Technologia i rozwiązania

Aplikacje internetowe z Django

Najlepsze receptury

Wykorzystaj potencjał Django!

Aidas Bendoraitis

Helion

[PACKT] open source community experience distilled PUBLISHING

Tytuł oryginału: Web Development with Django Cookbook

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-1107-7

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

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

Spis treści

ROZDZIAŁ 5

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 _(u"dziś")
    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 wywoływalnego 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 objaśnień. 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 filtru szablonowego do pobierania pierwszego obiektu mediów”
- Receptura „Tworzenie filtru szablonowego dostosowującego adresy URL do potrzeb użytkownika”

Tworzenie filtru 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 `&lt;`, `&gt;` oraz `&amp;`. 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', _("Szkic")),
    ('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 nieuwierzytelnionym, 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 }}js/" 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 %}
<img src="{{ img_path }}someimage.png" alt="" />
```

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()
            ↪if value
        ]).replace("&", "&amp;")
    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&amp;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

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Aplikacje internetowe z Django Najlepsze receptury

Django to szkielet napisany w języku Python, pozwalający na błyskawiczne tworzenie zaawansowanych aplikacji internetowych. Dostarcza wielu narzędzi, które przyśpieszają 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/nowosci

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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-1107-7

cena: 49,00 zł