na błyskawiczne tworzenie zaawansowanych aplikacji internetowych. Dostarcza wielu narzędzi, które przyspieszają i znacząco ułatwiają pracę programisty. Wśród nich znajdziesz system szablonów, system mapowania obiektowo-relacyjnego oraz automatyczny interfejs do zarządzania treścią. Ale to nie wszystko!

Przekonaj się, co jeszcze potrafi Django. Ta książka jest doskonałym źródłem informacji na temat tego systemu. Wiedza, którą dzięki niej zdobędziesz, jest przedstawiona w formie receptur. Kolejne rozdziały to świetne przepisy na: zarządzanie zależnościami, tworzenie struktury bazy danych, projektowanie formularzy i przetwarzanie pozyskanych z nich danych oraz korzystanie z systemów szablonów. Dowiesz się stąd, jak tworzyć własne filtry i znaczniki w szablonach, modelować panel administracyjny oraz korzystać z Django CMS. Poznasz także system mapowania obiektowo-relacyjnego oraz wdrożysz aplikację na serwerze. Dzięki tym znakomitym przepisom możesz błyskawicznie nauczyć się pracować z Django oraz uruchomić swoją pierwszą aplikację napisaną z jego wykorzystaniem!

Poznaj moc języka Python w tworzeniu aplikacji internetowych!



Dzięki tej książce:

- rozpoczniesz Twoją przygodę ze szkieletem Django
- zbudujesz odpowiednią strukturę katalogów
- skorzystasz z systemu szablonów
- poznasz system Django CMS
- uruchomisz aplikację na serwerze Apache
- wykorzystasz potencjał szkieletu Django

Aidas Bendoraitis — ekspert w zastosowaniu Django oraz jQuery. Interesuje się tworzeniem aplikacji mobilnych i internetowych oraz gier. Aktualnie zajmuje stanowisko starszego programisty w firmie Studio 38 oraz architekta w BeHype.

[PACKT] open source
PUBLISHING community experience distilled

Helion

35087 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne

☎ **0 801 339900**

📞 **0 601 339900**

Sprawdź najnowsze promocje:

● <http://helion.pl/promocje>

Książki najchętniej czytane:

● <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

● <http://helion.pl/novosci>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-1107-7



9 788328 311077

Informatyka w najlepszym wydaniu

cena: 49,00 zł