Școala
informală
de IT

Python Development

Forms & ModelForms

Cuprins

Forms

Forms & ModelForms

Forms

- Form-urile sunt baza comunicării dintre paginile (template-uri) și business logic-ul (views-urile) aplicației noastre.
- Dacă paginile aplicației le putem accesa introducând URL-ul dorit în browser, prin metoda GET, pentru a trimite date către server vom folosi form-urile.
- Django ne oferă un ajutor în definirea și folosirea form-urilor prin sistemul deja existent.
- La baza acestui sistem stă clasa **Form**.
- La fel cum un model Django descrie structura logică a unui obiect, comportamentul și reprezentarea sa, clasa **Form** descrie și determină cum arată și funcționează un form.
- La instanțierea unui form putem opta să îl lăsăm gol sau să îl pre-populăm.
- Pre-popularea unui form trebuie făcută atunci când:
    - modificăm un obiect deja existent
    - am trimis deja date către server, dar datele nu au fost validate, iar form-ul trebuie recompletat.

Forms

- Clasa **Form** face parte din **django.forms**. Pentru definirea unui form vom folosi trebuie să creăm o clasă care să moștenească clasa Form:

```python
from django import forms


class RegisterForm(forms.Form):
    pass
```

- Rolul Django este de a ne face viața mai ușoară, așa că un form pentru înregistrarea unui utilizator poate fi definit astfel:

```python
from django import forms
from django.contrib.auth.password_validation import password_validators_help_text_html


class RegisterForm(forms.Form):
    first_name = forms.CharField(label='First name', max_length=255, required=True)
    last_name = forms.CharField(label='Last name', max_length=255, required=True)
    email = forms.EmailField(label='Email address', required=True)
    password = forms.CharField(
        label='Password',
        widget=forms.PasswordInput,
        required=True,
        help_text=password_validators_help_text_html()
    )
    password_confirmation = forms.CharField(
        label='Password confirmation',
        widget=forms.PasswordInput,
        required=True,
        help_text='Please confirm your password.'
    )
```

Forms

- După definirea form-ului avem nevoie să instanțiem clasa într-un view și să o trimitem către template-ul nostru.
- Trebuie definită ruta de register. Aceasta se va face în **users.urls.py**:

```
path('register/', register_view, name='register'),
```

- Trebuie definit view-ul **register_view** care va gestiona call-urile către noua rută. Aceasta se va face în **users.views.py**:

```
from django.shortcuts import render
from users.forms import RegisterForm


def register_view(request):
    if request.method == 'GET':
        form = RegisterForm()
    else:
        form = RegisterForm(request.POST)

    return render(request, 'users/register.html', {
        'form': form
    })
```

Forms

- Pentru a afișa form-ul în pagină avem nevoie să scriem template-ul **users/register.html** astfel:

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

{% block content %}
    <form method="post" action="{% url 'users:register' %}">
        {% csrf_token %}
        {{ form }}
        <input type="submit" value="Register">
    </form>
{% endblock %}
```

- Această metodă de afișare este cea mai basic și va înșirui toate câmpurile inline. Rezultatul va fi:

Forms

- Alte două rapide mai OK de afișare a form-ului o reprezintă metoda de afișare sub formă de paragrafe (HTML **p** element) sau sub formă de tabel.

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

{% block content %}
    <form method="post" action="{% url 'users:register' %}">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Register">
    </form>
{% endblock %}
```

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

{% block content %}
    <form method="post" action="{% url 'users:register' %}">
        {% csrf_token %}
        <table>
            {{ form.as_table }}
        </table>
        <input type="submit" value="Register">
    </form>
{% endblock %}
```

Forms

- Cele două metode vor fi afișate astfel:

First name:

Last name:

Email address:

Password:

- Your password can’t be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can’t be a commonly used password.
- Your password can’t be entirely numeric.

Password confirmation: Please confirm your password.

Register

{{ form.as_p }}

First name:

Last name:

Email address:

Password:

- Your password can’t be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can’t be a commonly used password.
- Your password can’t be entirely numeric.

Password confirmation: Please confirm your password.

Register

{{ form.as_table }}

Forms

- Un mod într-adevăr elegant de a reprezenta un form necesită folosirea **Crispy Forms**. Detalii despre acest pachet puteți găsi la adresa https://django-crispy-forms.readthedocs.io/en/latest/. Este un pachet Python ce are rolul să stilizeze form-urile Django.
- Pentru a îl instala în proiect folosiți **pip install django-crispy-forms**.
- Adăugați aplicația **crispy_forms** la secțiunea **INSTALLED_APPS** din **settings.py**.
- Adăugați și setarea **CRISPY_TEMPLATE_PACK** pentru a defini ce template doriți să folosiți în pagină (mai multe detalii găsiți în documentația pachetului - există suport pentru Bootstrap):

```
CRISPY_TEMPLATE_PACK = 'uni_form'
```

Forms

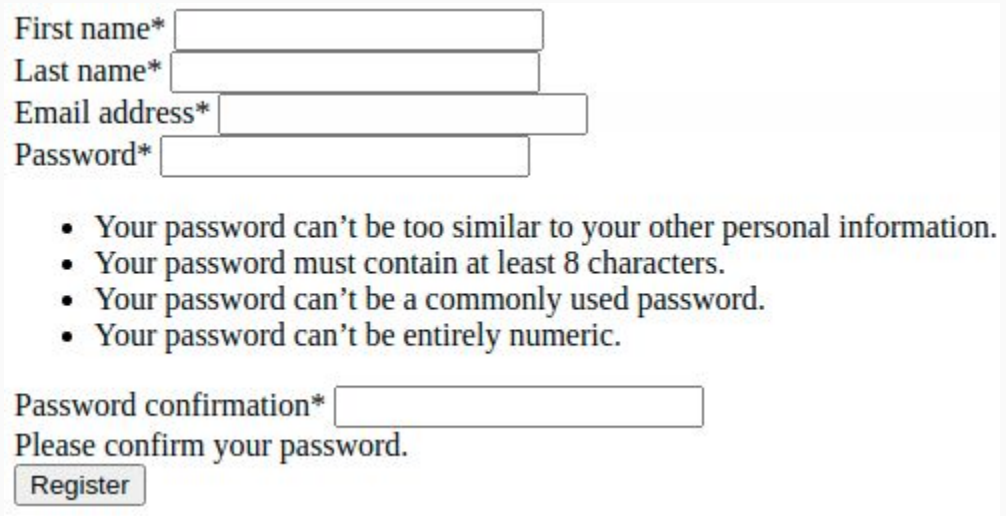
- Pentru randarea form-ului folosiți filtrul **crispy** astfel (pentru mai mult control în design-ul form-ului folosiți tag-ul **{% crispy %}**):

```
1  {% extends "base.html" %}
2  {% load crispy_forms_tags %}
3
4  {% block content %}
5      <form method="post" action="{% url 'users:register' %}">
6          {% csrf_token %}
7          {{ form | crispy }}
8          <input type="submit" value="Register">
9      </form>
10 {% endblock %}
```

- Observați importul de la L2 și folosirea filtrului crispy la L7.

Forms

- La o primă vedere rezultatul nu este unul foarte spectaculos. Avantajul oferit este că de aici mai departe form-urile pot fi stilizate foarte ușor datorită modului de randare al său.



Forms

- Un form HTML este un element folosit în colectarea informațiilor de la utilizator. Cu ajutorul lui putem obține date de la utilizator pe care le trimitem server-ului pentu procesare.
- Elementul form este un container pentru diferite tip-uri de elemente de tip input. Mai multe detalii puteți găsi pe https://www.w3schools.com/html/html_forms.asp.
- Datele dintr-un form vor fi trimise către server pe ruta definită de atributul **action** al form-ului.
- Metoda folosită, de obicei, în trimiterea datelor este metoda **POST**. Aceasta este definită de atributul **method** al form-ului.
- Pe lângă cele 2 atribute și randarea form-ului trimis din view cu oricare din metodele discutate anterior, un form trebuie să mai conțină obligatoriu:
    - tag-ul **{% csrf_token %}** - acest tag va adăuga un **input hidden** în form-ul nostru. Valoarea acestui input va fi un token ce ne protejează de **C**ross **S**ite **R**equest **F**orgery. Mai multe detalii puteți găsi pe https://docs.djangoproject.com/en/3.1/ref/csrf/.
    - un element **input** cu **type="submit"** care va face submit la form în momentul în care este apăsat (datele vor fi trimise către server).

Forms

- După trimiterea unui form, detaliile introduse de utilizator se vor găsi în atributul **POST** al parametrului **request**.

```
else:  # any other non-get methods (usually POST)
    form = RegisterForm(request.POST)

    if form.is_valid():
        form.save()

        return redirect('/')
```

- După verificarea metodei pe care se face request-ul, trebuie să instanțiem forma folosindu-ne de parametrii primiți de la utilizator.
- Urmează trei pași esențiali în ciclul de viață al oricărui form:
    - validarea form-ului. Fiecare form are o metodă **is_valid**, moștenită din clasa **django.forms.Form**.
    - salvarea informațiilor. Fiecare form trebuie să aibă o metodă save unde vom defini ce se întâmplă cu datele după validarea lor.
    - redirectarea către o nouă rută.

Forms

```python
def clean_email(self):
    email = self.cleaned_data.get('email')

    try:
        AuthUser.objects.get(email=email)
    except AuthUser.DoesNotExist:
        return email
    else:
        raise forms.ValidationError('Email is already taken.')

def clean_password(self):
    first_name = self.cleaned_data.get('first_name')
    last_name = self.cleaned_data.get('last_name')
    email = self.cleaned_data.get('email')
    password = self.cleaned_data.get('password')

    user = AuthUser(
        first_name=first_name,
        last_name=last_name,
        email=email
    )

    validate_password(password, user)

    return password

def clean_password_confirmation(self):
    password = self.cleaned_data.get('password')
    password_confirmation = self.cleaned_data.get('password_confirmation')

    if password_confirmation != password:
        raise forms.ValidationError('Password not confirmed.')

    return password_confirmation
```

- Validarea unui form se face de către Django în mai mulți pași. Unul din ei presupune normalizarea datelor. Acest pas va prelua datele din form și ni le va pune la dispoziție ca obiecte Python (ex. pentru **email** vom avea **string**-ul introdus de utilizator în loc de instanța clasei **EmailField**).
- Validarea datelor unui form ne revine nouă, ca programatori.
- Fiecare field poate fi validat individual cu ajutorul metodei **clean_fieldname**. Aceste metode pot fi scrise pentru fiecare field pe care dorim să-l validăm.
- Dacă avem nevoie să validăm form-ul pe baza a mai multor field-uri se va folosi metoda **clean**.
- Datele din form se obțin din dicționarul **cleaned_data**.

Forms

```python
def clean_email(self):
    email = self.cleaned_data.get('email')

    try:
        AuthUser.objects.get(email=email)
    except AuthUser.DoesNotExist:
        return email
    else:
        raise forms.ValidationError('Email is already taken.')

def clean_password(self):
    first_name = self.cleaned_data.get('first_name')
    last_name = self.cleaned_data.get('last_name')
    email = self.cleaned_data.get('email')
    password = self.cleaned_data.get('password')

    user = AuthUser(
        first_name=first_name,
        last_name=last_name,
        email=email
    )

    validate_password(password, user)

    return password

def clean_password_confirmation(self):
    password = self.cleaned_data.get('password')
    password_confirmation = self.cleaned_data.get('password_confirmation')

    if password_confirmation != password:
        raise forms.ValidationError('Password not confirmed.')

    return password_confirmation
```

- În exemplul alăturat aveți exemplul de validare al datelor unui form pentru înregistrarea utilizatorului.
- Field-urile care trebuie validate în acest caz vor fi:
  - email
  - password
  - password_confirmation
- Pentru validarea e-mail-ului vom căuta în baza de date un utilizator care ar putea avea asociat adresa respectivă de e-mail. Atenție că metoda get va arunca excepția **DoesNotExist** dacă nu va găsi nici o înregistrare. Astfel pe ramura except nu trebuie să facem nimic (cazul în care nu avem un utilizator cu această adresă de e-mail)
- Pentru validarea field-ului password vom avea nevoie să instanțiem clasa User pentru a putea folosi și validatorii care se referă la informațiile personale ale utilizatorului.
- Validarea parolei folosind validatorii Django se face apelând metoda **validate_password**. Aceasta se găsește în **django.contrib.auth.password_validation**
- Dacă valoarea field-ul **password_confirmation** nu este aceeași cu valoarea field-ului password, atunci putem arunca o excepție în acest sens.
- Pentru a marca o eroare într-un form trebuie aruncată **ValidationError** care se află în **django.forms**.

Forms

- Metoda **save** are rolul să creeze **user**-ul și să îl salveze în baza de date. Opțional îl putem returna în cazul în care dorim să îl folosim mai departe.

```
def save(self):
    first_name = self.cleaned_data['first_name']
    last_name = self.cleaned_data['last_name']
    email = self.cleaned_data['email']
    password = self.cleaned_data['password']

    user = AuthUser.objects.create_user(
        first_name=first_name,
        last_name=last_name,
        email=email,
        password=password
    )

    return user
```

Model forms

Forms & ModelForms

Model forms

- Model forms sunt asemănătoare cu form-urile normale, diferența fiind că acestea se bazează pe modele.
- Pentru a defini un model form avem nevoie să moștenim clasa **ModelForm** din **django.forms**.
- Folosind clasa Meta putem defini detaliile despre modelul pe care vrem să-l folosim astfel:

```
from django import forms
from django.contrib.auth import get_user_model

AuthUser = get_user_model()


class RegisterForm(forms.ModelForm):
    class Meta:
        model = AuthUser
        fields = ['first_name', 'last_name', 'email', 'password']
```

- Trebuie să definim atributul fields în care vom înșirui toate câmpurile pe care dorim să le folosim în form-ul nostru.
- O alta variantă este să folosim atributul exclude în care să înșiruim toate câmpurile pe care dorim să le excludem. Dacă nu dorim să excludem nici un câmp putem folosi `exclude = []`.

Model forms

- Pentru că avem nevoie să afișăm textul ajutător pentru setarea parolei și de câmpul de confirmare al parolei vom rescrie form-ul anterior astfel (a se observa că, pe lângă câmpurile din model putem defini și câmpuri proprii):

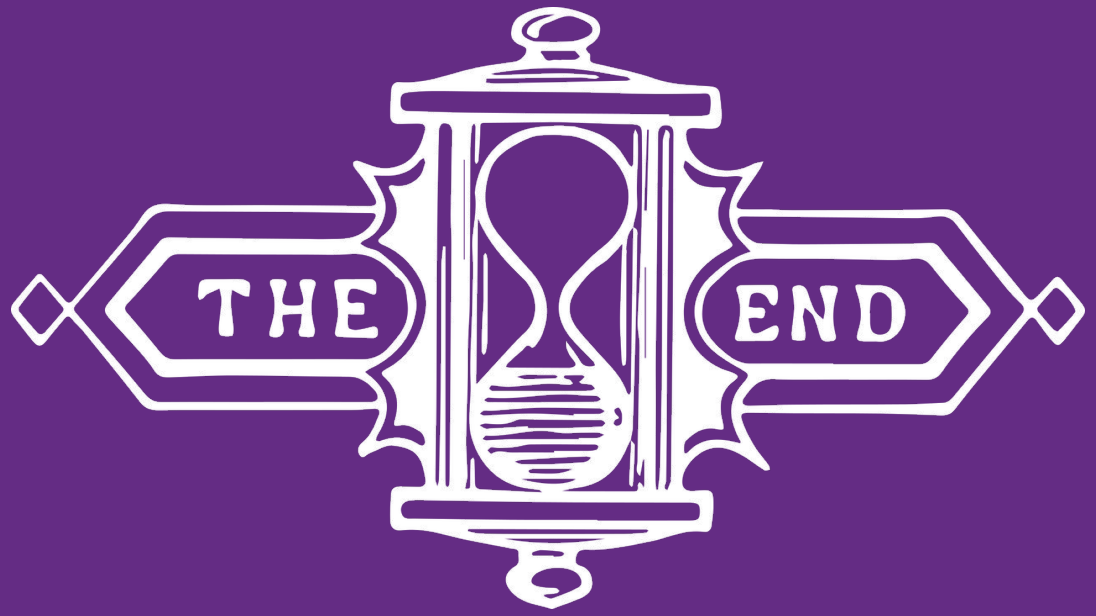
```
class RegisterForm(forms.ModelForm):
    class Meta:
        model = AuthUser
        fields = ['first_name', 'last_name', 'email']

    password = forms.CharField(
        label='Password',
        widget=forms.PasswordInput,
        max_length=255,
        required=True,
        help_text=password_validators_help_text_html()
    )

    password_confirmation = forms.CharField(
        label='Password confirmation',
        widget=forms.PasswordInput,
        max_length=255,
        required=True,
        help_text='Confirm your password'
    )
```

Model forms

- În cazul exemplului acestui form de înregistrare nu ne mai rămâne decât să validăm parola și confirmarea parolei.
- De validarea email-ului nu mai este nevoie pentru că avem deja cerință de unicitate în definirea atributului email.
- Metoda save va trebui să suprascrie metoda save a părintelui pentru că avem nevoie să setăm parola utilizatorului. Având în vedere că am folosit un field separat pentru parolă, avem nevoie să o setăm manual astfel:

```
def save(self, commit=True):
    password = self.cleaned_data.get('password')
    self.instance.set_password(password)
    return super().save(commit)
```

- Metoda save a clasei ModelForm primește un parametru default numit **commit**. Rolul său este să decidă dacă după crearea obiectului acesta va fi stocat și în baza de date sau nu.
- Valoarea predefinită a acestuia este **True**, deci la save automat modelul nostru va fi și salvat în baza de date.



Vă mulțumesc!