Django filmtoepassing

In deze toepassing zitten een aantal van de meest voorkomende technieken om een Django-toepassing te maken. Je kan ze dus gebruiken als blauwdruk voor toekomstige toepassingen. In de volgende tekst begeleid ik je stap voor stap met het opbouwen van de toepassing. Met telkens een woordje uitleg.

Wat gaan we doen? - Je ontwikkelt een filmreview-toepassing in Django. - Deze toepassing laat gebruikers toe om een filmoverzicht te bekijken en te zoeken doorheen dit overzicht - Elke film heeft een afbeelding, een titel, een beschrijving - Gebruikers hebben de mogelijkheid om in te loggen en reviews te posten (een reviewtekstje) over een film die ze hebben gezien - Bij elke film komt een lijstje van reviews te staan - Gebruikers die bij een bepaalde film een review hebben toegevoegd kunnen hun eigen review ook wijzigen en verwijderen - Vrijblijvend kan je ook een nieuwstoepassing toevoegen die filmnieuwtjes bevat en toont (titel, body, datum)

Project opstarten

- We werken in een Python virtuele omgeving
- We installeren django met pip
- We creëren ons project
- We starten de server

```
pip install django
python -m django startproject filmreviews
cd filmreviews
python manage.py runserver
```

Projectstructuur begrijpen en een toepassing aanmaken binnen het project

settings.py

Beheert de settings van je project, met ondermeer: - BASE_DIR: waar situeert het project zich (hoef je niet te wijzigen) - SECRET_KEY: gebruikt bij uitwisselen van data. Nooit delen met anderen - DEBUG: wanneer je de toepassing in productie plaatst zet je deze waarde op false - INSTALLED_APPS: oplijsten van de apps waaruit je project bestaat - MIDDLEWARE: verwijst naar ingebouwde Django functies om applicatie request/responses te verwerken. Bvb authenticatie, sessies en beveiliging. - ROOT_URLCONF: geeft aan waar de urls.py zich bevindt - TEMPLATES: beschrijft de template engine klasse, de lijst van filders waar moet gezocht worden naar templates, en specifieke template settings - AUTH_PASSWORD_VALIDATORS : geeft de mogelijkheid om mee te geven welke validaties we willen voor wachtwoorden. Bvb minimumlengte

urls.py

Ingangspunt van de toepassing. Welke views moeten worden gerenderd wanneer een bepaalde url wordt opgevraagd.

manage.py

Helpt ons administratieve functies te vervullen, zoals het aanmaken van toepassingen binnen ons project.

Een toepassing aanmaken

Elk Django-project bestaat uit één of meer apps. Deze werken samen als één webtoepassing. Netjes opsplitsen houdt je code overzichtelijk en leesbaar.

Om een app film aan te maken:

```
python manage.py startapp film
```

Vergeet niet deze app toevoegen aan INSTALLED_APPS in settings.py

Daarna kan je het project runnen:

```
python manage.py runserver
```

Django URLs beheren en linken aan views

Telkens een gebruiker een url ingeeft komt het request langs urls.py. Daar kan je paden laten matchen met views. Voor onze film app worden de views opgeslagen in /film/views.py.

We importeren de ingebouwde HttpResponse method om een response object aan de aanvrager terug te geven.

Bvb een eenvoudige home en over-pagina om te starten

urls.py:

```
from django.contrib import admin
from django.urls import path
from film import views as filmViews

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', filmViews.home),
    path('over/', filmViews.over),
]
```

/film/views.py:

```
from django.shortcuts import render
from django.http import HttpResponse

def home(request):
    return HttpResponse('<h1>Welkom op de homepagina</h1>')

def over(request):
    return HttpResponse('<h1>Welkom op de over-ons pagina</h1>')
```

HTML-pagina's genereren met templates

Met templates scheiden we HTML-code van onze Django views. We plaatsen de HTML-code dan in hun eigen bestanden. Hiervoor maak je in de film-folder zelf een folder templates aan. Het is een patroon dat gebruikelijk is voor elke toepassing die je aan een project toevoegt.

Je kan zelf eens proberen: maak een html-bestand aan met een standaardstructuur en wat elementen, home.html en plaats het in de templates-folder.

In /film/views.py kan je er als volgt naar linken:

```
def home(request):
    return render(request, 'home.html')
```

Data meegeven aan de templates

```
def home(request):
    return render(request, 'home.html', {'naam':'Kristof Michiels'})
```

We geven een dictionary mee met een key-value-paar. In home.html kunnen we er als volgt gebruik van maken:

```
<h1>Welkom op de homepage, {{ naam }}</h1>
```

We gebruiken hier DTL, de Django Template Language. Hier de documentatie: https://docs.djangoproject.com/en/4.1/ref/templates/language/.

Een zoekformulier toevoegen

We voegen een zoekformulier toe aan onze home.html template pagina.

```
<form action="">
          <label for="film">Zoeken naar een film:</label>
          <input type="text" name="zoekFilm" id="film"/>
          <button type="submit">Zoek film</button>
</form>
<p>Zoeken naar {{ zoekTerm }}</p>
```

In /film/views.py:

```
def home(request):
    zoekTerm = request.GET.get('zoekFilm')
    return render(request, 'home.html', {'zoekTerm':zoekTerm})
```

Standaard een GET-request voor een form. Een POST is ook mogelijk. Het verschil: informatie wordt hier meegegeven via de URL.

De form wordt hier verwerkt op dezelfde pagina, maar het is ook mogelijk de form naar een andere url te sturen voor verwerking:

```
<h2>Schrijf je in voor de nieuwsbrief:</h2>
<form action="{% url 'schrijfin' %}">
    <label for="email">Geef je email adres:</label>
    <input type="email" id="email" name="email" />
    <button type="submit" class="btn btn-primary">Schrijf </button>
</form>
```

Het action-attribuut van de form staat gericht op de 'schrijfin'-route. We maken dit mogelijk als volgt in de urls.py:

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', filmViews.home, name='home'),
    path('over/', filmViews.over, name='over'),
    path('schrijfin/', filmViews.schrijfin, name='schrijfin'),
]
```

We lossen dit als volgt op in views.py:

```
def schrijfin(request):
    email = request.GET.get('email')
    return render(request, 'schrijfin.html', {'email':email})
```

Dit impliceert een template schrijfin.html met daarin het volgende:

```
<h2>{{ email }} toegevoegd aan de mailing list</h2>
<a href="{% url 'home' %}">Home</a>
```

Merk tot slot nog op dat we met gebruik van home als name een backlink hebben gemaakt.

Werken met models

Django praat voor ons met de databank en doet dit aan de hand van models. Elk model is een klasse die erft van django.db.models.Model. Elk attribuut vertegenwoordigt een kolom in een databank tabel. We maken een eerste model aan in /film/models.py:

```
from django.db import models

class Film(models.Model):
    titel = models.CharField(max_length=100)
    beschrijving = models.CharField(max_length=250)
    afbeelding = models.ImageField(upload_to='film/afbeeldingen/')
    url = models.URLField(blank=True)
```

De model-field documentatie vind je hier: https://docs.djangoproject.com/en/4.1/ref/models/fields/

We gebruiken voor de afbeelding de upload_to option om een subfolder van de MEDIA_ROOT te gebruiken voor de opgeladen foto's. Omdat we afbeeldingen gebruiken moeten we Pillow (https://pypi.org/project/Pillow/) toevoegen aan ons project. Pillow voegt afbeeldingsverwerkingscapaciteiten toe aan de Python interpreter.

```
pip install pillow
```

Migraties laten ons toe om een databank-schema te creëren gebaseerd op model-code. Telkens we wijzigingen aanbrengen aan onze modellen moeten we nieuwe migraties creëren. Deze zal een databank creëren in ons db.sqlite3 bestand dat bij het aanmaken van ons project werd toegevoegd.

```
python manage.py makemigrations
python manage.py migrate
```

Het migrate commando voert migraties uit van alle toepassingen in de INSTALLED_APPS list. Kijk zeker eens naar het migratie-bestand in de migrations-folder van de film-app! Telkens je wijzigingen aanbrengt aan je models dien je de bovenstaande 2 commando's uit te voeren.

We gaan nu even kijken naar de Django admin interface. Django geeft je quasi automatisch een uiterst krachtige admin interface waarmee je je model data kunt beheren. Vooraleer in te loggen met de eerste gebruiker moet je een superuser aanmaken. Ik creëerde een gebruiker test met wachtwoord test. Je kan het wachtwoord later wijzigen via de admin interface, of via het commando

```
python manage.py createsuperuser
python manage.py changepassword <username>
```

Om ons film model toe te voegen aan de admin interface moeten we nog een wijziging aanbrengen in het film/admin.py bestand:

```
from django.contrib import admin
from .models import Film
admin.site.register(Film)
```

We willen onze gebruikers in staat stellen om afbeeldingen op te laden, dus moeten we nog enkele wijzigingen maken in filmreview/settings:

```
import os # bovenaan toevoegen
...
MEDIA_ROOT = os.path.join(BASE_DIR,'media') # beide regels onderaan de pagina
MEDIA_URL = '/media/'
```

MEDIA_ROOT is het absolute bestandssysteempad naar de folder dat bestanden zal bevatten die door de gebruikers werden opgeladen. En we joinen BASE_DIR met "media". MEDIA_URL is de url die media zal behandelen uit MEDIA_ROOT. Lees hier meer over de setting properties: https://docs.djangoproject.com/en/4.1/ref/settings/. Afbeeldingen van het film model zullen opgeslagen worden in /filmreviews/media.

Om afbeeldingen te kunnen tonen moeten we aan /filmreviews/urls.py het volgende toevoegen:

```
from django.conf.urls.static import static
from django.conf import settings

urlpatterns = [
...
]
urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

Db-informatie tonen op de site

We gaan kijken hoe we film-info kunnen tonen. Aan film/views.py voegen we het volgende toe:

```
from .models import Film  # het film-model importeren

def home(request):
    zoekTerm = request.GET.get('zoekFilm')
    films = Film.objects.all()  # alle filmobjecten uit de db halen en toekennen aan films
    return render(request, 'home.html', {'zoekTerm':zoekTerm, 'films': films})
```

Lijst met beschikbare database methods: : https://docs.djangoproject.com/en/4.1/topics/db/ queries/

In templates/home.html:

```
      {% for film in films %}
        <h2>{{ film.titel }}</h2>
        <h3>{{ film.beschrijving }}</h3>
        <img src="{{ film.afbeelding.url }}">
        {% if film.url %}
            <a href="{{ film.url }}">Film Link</a>
        {% endif %}
      {% endfor %}
```

Uitleg over hoe template tags werken en welke de mogelijkheden zijn:
https://docs.djangoproject.com/en/4.1/ref/templates/language/#tags

We gaan vervolgens de zoekfunctionaliteit doen werken door def home aan te passen in film/views.py:

```
def home(request):
    zoekTerm = request.GET.get('zoekFilm')
    if zoekTerm:
        films = Film.objects.filter(titel__icontains=zoekTerm)
    else:
        films = Film.objects.all()
    return render(request, 'home.html', {'zoekTerm':zoekTerm, 'films': films})
```

We gaan nu nog de app met nieuwsjes ontwikkelen. We gaan deze als een tweede toepassing toevoegen aan het project. Dit onderdeel zal ons veel leren over het werken in Django met meerdere toepassingen.

```
python manage.py startapp nieuws
```

Allereerst voeg je de nieuws app toe aan filmreviews/settings.py.

Ik ga de routes voor de nieuws app niet rechtstreeks toevoegen aan filmreviews/urls.py. Bij grote projecten riskeert dit bestand nogal omvangrijk te worden. Daarom werken we beter met urls.py-bestanden per toepassing en "includen" we deze aan onze centrale urls.py. Dit patroon verdient de voorkeur in onze projecten!

In filmreviews.py voeg je het volgende toe:

```
from django.urls import path, include  # bovenaan toevoegen
urlpatterns = [
    ...
    path('nieuws/', include('nieuws.urls')),
]
```

Je maakt dan zelf binnen nieuws/ een bestand urls.py aan, met daarin:

```
from django.urls import path
from . import views
urlpatterns = [
path('', views.nieuws, name='nieuws'),
]
```

In nieuws/views.py:

```
from django.shortcuts import render

def nieuws(request):
    return render(request, 'nieuws.html')
```

In /nieuws/ moeten we dan ook nog een templates folder aanmaken en we creëren er al een nieuws.html-bestand. We zullen dit bestand straks van informatie voorzien. Eerst gaan we het nieuws model invullen!

In /nieuws/models.py:

```
from django.db import models

class Nieuws(models.Model):
    koptekst = models.CharField(max_length=200)
    body = models.TextField()
    datum = models.DateField()
```

Niet vergeten in de terminal volgende commando's uit te voeren:

```
python manage.py makemigrations
python manage.py migrate
```

Ook moeten we het nieuws model nog toevoegen aan nieuws/admin.py

```
from django.contrib import admin
from .models import Nieuws

admin.site.register(Nieuws)
```

We moeten tot slot nog zorgen dat we nieuwsberichten kunnen tonen via nieuws.html.

in nieuws/views.py:

```
from django.shortcuts import render
from .models import Nieuws

def nieuws(request):
    nieuwsberichten = Nieuws.objects.all()
    return render(request, 'nieuws.html', {'nieuwsberichten':nieuwsberichten})
```

in nieuws/templates/nieuws.html:

```
{% for nieuwsbericht in nieuwsberichten %}
        <h2>{{ nieuwsbericht.koptekst }}</h2>
        <h5>{{ nieuwsbericht.datum }}</h5>
        <p>{{ nieuwsbericht.body }}</p>
{% endfor %}
```

Maak enkele nieuwsberichten aan in de admin en bekijk ze in de browser.

Om de nieuwsberichten chronologisch weer te geven met het nieuwste bericht bovenaan kunnen we nieuwsobjecten ordenen met order_by:

```
nieuwsberichten = Nieuws.objects.all().order_by('-datum')
```

De databank voor onze toepassing

Django gebruikt standaard SQLite. Een goeie keuze voor kleine projecten, maar weet je gemakkelijk kan overschakelen naar andere DBs.

Je kan de inhoud van de database bekijken bvb DB browser for SQLite.

De tabelnamen zijn afgeleid van _. We kunnen dus meerdere modellen hebben binnen één en dezelfde toepassing. Bvb. film_film en film_review.

Elke rij in de tabel heeft een automatische door Django gegenereerde primary key.

Onze Admin een beetje tweaken:

Op dit moment: nieuws object 1, nieuws object 2, enz... Voor betere leesbaarheid, in nieuws/models.py, binnen de klasse Nieuws:

```
def __str__(self):
    return self.koptekst
```

De **str** method in Python vertegenwoordigt de klasse objecten als string. **str** zal worden aangeroepen wanneer nieuws objecten opgelijst staan in admin.

Templates extenden

We maken een "globale" template waarvan al onze bestaande templates gebruik kunnen maken. Omdat de template globaal is zullen we ze toevoegen aan de folder filmreviews. Daarom creëren we daar een folder templates met daarin een bestand, base.html. Aan dit bestand voegen we alle gemeenschappelijke html toe.

We starten filmreviews/templates/base.html met:

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

Binnen deze blok komt de code van de child-pagina's.

We moeten ook nog de filmreviews/templates folder registreren door ze toe te voegen aan de TEMPLATES DIRS in /filmreviews/settings.py:

```
...
TEMPLATES = [
{
'BACKEND': 'django.template.backends.django
.DjangoTemplates',
'DIRS': [os.path.join(BASE_DIR,
'filmreviews/templates')],
'APP_DIRS': True,
...
```

film/templates/home.html passen we aan tot:

```
{% extends 'base.html' %}
{% block content %}
...
{% endblock content %}
```

De links toevoegen aan base.html:

```
<a href="{% url 'home' %}">Films</a>
<a href="{% url 'nieuws' %}">Nieuws</a>
```

Statische bestanden

Django kan ook statische bestanden meegeven naar de browser. Statische afbeeldingen, video's, maar ook css en JavaScript.

Hoe doe je dit? In /filmreviews/settings.py hebben we een STATIC_URL = '/static/' property met daarboven in comment een link naar de documentatie: https://docs.djangoproject.com/en/4.0/howto/static-files/

Onderaan het /filmreviews/settings.py bestand voeg je nog volgende code toe:

```
STATICFILES_DIRS = [
BASE_DIR / 'filmreviews/static/',
]
```

In /filmreviews maak je een folder 'static'. Erin kan je een folder aanmaken voor afbeeldingen, bvb 'img'. Ik heb er een afbeelding ingestopt en wens die te tonen via base.html. In de head tag heb ik ook een link naar een css-bestand toegevoegd. Dat doe je zo:

```
{% load static %}
...
<link rel="stylesheet" href="{% static 'css/stijl.css' %}">
...
<img src="{% static 'img/movieclapper.webp' %}"alt="" width="50" height="50"/>
```

Een filmdetail pagina aanmaken

Hier toon ik hoe je een detailpagina aanmaakt voor films. Allereerst voegen we een route toe aan /filmreviews/urls.py. Hiermee geven we aan dat alle paden van de film app in een eigen urls.py zullen gestopt worden:

```
urlpatterns = [
    ...
    path('film/', include('film.urls')),
]
```

In /film/urls.py voegen we de eigenlijke route toe:

```
from django.urls import path
from . import views

urlpatterns = [
    path('<int:film_id>', views.detail, name='detail'),
]
```

Vervolgens, in /film/views.py gaan we de def detail view toevoegen:

```
from django.shortcuts import render
from django.http import HttpResponse

from django.shortcuts import get_object_or_404
from .models import Film

...
def detail(request, film_id):
    film = get_object_or_404(Film,pk=film_id)
    return render(request, 'detail.html', {'film':film})
```

We gebruiken get_object_or_404 om ofwel het object te krijgen, en indien de id niet gevonden is, een 404-pagina.

Indien we film-object hebben, geven we het door aan detail.html. We gaan hiervoor in /film/templates/ dit bestand aanmaken:

```
<img src="{{ film.image.url }}" class="" alt="">
{{ film.titel }}
{% if film.url %}
    <a href="{{ film.url }}" class="">
        Film Link
    </a>
{% endif %}
```

We moeten op de film-overzichtpagina (/film/templates/home.html) nog een link toevoegen naar de detailpagina:

```
<a href="{% url 'detail' film.id %}">{{film.titel }}</a>
```

Gebruikers laten registreren en inloggen

Eerst maken we een registratieformulier. We gaan deze functionaliteit onderbrengen in een eigen toepassing, accounts. Maak de app aan (zie onder) en voeg accounts toe aan INSTALLED_APPS in settings.py.

```
python manage.py startapp accounts
```

Zoals bij 'nieuws' en 'film' zorgen we ook bij 'accounts' voor een eigen urls.py. Maak dit bestand aan in de folder accounts en voeg onderstaande regel toe aan de project-urls.py:

```
path('accounts/', include('accounts.urls')),
```

In /accounts/urls.py:

```
from django.urls import path
from . import views

urlpatterns = [
    path('registreer/', views.registreer, name='registreer'),
]
```

Vervolgens maak je een def registreer aan in /accounts/views.py:

```
from django.shortcuts import render
from django.contrib.auth.forms import UserCreationForm

def registreer(request):
    return render(request, 'registreer.html', {'form':UserCreationForm})
```

We importeren hier UserCreationForm. Django voorziet dit om eenvoudig een registratie-formulier aan te kunnen maken voor nieuwe gebruikers. We geven deze form mee aan onze registreer-functie.

Vervolgens maak je een /accounts/templates/registreer.html aan. Deze vul je met:

```
{% extends 'base.html' %}
{% block content %}
    <form method="POST">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Registreer</button>
    </form>
{% endblock content %}
```

Enkele opmerkingen: - Form heeft geen action attribuut, dus submit is gericht op dezelfde pagina - De form method is POST, de informatie die wordt meegestuurd is dus niet zichtbaar in de url - Django voorziet {% csrf_token %} om onze forms te beschermen tegen cross-site request forgery (CSRF) aanvallen. Gebruik het steeds - We gebruiken {{ form.as_p }} = Django's UserCreationForm, hier met de .as_p-method die de form elementen voorziet van p-tags

Nu gaan we ervoor zorgen dat een gebruiker wordt aangemaakt als de form correct wordt gesubmit.In /accounts/views.py passen we de functie registreer aan:

```
from django.shortcuts import render
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User  # Het Django User model
from django.contrib.auth import login  # De login functie
from django.shortcuts import redirect  # De redirect functie

def registreer(request):
    if request.method == 'GET':
        return render(request, 'registreer.html', {'form':UserCreationForm})
    else:
        if request.POST['password1'] == request.POST['password2']:
            user = User.objects.create_user(
                request.POST['username'],
                password= request.POST['password1'])
            user.save()
            login(request, user)
            return redirect('home')
```

Enkele opmerkingen: - We checken eerst of het om GET of POST gaat: hiermee weten we of we de form mogen tonen, of dat we de gebruiken mogen aanmaken - Vooraleer de gebruiker aan te maken checken we of beide wachtwoorden matchen - Het User model zit ingebouwd in Django en wordt voorzien door de Django Auth app (zie de betreffende import) - In de admin interface hebben we gebruikers. Bevat o.m. de superuser account die we eerder creëerden - user.save() slaat de nieuwe gebruiker op in de database. We gaan deze ook terugvinden in de admin interface - Tot slot loggen we de gebruiker in en redirecten we de gebruiker naar de homepage

We gaan nu proberen nog beter in te spelen op problemen die kunnen ontstaan bij het creëren van een account

Actie ondernemen indien de ingevoerde wachtwoorden niet matchen. Hiervoor voegen we een else blok toe in /accounts/views.py:

```
if request.POST['password1'] == request.POST['password2']:
    ...
else:
    return render(request, 'registreer.html', {'form':UserCreationForm, 'error':' Wachtwoorden matchen
```

We geven hier een foutboodschap mee, maar moeten die nog gebruiken in /accounts/templates/registreer.html:

```
{% if error %}
<div class="foutmelding" role="alert">
    {{ error }}
</div>
{% endif %}
```

Nagaan of een gebruikersnaam reeds bestaat

Er zijn fouten die enkel geïdentificeerd worden vanuit een databank - bvb als een gebruiker zich registreert met een username die reeds benomen is. Zo'n fout die gecapteerd wordt door de databank kan gedetecteerd worden met een try/except.

Voeg deze code toe aan /accounts/views.py:

```
from django.db import IntegrityError
...
try:
...
except IntegrityError:
    return render(request, 'registreer.html', {'form':UserCreationForm, 'error':'Gebruikersnaam is ree
```

We importeren IntegrityError en gebruiken een try-except blok om deze error te onderscheppen wanneer ze zich voordoet.

Aanpassen van de UserCreationForm

Deze form is in grote mate aanpasbaar. Wij gaan ons hier beperken tot het verwijderen van de standaard helptekst. We moeten hiervoor een nieuwe klasse aanmaken die erft van UserCreationForm. We creëren hiervoor, in /accounts/ een bestand forms.py. Dit is gebruikelijk in Django voor werken met forms.

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

class GebruikerRegistratieForm(UserCreationForm):
    def __init__(self, *args, **kwargs):
        super(GebruikerRegistratieForm, self).__init__(*args,**kwargs)
        for fieldname in ['username', 'password1','password2']:
            self.fields[fieldname].help_text = None
            self.fields[fieldname].widget.attrs.update({'class': 'form-klasse'})
```

- We maken dus een nieuwe form die erft van UserCreationForm
- We zetten de help_text van elk form element op None om ze te verwijderen
- We geven een klassenaam aan elk form element, gewoon om te tonen dat dat kan ;-)

Nu moeten we nog zorgen dat we deze form kunnen gebruiken. We doen dit door een aanpassing in /accounts/views.py:

```
...
from django.contrib.auth.forms import UserCreationForm
from .forms import GebruikerRegistratieForm
...
# we vervangen op drie plaatsen UserCreationForm door GebruikersRegistratieForm
```

Je zal zien dat de voordien getoonde helptekst nu niet meer aanwezig is.

Laten zien dat een gebruiker al dan niet ingelogd is

We willen login- en registreer-functionaliteit tonen voor niet-ingelogde gebruikers, en een logout knop voor ingelogde gebruikers.

In filmreviews/templates/base.html voeg je de volgende code toe:

```
{% if user.is_authenticated %}
    <a href="#">
        Uitloggen ({{ user.username }})
    </a>
{% else %}
    <a href="#">Login</a>
    <a href="#">Sign Up</a>
{% endif %}
```

Je ziet me hier gebruik maken van een user object dat ons standaard wordt voorzien door Django. Dit object bevat username, password, email, first_name, en last_ name properties. We kunnen daarnaast ook nagaan als een gebruiker is ingelogd met {% if user.is_authenticated %}. Als een gebruiker geauthenticeerd is, renderen we een uitlogknop met de overeenkomstige gebruikersnaam {{ user.username }}. Anders betekent het dat de gebruiker niet is ingelogd. In dat geval tonen we de login en de registreer-knoppen.

De loguit-functionaliteit toevoegen

In /accounts/urls.py het volgende toevoegen:

```
urlpatterns = [
    ...
    path('loguit/', views.loguit, name='loguit'),
]
```

In /accounts/views.py het volgende toevoegen:

```
...
from django.contrib.auth import login, logout
...
def loguit(request):
    logout(request)
    return redirect('home')
```

We roepen de logout functie aan die Django ons biedt en leiden af naar de homepagina.

In filmreviews/templates/base.html, voeg je tot slot het volgende toe:

```
...
<a href="{% url 'loguit' %}">Uitloggen ({{ user.username }})</a>
...
<a href="{% url 'registreer' %}">Registreer</a>
...
```

De login-functionaliteit toevoegen

We creëren een login-route in /accounts/urls.py:

```
path('login/', views.loginaccount, name='login'),
```

In /accounts/views.py het volgende toevoegen:

```
...
from django.contrib.auth.forms import AuthenticationForm
from django.contrib.auth import login, logout, authenticate
...
def loginaccount(request):
    if request.method == 'GET':
        return render(request, 'login.html', {'form':AuthenticationForm})
    else:
        user = authenticate(request,
        username=request.POST['username'],
        password=request.POST['password'])
        if user is None:
            return render(request,'login.html',{'form': AuthenticationForm(),'error': 'Gebruikersnaam
        else:
            login(request,user)
            return redirect('home')
```

We maken een nieuw bestand aan, accounts/templates/login.html, dat dezelfde logica zal volgen als registreer.html. Ik heb enkel 'registreer' vervangen door 'login'.

Tot slot passen we filmreviews/templates/base.html aan. We vervangen hier het href-attribuut:

```
<a href="{% url 'login' %}">Login</a>
```

Ingelogde gebruikers toelaten reviews te schrijven, lezen, aanpassen en verwijderen

Gebruikers een review laten posten

We gaan deze functionaliteit toevoegen binnen de film app. Aan /film/models.py voeg je een tweede model en een extra import statement toe:

```
from django.contrib.auth.models import User
...
class Review(models.Model):
    tekst = models.CharField(max_length=100)
    datum = models.DateTimeField(auto_now_add=True)
    user = models.ForeignKey(User,on_delete=models.CASCADE)
    film = models.ForeignKey(Film,on_delete=models.CASCADE)
    opnieuwBekijken = models.BooleanField()

    def __str__(self):
        return self.tekst
```

- Het veld 'tekst' bevat de tekst van het review

- Voor de datum kiezen we voor auto_now_add=True. Dit betekent dat bij aanmaak van een object automatisch de datetime van dat moment zal ingevuld worden. Je kan het veld hierdoor wel niet wijzigen (maar dat is niet nodig in dit geval)
- Voor de velden 'user' en 'film' gebruiken we ForeignKey die een many-to-one relatie mogelijk maken met deze tabellen. Dit betekent dat een gebruiker meerdere reviews kan aanmaken, én dat een film meerdere reviews kan hebben
- Met 'user' bedoelen we uiteraard het ingebouwde User model dat Django voor ons voor authenticatie voorziet
- Voor alle many-to-one relaties zoals ForeignKey moeten we ook een on_delete optie meegeven. Wat moet er gebeuren als we een gebruiker of een film verwijderen? Hier kiezen we ervoor om de review dan ook te verwijderen
- Met de Boolean opnieuwBekijken geven we de gebruiker de mogelijkheid om aan te geven of hij of zij de film ooit opnieuw zal bekijken

Om ons nieuw model te doen verschijnen in het admin dashboard pas je /film/admin.py als volgt aan:

```
from django.contrib import admin
from .models import Film, Review

admin.site.register(Film)
admin.site.register(Review)
```

Om ons nieuw model aan te maken in de DB:

```
python manage.py makemigrations
python manage.py migrate
```

Een review aanmaken in de DB

Deze blauwdruk laat zien hoe we gebruikers met modellen en objecten kunnen laten interageren zonder gebruik te maken van de standaard admin interface. We kunnen deze interactie zonder de minste problemen verweven met onze eigen applicatie en interface. Ook hier biedt Django de nodige ondersteuning.

In /film/urls.py voeg je volgend path toe:

```
path('<int:film_id>/maakreview', views.maakreview, name='maakreview'),
```

In /film/views.py voeg je def maakreview toe en vervang/voeg je het volgende toe aan de import statements:

```
...
from django.shortcuts import get_object_or_404, redirect
from .models import Film, Review
from .forms import ReviewForm
...
def maakreview(request, film_id):
    film = get_object_or_404(Film,pk=film_id)
    if request.method == 'GET':
        return render(request, 'maakreview.html', {'form':ReviewForm(), 'film': film})
    else:
        try:
            form = ReviewForm(request.POST)
            nieuwReview = form.save(commit=False)
            nieuwReview.user = request.user
            nieuwReview.film = film
            nieuwReview.save()
            return redirect('detail', nieuwReview.film.id)
        except ValueError:
            return render(request, 'maakreview.html', {'form':ReviewForm(),'error':'Niet correcte data
```

- We halen eerst het film object uit de databank. Gebruiken we een niet-bestaande id dan tonen we een 404-pagina. We doen hiervoor beroep op de Django get_object_or_404 shortcut
- Wanneer we een GET request binnenkrijgen dan betekent dit dat een gebruiker een review wil aanmaken. In dat geval tonen we maakreview.html met het review formulier zodat de gebruiker een review kan aanmaken
- Wanneer de gebruiker de form ingevuld terugstuurt zal deze functie een POST request ontvangen. Dan wordt de else uitgevoerd. We vangen eerst de form-waarden op en maken en saven een nieuw review object. We slaan nog niet op in de DB (commit=False) omdat we eerst nog gebruiker en film willen meegeven. Daarna slaan we wel op
- Tot slot leiden we de gebruiker weg naar de detailpagina van de film. Als er een exception gebeurt op basis van de ingevoerde data tonen we opnieuw maakreview.html en voegen we daar een foutmelding aan toe

Nu gaan we /film/templates/maakreview.html aanmaken, met volgende code:

```
{% extends 'base.html' %}
{% block content %}
<h2>Review toevoegen voor {{ film.titel }}</h2>
{% if error %}
<div role="alert">
    {{ error }}
</div>
{% endif %}
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Review toevoegen</button>
</form>
{% endblock content %}
```

Om de review form te maken kunnen we gebruik maken van Django's ModelForm om automatisch forms op basis van models te maken. In /film maak je een bestand forms.py aan:

```
from django.forms import ModelForm, Textarea
from .models import Review

class ReviewForm(ModelForm):
    def __init__(self, *args, **kwargs):
        super(ModelForm, self).__init__(*args,**kwargs)

    class Meta:
        model = Review
        fields = ['tekst','opnieuwBekijken']
        labels = {'opnieuwBekijken': ('Opnieuw bekijken?')}
        widgets = {'tekst': Textarea(attrs={'rows': 8}),}
```

- We laten ReviewForm erven van ModelForm
- We geven mee dat de form is gebouwd voor het Review model
- We geven ook de velden mee die we in de form wensen te zien (we hebben slechts 2 velden nodig)
- We beschikken over een labels object waarmee we custom labels kunnen aanmaken voor onze velden
- Met het widgets object geven we aan dat we een textarea willen gebruikt zien voor 'tekst'
- Meer informatie over ModelForm vind je hier: https://docs.djangoproject.com/en/4.1/ topics/forms/modelforms/

We moeten ook nog een 'Voeg review toe'-knop toevoegen op de film detail pagina (/film/templates/detail.html):

```
{% if user.is_authenticated %}
    <a href="{% url 'maakreview' film.id %}">
        Voeg review toe
    </a>
{% endif %}
```

Niet-ingelogde gebruikers krijgen deze link niet te zien. Eens een review toegevoegd is dit voor de superuser ook zichtbaar in het admin panel.

Toegevoegde reviews oplijsten

In /film/views.py, maak ik volgende aanpassing aan def detail:

```
def detail(request, film_id):
    film = get_object_or_404(Film,pk=film_id)
    reviews = Review.objects.filter(film = film)
    return render(request, 'detail.html',{'film':film, 'reviews': reviews})
```

- Ik maak gebruik van de filter functie om enkel de reviews voor een bepaalde film op te vragen
- We geven dan het film object en de reviews door aan detail.html

We gaan in /film/templates/detail.html de reviews oplijsten onderaan de informatie over de film:

```
 <h3>Reviews</h3>
<ul>
{% for review in reviews %}
<li>
    <h4>Review door {{ review.user.username }}</h4>
    <p>{{ review.datum }}</p>
    <p>{{ review.tekst }}</p>
    {% if user.is_authenticated and user == review.user %}
        <a href="#">Review aanpassen</a>
        <a href="#">Delete</a>
    {% endif %}
</li>
{% endfor %}
</ul>
```

Een gebruiker kan de eigen aangemaakte reviews wijzigen of verwijderen.

Een review wijzigen

In /film/urls.py voegen we een url-path toe:

```
 ...
path('review/<int:review_id>', views.wijzigreview, name='wijzigreview'),
...
```

Deze route neemt de review ID aan als parameter, bvb http://localhost:8000/film/review/2.

We voegen vervolgens def wijzigreview toe aan /film/views.py:

```
def wijzigreview(request, review_id):
    review = get_object_or_404(Review,pk=review_id,user=request.user)
    if request.method == 'GET':
        form = ReviewForm(instance=review)
        return render(request, 'wijzigreview.html',{'review': review,'form':form})
    else:
        try:
            form = ReviewForm(request.POST,instance=review)
            form.save()
            return redirect('detail', review.film.id)
        except ValueError:
            return render(request,'wijzigreview.html',{'review': review,'form':form,'error':'Foute dat
```

- We halen het review object op met zijn id
- We geven ook de ingelogde gebruiker mee om zeker te zijn dat andere gebruikers deze review niet kunnen aanpassen
- Als het een GET request betreft, dan tonen we ReviewForm (die we ook gebruikten voor het aanmaken van een review) zodat de gebruiker de review kan wijzigen. Deze keer geven we het review object mee zodat de bestaande waarden ingevuld staan op het formulier.
- In het else blok betreft het een POST request: hier probeert een gebruiker de update van de review te bekomen
- In dat geval vragen we de waarden van de form op en doen een form.save() om de update uit te voeren
- Vervolgens verwijzen we door naar de film detail pagina
- Als er een probleem is met de ingevoerde waarden vangen we dit op met de ValueError exception

We maken een nieuw bestand aan: /film/templates/wijzigreview.html en voegen het volgende toe:

```
{% extends 'base.html' %}
{% block content %}
<h5>Wijzig Review voor {{ review.film.titel }}</h5>
{% if error %}
    <div role="alert">{{ error }}</div>
{% endif %}
<form method="POST">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit">Review wijzigen</button>
</form>
{% endblock content %}
```

Tenslotte voegen we in /film/templates/detail.html de 'wijzigreview' url to aan de update-knop:

```
...
{% url 'wijzigreview' review.id %}
...
```

Een review verwijderen

We voegen een path toe in /film/urls.py om een review te verwijderen:

```
...
path('review/<int:review_id>/verwijder', views.verwijderreview, name='verwijderreview'),
...
```

In /film/views.py voegen we def verwijderreview toe:

```
def verwijderreview(request, review_id):
    review = get_object_or_404(Review, pk=review_id, user=request.user)
    review.delete()
    return redirect('detail', review.film.id)
```

We voegen tot slot nog de verwijderreview url toe aan /film/templates/detail.html:

```
...
{% url 'verwijderreview' review.id %}
```

Authorisatie toevoegen

We moeten bepaalde pagina's nog beschermen tegen niet-ingelogde gebruikers. Zij mogen die pagina's niet kunnen openen indien ze de pagina's handmatig invullen in de browserbalk.

Om dit te doen importeren we en voegen we de @login_required decorator toe aan de views die we willen authoriseren. In /film/views.py:

```
...
from django.contrib.auth.decorators import login_required
...
@login_required
def maakreview(request, film_id):
...
@login_required
def wijzigreview(request, review_id):
...
@login_required
def verwijderreview(request, review_id):
```

We herhalen dit voor /accounts/views.py:

```
...
from django.contrib.auth.decorators import login_required
...
@login_required
def loguit(request):
...
```

We dienen ook nog het volgende toe te voegen op het einde van /filmreview/settings.py:

```
LOGIN_URL = 'login'
```

Dit leidt een gebruiker (die niet is ingelogd) toe naar de login-pagina wanneer een geautorizeerde pagina wordt geopend.

Bedankt om mee te volgen. We zijn klaar ;-)