Django - funkcjonalności dla użytkowników

Zajęcia Prymus 2020

Agnieszka Rudnicka, rudnicka@agh.edu.pl

Komponenty w szablonach

Kod lubi się powtarzać. Prędzej czy później każdy znajduje jakiś kawałek, dłuższy lub krótszy, który powtarza się słowo w słowo i trzeba go w wielu miejscach utrzymywać w podobnym stanie. W przypadku szablonów HTML jest to wyjątkowo uciążliwe. Na szczęście są komponenty.

Tag `{% include %}`

Django pozwala na wydzielanie fragmentów kodu do osobnych plików. Potem taki fragment możemy załączyć za pomocą tagu `{% include "scieżka/do/szablonu.html" %}`.

Weźmy pod lupę listę książek. Element tej listy może się powtarzać na:

- głównej liście książek;
- liście książek napisanych przez autora na jego stronie profilowej;
- liście książek podobnych w opisie pojedynczej książki;

i tak dalej. Nie trudno jest znaleźć miejsca gdzie chcielibyśmy użyć tego samego kodu (komponentu).

Komponent książki

Wydzielmy więc fragment kodu HTML, który opisuje pojedynczy kafel z książką. W moim przypadku było to coś takiego:

```
{% load static %}

<div class="col-12 col-md-3 mb-3">
    <div class="card h-100">
        <img src="{% static 'sources/img/sky.jpg' %}" class="card-img-top" alt="...">
        <div class="card-body">
            <h5 class="card-title">{{ book.title }}</h5>
            <p class="card-text">{{ book.short_description|truncatewords:10 }}</p>
            <a href="{% url 'book_details' book.id %}" class="btn btn-primary">
                Szczegóły <i class="fa fa-arrow-right"></i>
            </a>
        </div>
    </div>
</div>
```

Zapiszmy to do osobnego pliku, np `templates/components/book_tile.html`.

Warto zauważyć, że skopiowaliśmy tylko kod HTML dotyczący pojedynczej książki, nie całą listę z `book_list.html`.

Ważne jest też dodane `{% load static %}` ponieważ pracujemy w nowym pliku i tutaj ten tag nie został jeszcze zaimportowany. Bez niego nie zadziała wstawianie plików statycznych za pomocą `{% static 'sources/img/sky.jpg' %}`. W moim przypadku to domyślne tło książki, ponieważ nie mam jeszcze pola w modelu na zdjęcie okładki.

Wróćmy teraz do szablony `book_list.html` z którego został skopiowany kod kafla książki.

```
{% for book in books %}

    {% include "componenets/book_tile.html" %}

{% endfor %}
```

Wystarczy, że użyjemy tagu `{% include %}` aby wykorzystać nasz nowo utworzony komponent.

Strona powinna działać bez zmian z punktu widzenia użytkownika. Za to my zyskaliśmy komponent, którego możemy używać za każdym razem, gdy chcemy wyświetlić książkę i w razie jakichkolwiek zmian, wystarczy, że edytujemy jeden plik, a nie wszystkie jego wystąpienia w repozytorium.

Formularze

Formularze są podstawowym sposobem komunikacji z użytkownikiem. My jako deweloperzy budujemy formularz i wysyłamy go do przeglądarki użytkownika, ten go wypełnia (albo i nie) i przysyła nam dane, zwykle poprzez zatwierdzenie formularza przyciskiem.

Konstrukcja prosta i stara, która do działania potrzebuje:

- elementu HTML `<form>`;
- informacji o metodzie przesłania danych (np `method="POST"`, domyślna to GET);
- przycisku pozwalającego na zatwierdzenie formularza (`<button type="submit">Wyślij</button>`);

... oraz oczywiście pól, w których użytkownik może wpisać lub wyklinać dane do przesłania.

Podstawowe formularze w Django

Django pozwala na zdefiniowanie klasy opisującej formularz. Całą pracę związaną z przetwarzaniem realizuje framework, my jedynie definiujemy potrzebne pola.

Stwórzmy nowy plik `forms.py` w katalogu naszej aplikacji `books`.

```
from django import forms

class SimpleForm(forms.Form):
    name = forms.CharField(label="Jak masz na imie?")
```

Powyższa klasa definiuje formularz z jednym polem tekstowym. Jest też podana etykieta (`label=`), czyli coś co pokażemy użytkownikowi w formularzu na stronie.

Zdefiniujmy standardowo widok w `views.py`:

```
from django.views.generic import FormView
from books.forms import SimpleForm

class SimpleFormView(FormView):
    form_class = SimpleForm
    template_name = "books/simple_form.html"
```

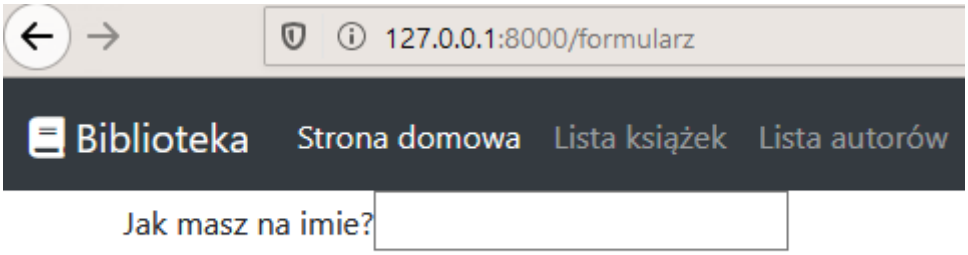
Podepnijmy do routingu w `urls.py`:

```
    path("formularz", views.SimpleFormView.as_view()),
```

I na koniec dodajmy szablon HTML (`books/simple_form.html`) jak zdefiniowaliśmy w widoku.

```
{% extends "base.html" %}

{% block content %}
    {{ form }}
{% endblock %}
```

Wystarczy ☺



Proste, prawda? Problem w tym, że nasz widok nie wiele robi z przesłanymi danymi... których jeszcze nie da się przesłać. Potrzebujemy dodać przycisk zatwierdzający formularz oraz tag HTMLowy `<form>`.

Wyświetlanie formularzy w HTML

Poprawmy szablon `simple_form.html` aby zawierał potrzebne elementy do interakcji z użytkownikiem.

```
<form method="post">
    {% csrf_token %}

    {{ form }}

    <button type="submit">Wyślij</button>
</form>
```

Mamy tu wspomniane wcześniej tagi HTML `<form>`, informacje o metodzie przesłania danych (`POST`), token zabezpieczający formularz przed atakami CSRF (`{% csrf_token %}`), formularz generowany przez framework Django `{{ form }}` oraz guzik zatwierdzający formularz.

Na naszej stronie powinien się teraz pojawić guzik. Spróbujmy wypełnić formularz i zatwierdzić.

```
ImproperlyConfigured at /formularz

No URL to redirect to. Provide a success_url.

Request Method:     POST
Request URL:    http://127.0.0.1:8000/formularz
```

Powyższy błąd wskazuje, że jesteśmy na dobrej drodze. Dane zostały otrzymane przez backend, ale Django nie wie co zrobić po ich przetworzeniu. Innymi słowy musimy podać `success_url`, który jest adresem strony do przekierowania po wypełnieniu formularza. W przypadku logowania się użytkownika i wylogowania mieliśmy odpowiednio stronę profilową oraz stronę główną.

Akcje po wypełnieniu formularza

Możemy zmienić nasz widok, żeby przekierować użytkownika na stronę główną:

```
class SimpleFormView(FormView):
    form_class = SimpleForm
    template_name = "books/simple_form.html"
    success_url = "/"  # <-- dodane
```

Albo pójść o krok dalej i wyświetlić stronę z podziękowaniem za przesłanie wypełnionego formularza. Zróbmy to drugie. Dodajmy do naszego widoku metodę `def form_valid(self, form):`.

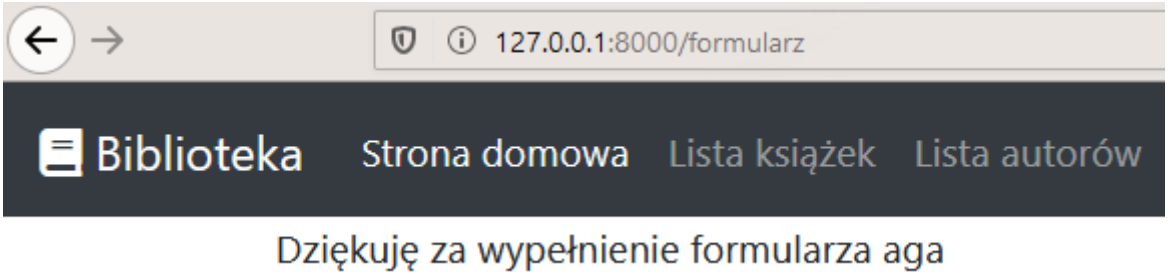
```python
class SimpleFormView(FormView):
    form_class = SimpleForm
    template_name = "books/simple_form.html"

    def form_valid(self, form):
        name = form.cleaned_data["name"]
        return render(
            request=self.request,
            template_name="books/simple_form_success.html",
            context={"name": name}
        )
```

Dopisanie `form_valid` pozwala nam wykonanie zdefiniowanych przez nas akcji po przetworzeniu formularza przez Django. Dodatkowo jest to metoda automatycznie wywoływana przez framework tylko jeśli dane w formularzu przeszły walidację. Czyli między innymi format był poprawny i wszystkie wymagane pola wypełnione.,

`name = form.cleaned_data["name"]` pozwala nam odczytanie zwalidowanych danych ze słownika `form.cleaned_data` aby następnie przesłać je w kontekście do szablonu. Warto zauważyć, że jest to INNY szablon niż przedtem: `books/simple_form_success.html`. Jego zawartość, to:

```
{% extends "base.html" %}

{% block content %}
    Dziękuję za wypełnienie formularza {{ name }}
{% endblock %}
```

Efekt końcowy



Oczywiście w znakomitej większości przypadków formularze służą nam do edycji i tworzenia danych, które zapisujemy w bazie danych. Moglibyśmy sami ręcznie pisać tego typu logikę w metodzie `form_valid`, ale byłoby to bardzo powtarzalne. Znów mielibyśmy kod, który robi praktycznie to samo dla każdego modelu.

Dlatego Django dostarcza specjalny typ formularza - `ModelForm`.

ModelForm, czyli automatyczne formularze z modeli

Programiści frameworka nas wyręczają w trudach tworzenia formularzy, które miałyby zapisywać dane do bazy.

Aby stworzyć formularz, który pokazuje pola bazując na modelu w pliku `forms.py` piszemy:

```
from django import forms
from books.models import Book


class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        exclude = []
```

Definiujemy klasę formularza, która dziedziczy po `forms.ModelForm` z frameworka Django. Następnie w podklasie `class Meta:` definiujemy metadane dotyczące naszego formularza, czyli jaki model ma być użyty do wygenerowania. Piszemy też `exclude = []`, żeby przekazać, że nie chcemy wykluczać żadnych pól z formularza.

Zawsze musi być podana albo lista nazw pól, które mają być w formularzy `inlcude = ["title", ]` albo lista wykluczonych nazw pól `exclude = ["title", ]`. W przeciwnym przypadku dostaniemy wyjątek źle skonfigurowanego formularza modelowego: `django.core.exceptions.ImproperlyConfigured: Creating a ModelForm without either the 'fields' attribute or the 'exclude' attribute is prohibited; form BookForm needs updating.`.

Widok na podstawie formularza modelowego

```
from django.contrib.auth.mixins import LoginRequiredMixin
from books.forms import BookForm

class BookCreate(LoginRequiredMixin, CreateView):
    form_class = BookForm
```

Skorzystamy z klasy `CreateView` z Django, która implementuje obsługę widoków tworzenia. Podamy też nasz nowo stworzony formularz w konfiguracji. `LoginRequiredMixin` pozwoli nam na schowanie widoku dla niezalogowanych użytkowników. Jeśli taki użytkownik wejdzie na stronę dodawania książki, zostanie przekierowany do formularza logowania, a następnie z powrotem do formularza.

W `urls.py` standardowo nowy wpis:

```
path("ksiazki/nowa", views.BookCreate.as_view(), name="book_create"),
```

A także nowy szablon HTML:

```
{% extends "base.html" %}

{% block content %}
    <h1>Tworzenie nowej książki</h1>
    <form method="post">
        {% csrf_token %}
        {{ form }}
        <button type="submit">Wyślij</button>
    </form>
{% endblock %}
```

Akcja po zapisaniu formularza

Warto też dodać `success_url` do klasy widoku. Albo alternatywnie dopisać metodę `get_absolute_url` do modelu książki. W ten sposób za każdym razem jak dodamy lub edytujemy ją przez formularz Django zostaniemy przeniesieni automatycznie na widok szczegółów książki.

Przejdźmy do definicji modelu książki w `models.py` i dopiszmy `def get_absolute_url(self)` .

```
# ... model Book

    def __str__(self):
        return "Książka: " + self.title

    def get_absolute_url(self):
        return reverse("book_details", args=[self.pk])
```

Możemy teraz wejść na stronę http://127.0.0.1:8000/ksiazki/nowa i spróbować dodać książkę :)

Dodatkowe

Warto poświecić odrobinę czasu na poprawienie generowania naszych formularzy, żeby były ładniejsze i czytelniejsze. Pomogą nam w tym np zewnętrzne biblioteki, przykład użycia z Bootstrapem w poście: https://simpleisbetterthancomplex.com/2015/12/04/package-of-the-week-django-widget-tweaks.html

Wyszukiwanie

Jedna z częściej spotykanych funkcjonalności w serwisach internetowych. A czasem jedna z głównych (Google, Allegro, Ceneo etc.).

Formularz

W pasku nawigacyjnym umieśćmy malutki formularz z jednym polem i przyciskiem zatwierdzającym:

```
<form class="form-inline" action="{% url 'search' %}" method="get">
    <input class="form-control" type="text" placeholder="Szukaj..." name="q">
    <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button>
</form>
```

Zwróćmy uwagę na `action` oraz `method`. Mamy tu kolejno podany link do widoku, który zaraz zrobimy oraz metodę GET. Dzięki temu ostatniemu, parametry zapytania będą przekazywane przez adres URL w przeglądarce, np: http://127.0.0.1:8000/wyszukiwanie?q=test

Taki adres możemy przesłać komuś naszym ulubionym komunikatorem i powinien otrzymać te same wyniki.

Kolejna ważna rzecz, to `name="q"` w elemencie `<input>`. Dzięki temu formularz "wstawi" w pasek adresu `?q=zapytanie`. Gdybyśmy zmienili `name="fraza"`, to otrzymamy http://127.0.0.1:8000/wyszukiwanie?fraza=test.

Potrzebujemy nazwy tego pola, żeby odebrać informacje na backendzie.

Widok, który filtruje wyniki

Stwórzmy prosty widok wystukiwania w `views.py`.

```
def search(request):
    query = request.GET.get("q")
    if query:
        results = Book.objects.filter(title__icontains=query)
    else:
        results = []
    return render(request, template_name="search_results.html", context={"results":
results})
```

Sprawdzamy czy w informacjach przesłanych przez formularz (`request.GET`) jest informacja o frazie `q`, zgodnie z ustawionym `name="q"` w HTMLu formularza. Jeśli takową została przesłana wykorzystujemy `q` do przefiltrowania listy książek. Można ten fragment rozbudować i filtrować po autorach, książkach i recenzjach i zwracać 3 listy do szablony HTML, które potem odpowiednio wyrendererujemy.

Szablon HTML

Rezultaty przekazujemy do kontekstu HTML, żeby je wyświetlić użytkownikowi. Szablon (`search_results.html`) może wyglądać tak:

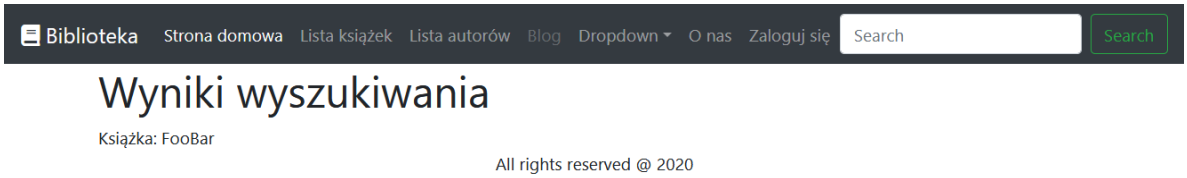
```
{% extends "base.html" %}

{% block content %}

    <h1>Wyniki wyszukiwania</h1>
    {% for result in results %}
        {{ result }}<br>
    {% empty %}
        <div class="alert alert-info">
            Brak wyników
        </div>
    {% endfor %}

{% endblock %}
```

Nie zapomnijmy dodać również odpowiedniego wpisu w `urls.py`, żeby można było dotrzeć do tego widoku.

```
    path("wyszukiwanie", views.search, name="search"),
```

Rezultat

To wszystko. Pora przetestować! Wyszukiwanie powinno działać na każdej stronie naszego portalu.



Wyniki wyszukiwania

Książka: FooBar

Dalsze prace

Pamiętacie niedziałający link do bloga na pasku tytułowym? :) To dobry sposób, żeby przećwiczyć i podsumować zdobyta wiedzę. W Internecie jest wiele przykładów jak można takiego bloga zrobić, dlatego też nie było to tematem naszych zajęć.

Z mojej strony polecam zrobienie bloga jako osobnej appki, tak jak na tych zajęciach mieliśmy aplikacje `books` w projekcie `libproj`, tak na aktualności może być jakiś `blog` albo `news`.

Źródła

- https://docs.djangoproject.com/en/3.0/topics/forms/#the-template
- https://docs.djangoproject.com/en/3.0/intro/tutorial01/
- https://simpleisbetterthancomplex.com/2015/12/04/package-of-the-week-django-widget-tweaks.html
- https://docs.djangoproject.com/en/3.0/topics/db/search/