UNIVERSITATEA DIN BUCURESTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
BAZE DE DATE ŞI TEHNOLOGII WEB

LUCRARE DE DIZERTAȚIE

UTILIZAREA ALGORITMILOR DE SUMARIZARE AUTOMATĂ ÎNTR-O APLICAȚIE DE PREZENTARE A ŞTIRILOR

Coordonator ştiinţific:
Conf. Dr. Popescu Marius

Student:
Făşie Ilie Mădălin

BUCUREŞTI
2018

Cuprins

1. Introducere....4

2. Tehnologii utilizate....6

2.1. Python....6
2.2. Django....6
2.3. HTML şi CSS....8
2.4. JavaScript....8
2.5. Redis....9
2.6. Celery....9
2.7. Sublime Text 3....10

3. Descrierea aplicaţiei....12

3.1. Audienţa aplicaţiei....12
3.2. Analiza functională folosind metodele SWOT şi MoSCoW....12
3.3. Scenarii de utilizare....14
3.3.1. Pagina de pornire....14
3.3.2. Pagina Blog....16
3.3.3. Paginile utilizatorului....17
3.3.4. Pagina de sumarizare a propriului text şi pagina de contact....17
3.3.5. Diagrama UML....18
4.1. Ce este MVC?....20
4.2. Introducere în Django....20
4.2.1. Structura unui proiect....20
4.2.2. Construirea view-urilor....22
4.3. Preluarea automată a ştirilor....23
4.3.1. Parsarea ştirilor....23
4.3.2. Sumarizarea ştirilor....25
4.3.3. Generarea automată de etichete....28
4.3.4. Programarea procesului....31
4.4. Analiza tehnică a aplicaţiei....33
4.4.1. Sistemul de votare....33
4.4.2. Înregistrarea şi autentificarea utilizatorilor....36
4.4.3. Paginarea şi căutarea....38
4.4.4. Pagini modale....40
4.4.5. Adăugarea imaginilor....41
4.5. Modelul bazei de date....42
4.5.1. Ce este ORM?....42
4.5.2. Vizualizarea bazei de date....44
4.5.3. Tabelele şi relaţiile dintre acestea....46
4.5.4. Modele şi migrări în Django....49
4.5.5. Alegerea sistemului de baze de date....52
4.6. Securitatea aplicaţiei....53
4.6.1. Securitatea formularelor....53
4.6.2. Alte elemente de securitate....55

5. Îmbunătăţiri ulterioare.....58

6. Concluzii.....61

Bibliografie.....63

1. Introducere

Istoria internetului începe în anii 1950 odată cu începerea dezvoltării calculatoarelor. În anii următori, începe să se contureze conceptul de „rețea pe arie lungă”, rezultând în formarea proiectului *ARPANET*, precursorul internetului. În 1969 a fost trimis primul mesaj prin această rețea. Mesajul trimis era litera “L” şi a fost transmis între Univesitatea Californiei (UCLA) şi institutul de cercetare Standford (SRI).

În anii 1970, suita de protocoale TCP/IP a devenit standard în ARPANET, iar în anii 1980 cercetătorul britanic Tim Berners-Lee a dezvoltat World Wide Web. În 1990 proiectul ARPANET a fost închis.

Din 1990 internetul continuă să se dezvolte într-un ritm accelerat, luând viață servicii precum poşta electronică (e-mail), mesageria instant, realizarea de apeluri telefonice prin internet, apeluri video, forumuri, bloguri şi multe altele. Din punct de vedere istoric, aceasta creştere este aproape instantanee.

Astăzi, internetul a devenit o parte importantă a societății noastre, permițând oricărei persoane să aibă acces la informații, comunicarea între persoane este mult mai uşoară, iar majoritatea serviciilor au un analog in mediul online.

Având toate aceste informații şi servicii la numai un click distanță, internetul poate părea copleşitor. Suntem conectați cu toată lumea, ştirile se răspândesc rapid şi, prin urmare, devine foarte greu, dacă nu chiar imposibil, să ținem pasul cu tot ce se întâmplă în jur. Totuşi, observăm că, în general, textele de dimensiuni mari pot fi reduse la numai câteva propoziții, fără a-şi pierde esența.

Pornind de la această observație, prezenta lucrare îşi propune a oferi o modalitate de agregare a articolelor diferitelor publicații de ştiri pe care, mai apoi, le sumarizează în trei paragrafe păstrând astfel doar esența textului.

Mai mult, pe lângă rezumarea articolelor, aplicația îşi doreşte crearea unei comunități de utilizatori. Într-adevăr, cei mai buni critici sunt oamenii pasionați de un anumit subiect, iar cu cât numărul lor creşte, cu atât ideea pe care vor să o transmită este mai bine conturată. Cu alte cuvinte, având o comunitate gata să ofere critici (atât pozitive cât şi negative), ne va fi de folos în încercarea de a perfecționa atât algoritmul de sumarizare cât şi alte aspecte şi funcționalități din aplicație.

În capitolele care urmează, vom analiza în detaliu această aplicație astfel că, în al doilea capitol vom discuta despre tehnologiile folosite pentru dezvoltarea acesteia. După cum vom vedea, tehnologia principala a fost Python împreună cu framework-ul web Django. Sintaxa uşor de citit şi înțeles a limbajului de programare Python combinată cu

metodele specifice aplicațiilor web şi arhitectura MVC incluse în Django, ne oferă un mediu de dezvoltare foarte stabil şi puternic.

Următorul capitol are drept scop prezentarea aplicației din punctul de vedere al unui utilizator normal. După cum vom vedea în acest capitol, aplicația deşi porneşte de la o idee aparent simplă, este, totuşi, foarte interactivă, oferind dreptul utilizatorului să voteze ştirile pe care le consideră importante, să îşi sumarizeze propriul articol care nu a fost publicat în aplicație sau să scrie propriile articole într-o secțiune separată.

Capitolul patru se va concentra pe aspectele tehnice ale lucrării. Astfel, vom vedea, pe scurt, arhitectura folosită pentru acest proiect, ne vom uita la structura unui proiect Django oarecare cu scopul de a ne familiariza cu framework-ul şi pentru a asigura o înțelegere deplină a deciziilor luate în dezvoltarea aplicației. Odată trecuți de partea introductivă a acestui capitol, vom aborda şi descrie în detaliu câteva funcționalități care stau la baza aplicației. Exemple de astfel de funcționalități pot fi algoritmul de sumarizare, algoritmul de etichetare automata, sistemul de votare al articolelor şi aşa mai departe.

În capitolul cinci va fi rezervat prezentării îmbunătățirilor care nu au putut fi prezente în versiunea aceasta a aplicației, dar pe care le vom adăuga în versiuni viitoare ale acesteia.

2. Tehnologii utilizate

2.1. Python

Python este un limbaj de programare foarte cunoscut, open-source, interpretat, creat de *Guido van Rossum* şi lansat către public în anul 1991. Acesta a fost dezvoltat cu gândul de a fi un limbaj uşor de scris şi de citit, astfel că sintaxa preferă cuvintele în locul simbolurilor, blocurile de cod sunt delimitate de indentare, finalul de linie nu necesită prezența semnului „ ; ”, tipul variabilelor este stabilit de interpretator la momentul rulării şi aşa mai departe.

Potrivit creatorului, dezvoltarea limbajului a pornit din dorința sa de a avea un „hobby” pe parcursul vacanței de Crăciun. Denumirea „Python” nu se referă la o specie de şarpe, ci la comedia britanică „Monthy Python’s Flying Circus” („Circul zburător al lui Monthy Python”).

Cu toate acestea Python reuşeste să fie un limbaj foarte complex şi capabil, suportând mai multe paradigme de programare precum programarea orientată pe obiecte, programarea funcțională, programarea procedurală sau programarea imperativă. Aplicațiile sale sunt, de asemenea, numeroase, el fiind folosit în domenii precum inteligența artificială, robotică, automatizare, jocuri, aplicații web etc.

Unul dintre punctele forte cele mai proeminente ale acestui limbaj de programare îl reprezintă dimensiunea impresionantă a librăriei standard. Fără nici un adaos, avem la dispoziție modalități de construire a interfețelor grafice (GUI), de lucru cu baze de date relaționale, de manipulare a expresiilor regulate, teste unitare, networking şi multe altele.

Important de reținut este faptul că, în momentul acesta, sunt două versiuni de Python suportate: Python 2 şi Python 3. Cu toate acestea, cele două versiuni nu sunt compatibile între ele (un program scris în Python 2, cel mai probabil nu va putea fi rulat în Python 3 fără modificări considerabile). În lucrarea de față, se va folosi Python 3 întrucât aceasta este versiunea oficială aflată în dezvoltare, suportul oficial pentru Python 2 urmând să fie întrerupt din 2020.

2.2. Django

Django este un framework web creat în 2003 şi lansat către public în 2005 construit în întregime în Python, care permite crearea de aplicații web într-un mod foarte simplu şi rapid. Este contruit în jurul arhitecturii MVC (Model-View-Controller) punând accent pe reutilizarea codului şi modularitate.

Pe lângă Django, un alt framework web pentru Python este Flask. Acesta din urmă are avantajul că oferă o libertate sporită dezvoltatorului asupra structurii aplicației pe care vrea să o dezvolte. Totuşi, acest framework este relativ nou comparativ cu Django (fiind lansat în 2010) şi din acest motiv comunitatea nu este la fel de mare. Prin urmare, vom alege Django pentru această lucrare întrucât suportul sporit din partea comunității şi utilitățile puse la dispoziție de acesta ne vor uşura considerabil munca.

Câteva site-uri cunoscute dezvoltate cu Django sunt: Mozilla, Instagram, Bitbucket, Disqus, Dropbox şi multe altele.

Un proiect dezvoltat în Django este împărțit în mai multe module de sine stătătoare, denumite „aplicații". Avantajul unei astfel de structuri este acela că se pot copia aplicații de la un proiect la altul, fără a fi nevoie de modificări suplimentare. Singurul lucru necesar fiind specificarea denumirii noii aplicații în variabila INSTALLED_APPS din fişierul settings.py al aplicației principale. Pentru a evita orice fel de confuzie, pe parcursul lucrării, prin „aplicație" ne vom referi la întregul proiect, urmând să fie specificate explicit cazurile în care ne referim la o aplicație Django.

Pentru a facilita dezvoltarea de aplicații web dinamice, Django include propriul său limbaj de template. Acest limbaj poate fi inclus în documentele HTML ale aplicației şi permite utilizarea instrucțiunilor specifice unui limbaj de programare precum *if-else, for, while* etc şi permite manipularea datelor trimise de către server direct din codul HTML al paginii. Mai mult, folosind acest limbaj, este posibilă extinderea fişierelor HTML sau includerea acestora în fişierul curent, astfel facilitând reutilizarea codului şi asigurând o consistență sporită a aplicației.

Un alt supliment oferit de Django este propriul ORM (Object-Relational Mapping) care uşurează foarte mult lucrul cu baza de date în aplicație. Un ORM permite automatizarea transferului de date între tabelele unei baze de date relaționale folosind cod Python în loc de cod SQL. Două avantaje importante ale utilizării unui ORM sunt posibilitatea de a dezvolta aplicația într-un singur limbaj de programare, fără a fi nevoie să se schimbe între Python şi SQL. Mai mult, utilizând un astfel de sistem, avem posibilitatea de a schimba baza de date foarte uşor. De exemplu, dacă vrem ca în mediul de dezvoltare să folosim SQLite, iar în mediul de producție PostgreSQL, nu trebuie decât să specificăm informațiile necesare pentru fiecare conexiune. Un alt exemplu ar fi dacă ne dorim să înlocuim sistemul de gestionare al bazei de date curent cu un alt sistem de gestionare mult mai capabil.

Deşi nomenclatura unei aplicații Django poate crea confuzie, tipul de arhitectură standard ales de framework este MVC. Astfel, pentru a lega modelele de date reprezentate prin clase Python, de baza de date relațională, se foloseşte ORM-ul prezentat anterior şi constituie „modelul” arhitecturii, procesarea cererilor HTTP este făcută de sistemul de template alcătuind astfel „view-ul”, iar fişierul *urls.py* conține toate rutele aplicației folosindu-se de expresii regulate şi alcătuind „controller-ul” aplicației.

2.3. HTML şi CSS

Bineînțeles, o aplicație web nu ar fi posibilă fără HTML şi CSS. Aceste două standarde, împreună cu JavaScript, stau la baza a tot ceea ce vedem pe web şi sunt întreținute de World Wide Web Consortium (W3C).

HTML (Hypertext Markup Language) este limbajul standard de creare a paginilor web. Prin intermediul tagurilor şi a atributelor acestora, browserul reuşeste să construiască pagina cerută.

CSS (Cascading Style Sheets) defineşte aspectul grafic al paginilor web. Deşi este folosit cel mai mult împreună cu HTML, acesta poate fi folosit şi cu orice document XML. Cele mai importante avantaje ale acestui limbaj sunt posibilitatea de a schimba aspectul mai multor elemente modificând doar câteva linii din documentul CSS şi posibilitatea de a crea o schemă de prioritate pentru toate regulile astfel încât rezultatul final să poată fi deductibil.

Deşi aceste două limbaje sunt foarte simple şi uşor de înțeles, atunci când dezvoltăm aplicații mai mari, documentele HTML şi CSS pot deveni destul de complicate. Pentru a permite dezvoltatorilor să se concentreze mai mult pe aspectele tehnice ale aplicației şi nu pe aspect, a fost dezvoltat framework-ul Bootstrap. Aflat acum la versiunea 4, Bootstrap ofera o mulțime de componente HTML şi CSS care pot fi combinate într-o mulțime de moduri diferite pentru a obține o interfață grafică plăcută şi unică cu un efort minim.

2.4. JavaScript

JavaScript este un limbaj de programare, dezvoltat în 1995, care permite crearea de pagini web dinamice. Împreună cu HTML şi CSS, JavaScript completează trioul care alcătuieşte nucleul tehnologiilor W3C.

Deşi sintaxa este simplă, JavaScript este capabil să folosească paradigme de programare precum programarea orientată pe obiecte, programarea functională, programarea imperativă şi programarea bazată pe evenimente. De asemenea, deşi denumirea ar putea sugera contrariul, JavaScript şi Java sunt două limbaje de programare total diferite.

Mai mult, deşi acest limbaj este gândit pentru a rula în partea clientului (client-side), de-a lungul timpului, odată cu creşterea popularității sale, au început să apară framework-uri care permit dezvoltarea părții serverului (server-side) folosind tot JavaScript. Exemple de astfel de framework-uri pot fi: Node.js, Angual.js, ReactJS, Meteor.js şi multe altele. Un lucru similar s-a întâmplat cu PHP.

2.5. Redis

Redis (REmote Dictionary Server) reprezintă o bază de date cheie-valoare in-memory (toate operațiile se efectuează în memorie, nu pe disk), cunoscut şi drept un server de structuri de date. Redis oferă o metodă foarte uşoară şi rapidă de a manipula structuri de date de nivel înalt precum liste, hărți (map), seturi şi seturi sortate.

Printre cele mai utilizate aplicații ale ale acestei tehnologii se numără: caching (fiind mult mai rapid decât bazele de date tradiționale sau folosirea soluției *memcached*), publish & subscribe (utilizând paradigma Publish/Subscribe, Redis permite distribuirea de date), cozi (de exemplu, cozi pentru job-uri ce rulează în background) şi multe altele.

Câteva companii cunoscute care folosesc Redis sunt Twitter care foloseşte clustere Redis pentru a stoca cele mai recente 800 de postări ale tuturor utilizatorilor, Pinterest care păstrează grafurile cu persoanele urmărite de fiecare utilizator în clustere Redis sau Github care a dezvoltat o librărie open-source bazată pe Redis cu scopul de a facilita execuția job-urilor de fundal aflate într-o coadă.

În lucrarea de față, nu vom folosi Redis în mod direct, însă vom utiliza o soluție care îl implementează.

2.6. Celery

Atunci când dezvoltăm o aplicație web, trebuie să avem întotdeauna în vedere confortul utilizatorului atunci când o foloseşte. De aceea, viteza de acces devine un

aspect foarte important. Cu toate acestea, pot exista cazuri în care un proces durează foarte mult, iar de rezultatul său depind multe alte funcționalități. Nu putem renunța nici la acest proces, nici la dorința de a avea o aplicație care răspunde rapid la orice comandă.

Un alt exemplu poate fi cazul în care avem nevoie să rulăm un anumit proces doar la o anumită oră din zi, sau odată pe oră.

Celery ne oferă o soluție pentru aceste probleme. Celery este o coadă de joburi asincrone bazată pe trimiterea de mesaje distribuite.

După un setup inițial în care vom specifica unde se găseşte fişierul *settings.py* al aplicației şi aplicațiile Django care pot folosi acest utilitar, ne îndreptăm atenția către funcția care conține procesul pe care vrem să îl rulăm asincron. Ceea ce ne dorim, de fapt, este să transformăm funcția noastră într-un task Celery. Acest lucru este posibil adaugând deasupra definiției funcției decoratorul *@task(name="task name")*, unde *name* reprezintă numele taskului.

Am rezolvat problema rulării unei funcții asincron, dar încă mai trebuie tratat cazul rulării unui task periodic. Celery ne oferă o soluție la fel de uşoară şi pentru această problemă. Tot ce trebuie să facem este să schimbăm decoratorul *@task* cu decoratorul
@periodic_task(run_every=(crontab(minute='/15')),name="get_news",ignore_result=True)*. Parametrul *run_every* primeşte un obiect *crontab* cu un interval de repetiție (în exemplul nostru, am vrut să rulăm acest task odată la fiecare 15 minute), parametrul *name* reprezintă numele taskului, iar parametrul *ignore_result* ne asigură că nu se va stoca starea taskului curent, acest parametru este util atunci când nu ne interesează valoarea returnată de taskul nostru.

Pentru a rula Celery cu un task simplu (neperiodic), folosim comanda *celery -A nume_aplicație worker -l info*, unde *nume_aplicație* reprezintă numele proiectului pentru care va fi rulat workerul Celery.

Pentru a rula Celery cu un task periodic, folosim comanda de mai sus, iar în altă linie de comandă, rulăm *celery -A nume_aplicație beat -l info*, unde *nume_aplicație* reprezintă numele proiectului. Această comandă se va asigura că fiecare „beat" rulează un task la intervalul de timp prestabilit.

2.7. Sublime Text 3

O parte importantă a dezvoltării oricărui tip de aplicație constă în alegerea mediului de dezvoltare potrivit. Pentru proiectul prezent, nu avem nevoie de mediu foarte complex întrucât vom folosi în cea mai mare parte cod Python şi HTML care au o sintaxă suficient de uşor de urmărit şi înțeles. Cu toate acestea, ne vor fi de folos funcționalități precum colorarea textului şi oferirea de sugestii.

Astfel, pornind de la aceste considerente, vom alege editorul de text *Sublime Text 3* pentru dezvoltarea aplicației propuse. Sublime Text este un editor de text care suportă o mulțime de limbaje de programare și oferă posibilitatea de auto-completare, colorare a instrucțiunilor și navigare prin structura proiectului.

3. Descrierea aplicației

3.1. Audiența aplicației

Trăim într-o societate aflată în continuă dezvoltare unde libertatea este considerată atât o necesitate cât şi un drept. Libertatea la liberă exprimare, libertatea de a alege, libertatea de a călătorii, toate acestea au devenit părți intrinseci ale societății. Cei care dețin cele mai multe informații şi cunoştințe, sunt capabil să ocupe un loc mai bun pe scara socială cu un efort mult mai redus. Informațiile ne permit să luăm deciziile optime pentru dezvoltarea noastră şi a comunității din care facem parte.

De exemplu, ştirile politice poate nu ne vor ajuta în dezvoltarea noastră profesională, dar ne pot crea o opinie despre clasa politică, iar când va veni momentul, vom putea alege persoana care ne poate reprezenta cel mai bine. O ştire despre situația financiară a unei alte țări poate nu ne afectează în mod direct, dar ne poate oferi o idee despre sustenabilitatea unor investiții în acea zonă. Astfel, putem spune că informația în sine nu deține o putere impresionantă, dar modul în care ea este folosită, poate schimba totul.

Prin urmare, lucrarea de față se adresează persoanelor care îşi doresc să fie informate despre tot ce se întâmplă în lume, fără a se pierde în detalii.

3.2. Analiza functională folosind metodele SWOT şi MoSCoW

Uneori o analiză detaliată a proiectului, poate duce la scurtarea drastică a timpului necesar dezvoltării. Fixarea punctelor cheie ale aplicației şi conturarea unui flux logic al acțiunilor, ne permite să dezvoltăm aplicația într-un mod în care toate componentele sunt bine legate între ele şi oferă o experiență intuitivă şi plăcută utilizatorului.

O primă metodă utilă pentru a analiza punctele tari şi punctele slabe ale aplicației noastre o reprezintă metoda SWOT. SWOT este un acronim pentru Strengths (puncte tari), Weaknesses (puncte slabe), Opportunities (oportunități) şi Threats (pericole).

Această metodă poate fi folosită nu numai pentru întreaga aplicație, ci şi pentru părți ale acesteia. Scopul fiind acela de a decide dacă o anumită funcționalitate merită să fie introdusă sau nu, care sunt direcțiile ce trebuie urmate pentru implementarea ei, la ce limitări ne putem aştepta şi aşa mai departe. Astfel, analiza SWOT a aplicației noastre de sumarizare a articolelor de ştiri este următoarea:

Puncte tari (S)	**Puncte slabe (W)**
• Posibilitatea de a citi doar un rezumat al articolelor fără a se pierde esența acestora; • Posibilitatea de a rezuma propriul text; • Design responsive • Posibilitatea de adaugare a propriilor articole sau de a citi articolele publicate de alți utilizatori; • Votarea articolelor considerate interesante, încurajând conținutul de calitate.	• Articolele de blog pot avea o calitate inferioară; • Algoritmul nu este perfect, astfel unele articole care conțin text provenit din legaturi externe (de exemplu, Twitter), pot distorsiona rezultatul; • Lipsa unui server pentru procesarea email-urilor, previne implementarea cu succes a unui sistem de verificare prin email.
Oportunități (O)	**Pericole (T)**
• Pot fi introduse reclame prin intermediul unor platforme cunoscute pentru a monetiza aplicația; • Lipsa unei concurențe imporante şi folosind o campanie de publicitate adecvată, ar putea obține un monopol în piață.	• Trebuie respectată politica site-urilor sursă cu privire la preluarea automată a conținutului acestora.

O altă metodă de analiză a aplicației este metoda MoSCoW. Asemeni metodei anterioare, şi de această dată avem de-a face cu un acronim: Trebuie să aibă (Must have), Ar trebui să aibă (Should have), Ar putea avea (Could have) şi Nu va avea – încă (Won't have – yet). Observăm că „o"-urile sunt adaugate doar pentru a uşura citirea.

Această metodă este folosită preponderent în arhitecturi de tip Agile pentru a stabili împreună cu toate părțile implicate în proiect, care sunt specificațiile cele mai importante, pe ce se va pune accent şi ce lucruri nu vor fi incluse în produsul final. În acest fel, ne asigurăm că dezvoltatorul (sau echipa de dezvoltatori) se concentrează pe ceea ce contează cu adevărat pentru buna funcționare a aplicației şi că se vor respecta limitările de timp impuse.

Metoda MoSCoW aplicată întregii noastre aplicații arata astfel:

Trebuie să aibă (Must have)	**Ar trebui să aibă (Should have)**
• Sistem de autentificare; • Algoritm de sumarizare ; • Algoritm de etichetare automată; • Legături către surse; • Aspect plăcut şi intuitiv; • Posibilitatea de a căuta o ştire; • Posibilitatea de a vota articolele; • Imagini asociate ştirilor.	• O pagină de blog, unde utilizatorii îşi pot exprima opinia despre evenimentele curente sau pot adauga ştiri care nu apar pe site; • O pagină de administrare a blogurilor fiecărui utilizator; • Posibilitatea de a asocia imagini articolelor publicate; • O pagină de contact.
Ar putea avea (Could have)	**Nu va avea – încă (Won't have – yet)**
• O pagină pentru sumarizarea textelor proprii; • Top cele mai votate ştiri; • Top cele mai votate articole de blog.	• Posibilitatea de a vota fără a se reîncărca pagina; • Pagină de profil pentru fiecare utilizator; • Modalitate de raportare a conținutului de calitate scăzută.

3.3. Scenarii de utilizare

3.3.1. Pagina de pornire

Poate cel mai important lucru pentru un utilizator este ca pagina web pe care a accesat-o să aibă un design plăcut, iar interacțiunile cu aceasta să fie intuitive. Proiectul de față a fost dezvoltat cu aceste caracteristici în minte.

Astfel, pagina de pornire va conține cele mai importante informații disponibile, anume lista ştirilor sumarizare, în ordine descrescătoare a datei în care a fost publicat articolul. Spunem „cele mai importante informații" deoarece aceasta este tema aplicației, iar aşteptările utilizatorului (în general) sunt de a fi la curent cu cele mai noi ştiri.

Lista articolelor va fi paginată, astfel că, la un moment dat, vor fi afişate numai zece articole pe pagină, iar pentru a afişa urmatoarele zece, este necesară schimbarea paginii (printr-un click pe numărul paginii dorit). De asemenea, pentru a da un aspect plăcut aplicației, dar şi pentru a oferi utilizatorului minumul de informații de care are nevoie pentru a decide dacă articolul respectiv este de interes sau nu, fiecare element al listei va afişa titlul, descrierea şi imaginea asociată.

Accesarea unui articol, va deschide o fereastră modală ce conține imaginea, titlul, data publicării şi rezumatul articolului, dar şi tagurile (etichetele) generate automat pentru acel articol şi opțiunile de a distribui acest rezumat, sau de a vizita pagina originală a articolului.

O altă modalitate prin care utilizatorul poate decide dacă un articol este de interes pentru el, este prin intermediul punctajelor asociate acestora. Fiecare articol va putea fi votat de către toți utilizatorii autentificați, astfel creând o imagine asupra calității subiectului tratat (tema articolului) sau a rezumatului generat.

Sistemul de votare este prezent în stânga fiecărui articol şi este constituit din două butoane: votare în sus pentru o părere pozitivă asupra articolului şi votare în jos pentru o părere negativă. Între cele două butoane va fi afişat scorul final (diferența dintre voturile pozitive şi cele negative), iar sub cele trei elemente va fi prezentat numărul total de voturi (indiferent de natura acestora).

Mai mult, înainte de lista articolelor, este afişată o secțiune ce conține cele mai votate trei articole în ultimele şapte zile. Aceste trei elemente sunt afişate în ordinea crescătoare a voturilor de la stânga la dreapta. Este important de observat totuşi că atunci când nici o ştire nu este votată, nu există o ordine prestabilită.

Ultimul element afişat pe această pagină îl reprezintă bara de navigare care conține atât legături către celelalte pagini ale aplicației, cât şi un câmp text prin intermediul căruia putem căuta anumite ştiri în aplicație în funcție de conținut (titlu, descriere, rezumat) sau de tagurile asociate acestora.

Paginile principale ce vor putea fi accesate folosind bara de navigare sunt: pagina de bloguri, pagina de contact, pagina „Summarize your text” ce permite oricărui utilizator să îşi sumarizeze propriul text şi un dropdown (meniu vertical care îşi afişează elementele doar atunci când este selectat) cu opțiuni specifice utilizatorului (dacă utilizatorul nu este autentificat, se vor afişa opțiuni de autentificare şi înregistrare, iar dacă este deja autentificat se vor afişa opțiunea de „de-autentificare” şi de a accesa pagina cu lista blogurilor publicate de acesta). De asemenea, textul elementului dropdown va afişa textul „My account” atunci când utilizatorul nu este autentificat şi „Welcome <*nume_utilizator*>” atunci când este autentificat, unde <*nume_utilizator*> este prenumele utilizatorului precizat la înregistrare.

Un ultim aspect important, îl reprezintă faptul că acest design este responsive. Cu alte cuvinte, atunci când redimensionăm fereastra browser-ului, elementele vor fi la rândul lor redimensionate astfel încât elementele să păstreze o structură uşor de urmărit. Câteva astfel de modificări reprezintă înlocuirea listei celor mai votate articole ale săptămânii cu un design de tip carusel (numai un element este afişat la orice un moment de timp, şi se poate apăsa un buton în stânga sau în dreapta pentru a afişa celelalte

articole), altă modificare o constituie bara de navigare care pentru dimensiuni mici ale ferestrei, va combina toate elementele prezente pe aceasta într-un singur meniu ascuns ce poate fi accesat prin apăsarea unui buton în partea dreaptă a bării (numele aplicației rămâne totuşi în colțul din stânga).

3.3.2. Pagina Blog

Blogurile au scopul de a crea o comunitate asociată aplicației şi de a oferi utilizatorilor dreptul la o liberă exprimare. Asemeni articolelor de ştiri de pe prima pagină, şi blogurile pot fi votate pentru a indica calitatea acestora.

Crearea unei comunități vine din dorința ca această aplicație să nu fie o simplă aplicație statică de citit articole. Dimpotrivă, atât sistemul de votare, cât şi blogurile sau funcția de sumarizare a propriului text, constituie o modalitate minimală de a transforma această aplicație, într-o platformă de împărtăşit idei.

Aşa cum se poate deduce din cele de mai sus, pentru a adăuga un blog, este necesară logarea utilizatorului. Accesarea paginii de adăugare se realizează prin apăsarea butonului „+” din colțul dreapta-sus. În acest moment, utilizatorul este rugat să introducă un titlu, o descriere şi un conținut pentru blogul pe care vrea să îl creeze. De asemenea, este posibilă adăugarea unei imagini reprezentative articolului de blog. În cazul în care nici o imagine nu este selectată, aplicația va adăuga o imagine standard cu textul „None” (inexistent) indicând absența unei imagini asociate acelui articol. Scopul acestei imagini standard este de a asigura un aspect consistent tuturor articolelor prezente pe pagină.

Mai mult, pentru că nu întotdeauna articolul scris este perfect, sau pentru că uneori, poate include greşeli atât de importante încât ar fi mai bine dacă articolul respectiv nu ar mai exista, utilizatorul va putea modifica sau şterge articolele publicate de el.

Pentru a accesa această funcționalitate, se va deschide meniul vertical din dreapta bării de navigare şi se va alege opțiunea „My blogs”. În acest moment, utilizatorul va putea vedea toate articolele care îi aparțin, iar în locul butoanelor de votare, vor fi alte două butoane cu un aspect sugestiv care indică *editarea*, respectiv *ştergerea* articolului.

Este important de remarcat totuşi că ştergerea unui articol folosind butonul din aplicație, nu va rezulta în ştergerea definitivă a acestuia din baza de date, ci va fi marcat ca „şters” oferind valoarea *False* coloanei *visible* a acelei înregistrări, iar în aplicație vor fi afişate doar articolele care au o valoare pozitivă pe acea coloană.

3.3.3. Paginile utilizatorului

Cum putem observa din prezentarea paginilor anterioare, utilizatorul joacă un rol important în designul aplicației noastre. Acest lucu se va observa pe tot parcursul lucrării.

Pentru a interacționa cu contul său, utilizatorul are la dispoziție un meniu dropdown (meniu vertical) numit „My account”. Atunci când acesta nu este autentificat în aplicație, va avea la dispoziție opțiunile „Log in” (autentificare) şi „Sing up” (înregistrare).

Pentru înregistrare sunt necesare numele de utilizator (care poate fi diferit de numele real al acestuia), numele şi prenumele reale, adresa de e-mail şi parola introdusă de două ori pentru verificarea corectitudinii acesteia.

Odată înregistrat, utilizatorul trebuie să acceseze din acelaşi meniu vertical, opțiunea de autentificare. Această opțiune necesită numele de utilizator şi parola alese la pasul anterior. De asemenea, pagina de „Log in” prezintă şi opțiunea de „Forgot my password” (mi-am uitat parola) care va trimite un link pe mail-ul utilizatorului conținând un link unic de resetare a parolei.

După autentificare, textul meniului dropdown va afişa textul „Welcome” urmat de prenumele utilizatorului. Bineînțeles, şi opțiunile din acest meniu s-au schimbat, ele devenind „My blogs” (blogurile mele) şi „Log out” (de-autentificare). Funcționalitatea opțiunii din urmă este evidentă, iar cea dintâi este în strânsă legătură cu pagina de bloguri discutată anterior.

3.3.4. Pagina de sumarizare a propriului text şi pagina de contact

Ultimele două meniuri importante ale aplicației le reprezintă paginile „Summarize your text” (sumarizați-vă propriul text) şi „Contact”. Aceste pagini nu afectează conținutul de informații prezent pe aplicație, însă oferă utilizatorului un plus de flexibilitate atunci când nu poate găsi informațiile pe care le caută.

Astfel, pagina de sumarizare a textului propriu, aşa cum şi numele o spune, oferă posibilitatea de a rezuma un text oarecare într-un numar ales de propoziții. Pagina conține două câmpuri, unul numeric corespunzător numărului de propoziții care vor alcătui rezumatul (în mod implicit sunt alese trei propoziții) şi unul text ce va conține textul care va fi sumarizat. Odată introduse aceste informații, prin apăsarea butonului

„Make the summary” (efectuează rezumatul), se va afişa dedesupt, rezumatul textului introdus.

Pagina de contact, oferă posibilitatea utilizatorului de a ne trimite un mesaj. Această opțiune poate fi benefică proiectului pentru că putem primi păreri despre ce am putea îmbunătăți la el, ce funcționalități sunt cele mai utile, ce probleme există, bug-uri etc. Aceste păreri pot transforma această aplicație din ce credem noi că vrea utilizatorul, în ceea ce îşi doreşte cu adevărat.

De asemenea, niciuna din aceste pagini nu necesită autentificare. În pagina de contact, utilizatorul fiind rugat să îşi introducă adresa de e-mail alături de subiectul şi textul mesajului.

3.3.5. Diagrama UML

Vom încheia acest capitol prezentând diagrama UML corespunzătoare aplicației noastre (Figura 1).

Diagramele UML (Unified Modeling Language) au fost concepute în anii 1990 cu scopul de a abstractiza complexitatea unei aplicații dezvoltată folosind paradigma programării orientată pe obiecte.

Cu toate acestea, datorită versatilității acestora, domeniul de aplicabilitate a fost lărgit, astfel că, pe lângă utilitatea în domeniul IT, se remarcă şi utilizarea lor în managementul proiectelor sau al afacerilor.

Deşi se doreşte a fi un limbaj simplu, datorită tuturor îmbunătățirilor şi funcționalităților aduse de-a lungul timpului, diagramele UML pot conține foarte multe componente care, dacă nu sunt utilizate cu grijă, pot rezulta în diagrame cu o complexitate mult prea mare pentru a fi considerate practice.

În cazul diagramei noastre am ales să nu utilizăm o structură riguroasă a acestui limbaj, ci mai degrabă, ne-am limitat la a descrie „mersul” unui utilizator prin funcționalitățile aplicației noastre.

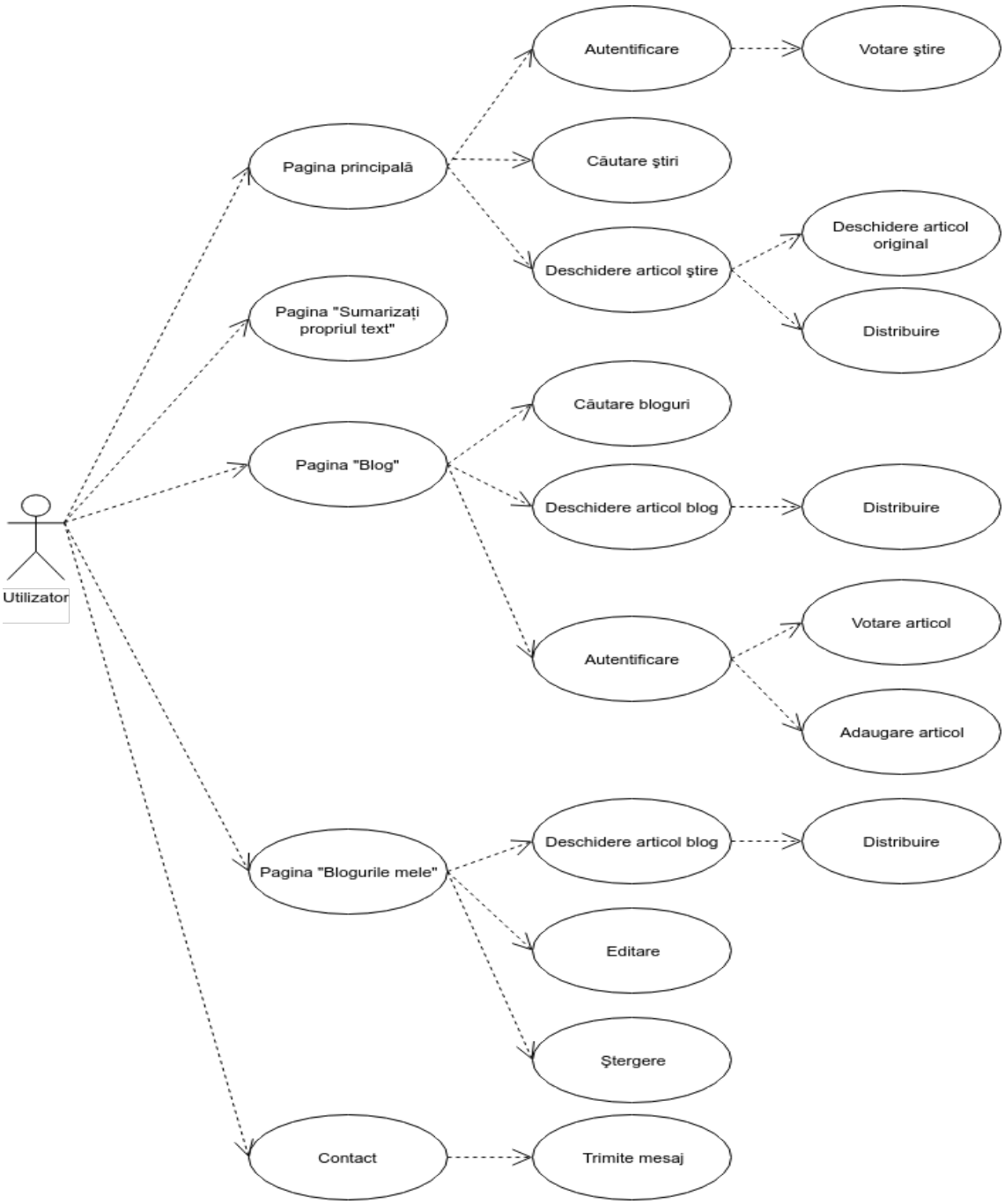

Figura 1. Diagrama E/R

4. Arhitectura aplicației

4.1. Ce este MVC?

Modelul arhitectural ales pentru lucrarea de față este Model-View-Controller. Acest *design pattern* își are originea în dezvoltarea aplicațiilor desktop, dar a fost introdus și în dezvoltarea aplicațiilor web.

După cum spune și numele, acest pattern împarte aplicația în trei componente principale:

- Model – responsabil pentru stocarea datelor și impunerea unor reguli și restricții asupra lor.
- View – poate fi orice reprezentare a datelor din model. De exemplu, o pagina web, o interfață grafică desktop, un grafic, un tabel etc
- Controller – leagă cele două componente de mai sus.

Câteva avantaje ale unui astfel de model arhitectural pot fi posibilitatea ca mai mulți dezvoltatori să lucreze la aplicație simultan, permite gruparea logică a acțiunilor dintr-un controller sau a view-urilor, modificările de cod se pot face cu ușurință, același model poate avea mai multe view-uri.

Dezavantajele unui astfel de model sunt greutatea de a naviga prin cod datorită straturilor de abstractizare suplimentare și împărțirea aplicației în trei componente va constrânge dezvoltatorul să găsească soluții care păstrează nemodificată această structură.

4.2. Introducere în Django

4.2.1. Structura unui proiect

Un proiect Django poate fi creat folosind comanda `django-admin startproject <nume_proiect>`. Un proiect nou conține un fișier numit *manage.py* care are același rol precum comanda *django-admin*, dar are grijă și de câteva setări în locul nostru și un folder cu același nume cu al proiectului. În interiorul acestui folder vom găsi alte patru fișiere.

Fişierul *__init__.py* este un fişier (în general gol) necesar pentru a indica faptul că acest folder este un pachet Python. Fără acest fişier, nu am putea importa folderul ca pe un modul Python.

Fişierul *settings.py* conţine toate setările proiectului nostru. Câteva setări importante din acest fişier pot fi, variabila *INSTALLED_APPS* care conţine o listă cu toate aplicaţiile Django folosite de proiect, iar de câte ori creăm o nouă aplicaţie, va trebui adăugată în această listă. *DATABASES* conţine informaţiile necesare realizării conexiunii cu baza de date. *SECURITY_KEY* conţine un şir de 50 de caractere şi reprezintă cheia folosită pentru criptarea datelor în tot proiectul. Este extrem de important ca această cheie să rămână secretă pentru a asigura integritatea aplicaţiei.

Fişierul *urls.py* conţine o listă de căi către celelalte aplicaţii din proiect şi se comportă ca un controller pentru aplicaţia noastră.

Fişierul *wsgi.py* este folosit pentru a asigura conformitatea cu platforma WSGI. Această platformă este un standard Python pentru servere web ce face posibilă procesarea de către server a oricărei aplicaţii indiferent de framework-ul web folosit.

Django a fost construit cu gândul de a fi modular. Cu alte cuvinte, dacă ne dorim să folosim o functionalitate din proiectul nostru şi în alte proiecte, nu trebuie decât să copiem acea aplicaţie Django în proiectele respective şi să adaugăm numele acesteia în variabila *INSTALLED_APPS* din fişierul *settings.py*.

Un astfel de modul poartă numele de *aplicaţie*. Un proiect poate conţine mai multe aplicaţii, iar fiecare aplicaţie este responsabilă cu o funcţionalitate sau o pagină web a acestuia. O aplicaţie Django este creată prin comanda `django-admin startapp <nume_aplicaţie>`.

Odată creată o aplicaţie, vom observa în interiorul acesteia şase fişiere Python şi un folder. Fişierul *__init__.py* are acelaşi rol cu cel din aplicaţia principală a proiectului.

Fişierul *apps.py* este folosit pentru a crea aplicaţii Django de tip plugin. Cu alte cuvinte, ne permite să construim o aplicaţie care poate fi copiată cu uşurinţă în alte proiecte fără a fi nevoiţi să o adăugăm în variabila *INSTALLED_APPS*.

Fişierul *tests.py*, aşa cum reiese din denumirea sa, va conţine toate testele pentru aplicaţia curentă. Deşi nu este obligatoriu să le scriem, testele sunt foarte importante pentru asigurarea unei bune funcţionări a aplicaţiei şi pot reduce cu mult timpul de dezvoltare identificând din timp eventualele bug-uri şi erori de logică.

Fişierul *models.py* conţine informaţii despre tabelele din aplicaţie. Fiecare tabel este reprezentat de o clasă, iar coloanele sunt atributele acestor clase. Pentru a adăuga tabelele acestea în baza de date, folosim comanda `python manage.py migrate`, iar atunci când efectuăm modificări asupra tabelelor, este necesar ca mai întâi să rulăm `python manage.py makemigrations`, iar după comanda anterioară. Aceste migrări sunt salvate în folderul *migrations* din interiorul aplicaţiei.

Fişierul *views.py* conține toate funcțiile şi clasele care definesc informațiile ce vor fi prezentate utilizatorului. Altfel spus, în acest fişier vom avea funcțiile callback pentru url-urile din fişierul *urls.py*. Deşi denumirea ne poate induce în eroare, fişierul acesta nu reprezintă componenta „view" din arhitectura MVC, acest rol revine template-urilor despre care vom discuta mai jos.

O aplicație Django are inclusă o interfață de administrare numită *admin* şi descrisă în fişierul *admin.py*. Folosind această interfață, putem gestiona datele din baza de date, utilizatorii aplicației şi multe altele. Pentru a folosi acest utilitar, trebuie să importăm modelul pe care vrem să îl gestionăm (descris în fişierul *models.py*) şi să adăugăm la finalul fişierului linia `admin.site.register(NumeModel)`, unde *NumeModel* este numele modelului descris în fişierul *models.py*.

Observăm totuşi că, în mod implicit, o aplicație Django nu conține un fişier *urls.py*. Acest lucru ne indică faptul că nu suntem obligați să avem legături de orice fel cu alte părți ale proiectului şi putem folosi această aplicație doar pentru prezentarea de date sau doar pentru a dezvolta o anumită funcționalitate cu scopul de a fi integrată în alte aplicații, dar care nu poate reprezenta o pagină web de sine stătătoare.

4.2.2. Construirea view-urilor

O altă întrebare la care trebuie să răspundem este: *unde este view-ul?* Structura Django ne indică să creăm un folder numit „templates" în interiorul aplicației şi să creăm toate fişierele HTML în acel director.

Totuşi pentru Django există un singur director „templates", iar tot conținutul folderelelor „templates" din toate aplicațiile proiectului sunt văzute de framework sub un acelaşi folder. Acest lucru poate duce la conflicte de nume între fişiere şi astfel se adaugă timp suplimentar pentru depanarea aplicației. Pentru a evita acest inconvenient, putem crea un alt director cu acelaşi nume cu cel al aplicației Django în care ne aflăm având astfel pentru fiecare aplicație structura: *templates/numeAplicație/fişier.html*.

Un fişier HTML poate deveni foarte complicat şi greu de întreținut, iar o aplicație web conține mai multe astfel de fişiere, mărind gradul te complexitate. Pentru a veni în ajutorul dezvoltatorilor, Django pune la dispoziție propriul limbaj de template care permite utilizarea instrucțiunilor de tip *if*, *for*, utilizarea variabilelor, extinderea sau includerea altor fişiere HTML etc. Aceste utilități uşurează întregul proces de dezvoltare şi facilitează reutilizarea codului.

Acest limbaj de template este evidențiat în document prin semnele „{% %}" pentru instrucțiuni şi „{{ }}" pentru variabile şi se bazează pe blocuri de instrucțiuni. Blocurile pot fi standard (de exemplu blocul unei instrucțiuni *if*) sau putem defini propriile noastre blocuri (`{% block numeBloc %} {% endblock %}`). Exemple de instrucțiuni folosind acest limbaj de template: `{% block for variabilă in listă_variabile %} *cod* {% endblock %}`, `<h1>`

`{{ variabilă }} </h1>`. Variabilele sunt trimise de către server atunci când este încărcată pagina.

Un mod bun de a crea un view reprezintă crearea unui fişier HTML care conţine structura de bază a paginii noastre, care nu depinde de datele din aplicaţie (vom numi acest fişier *header*), iar celelalte fişiere îl vor extinde, asigurând astfel un aspect consistent al aplicaţiei. Înainte de a extinde un fişier, este nevoie să includem cel puţin un bloc de tipul `{% block nume_bloc %}{% endblock %}` în fişierul pe care îl extindem. Pentru a extinde un fişier, tot ce trebuie să facem este să precizăm la începutul documentului copil „`{% extends „caleCătreFişier/header.html" %}`", iar apoi numele blocului în care vrem să adăugăm codul `{% block nume_bloc %} *codul html* {% endblock %}`, unde `nume_bloc` indică acelaşi nume cu cel al blocului din documentul părinte al cărui conţinut vrem să îl implementăm. Astfel, la încărcarea paginii, se va afişa întreg conţinutul fişierului părinte, iar în locul blocului, se va afişa conţinutul din fişierul copil. Mai mult, putem avea mai multe astfel de blocuri, iar un fişier poate fi extins de oricâte alte documente.

4.3. Preluarea automată a ştirilor

Aşa cum am stabilit încă de la începutul lucrării, ne dorim o aplicaţie care să poată prelua în mod automat articolele de ştiri şi să le sumarizeze. În cele ce urmează, vom dezvolta modul în care putem realiza acest lucru.

4.3.1. Parsarea ştirilor

Primul pas către soluţionarea problemei prezentate îl constituie însuşi preluarea ştirilor. Avem nevoie să identificăm ce ştiri dorim să prezentăm în aplicaţie, de la ce publicaţii am putea prelua conţinutul şi ce metode avem la îndemână pentru realizarea acestui ţel.

Identificarea tipului de ştiri şi a publicaţiilor este subiectivă, noi vom alege doar ştirile din categoria „world" (din întreaga lume) de la publicaţiile: *The New York Times*, *The Huffington Post* şi *BBC*, însă cu modificări minimale, putem adăuga şi alte publicaţii sau alte categorii de ştiri.

Din fericire, toate aceste publicaţii ne pun la dispoziţie un feed RSS ce ne permite să vedem într-un format simplificat toate articolele din categoria respectivă. Astfel, avem acces foarte rapid la titlul, descrierea, adresa URL şi imaginea asociată fiecărui articol de ştire.

Pentru a analiza un astfel de feed, vom folosi o librărie Python numită sugestiv *feedparser*.

Pentru o modularitate sporită, vom crea un fişier care va conţine codul pentru parsarea unui feed şi un fişier pentru sumarizarea articolelor returnate de acest parser. Vom numi aceste fisiere *feed_parser.py*, respectiv *get_summary.py.*

În fişierul *feed_parser.py* vom declara o funcţie *parse_rss(url)*, unde *url* este adresa feed-ului RSS. În această funcţie, creăm un dicţionar *d = feedparser.parse(url)* care va conţine toate informaţiile prezente în feed. Pentru a accesa aceste informaţii, este de ajuns să parcurgem lista *d.entries*. Fiecare element din această listă reprezintă o ştire, iar pe noi ne vor interesa numai titlul, descrierea, data publicării, adresa URL a articolului original şi adresa URL a imaginii asociate.

Vom adăuga toate aceste informaţii într-o tuplă, iar tupla într-o listă, astfel că, la final, vom avea o lista de tuple, unde fiecare tuplă reprezintă o ştire.

Până acum arată bine, însă lipseşte ceva: articolul în sine. Din păcate, acesta nu este prezent în niciunul din feed-urile noastre RSS. Pentru a depăşi această problemă, ne uităm la ce avem şi încercăm să găsim o soluţie în funcţie de acesta. Candidatul perfect este adresa URL a fiecărei ştiri. Dar nu e de ajuns, acea adresă ne va duce către pagina unde este prezentată ştirea în întregime, iar noi avem nevoie doar de textul acesteia.

Aşadar, problema s-a redus la a găsi o modalitate prin care să analizăm un fişier HTML şi să preluam textul dintre anumite taguri (i.e. textul articolului). Pentru aceasta, avem la dispoziţie librăria BeautifulSoup. Cu ajutorul acesteia putem identifica şi manipula foarte uşor elementele HTML dintr-un fişier.

Construim o nouă listă folosind lista construită la pasul anterior ce va conţine toate URL-urile ştirilor. Parcurgem noua listă şi pentru fiecare URL, accesăm acea adresă folosind `news_url = urllib.request.build_opener(urllib.request.HTTPCookieProcessor).open(Request(url,headers={'User-Agent': 'Mozilla/5.0'})).read()`, unde *url* reprezintă adresa fiecărui articol. În această linie, deschidem un request (o cerere) către o adresă specificată de noi şi apoi citim răspunsul. Parametrul *headers* îi spune serverului că această cerere vine de la un browser (Mozilla). Apoi instanţiem un obiect BeautifulSoup prin `soup = bs.BeautifulSoup(news_url, 'lxml')`, unde *news_url* este creat anterior, iar '*lxml*' reprezintă tipul de parser (analizator) folosit. În afară de *lxml*, se mai pot folosi şi *html.parser*, *lxml-xml*, *xml* sau *html5lib*.

Mai departe, încercăm să găsim modele în funcţie de care putem identifica articolele. Pentru a face acest lucru, putem da click dreapta pe una dintre paginile ştirilor

şi selectăm „inspectați elementul” („inspect element”). Astfel avem acces la codul HTML din spatele acelei pagini. Pentru a selecta un element folosind BeautifulSoup, putem folosi metodele *find* şi *find_all*. Diferența dintre cele două este că *find* returnează primul element care îndeplineşte condiția oferită ca parametru, iar *find_all* returnează o listă cu toate elementele care îndeplinesc această condiție.

În cazul nostru, observăm că pentru a identifica un articol sunt folosite tagurile *p*. Cum aceste tag-uri pot să nu aibă nici o clasă sau id asociate în funcție de care să căutăm, încercăm să găsim un element părinte care să cuprindă toate aceste *p*-uri, dar care are cât mai puține elemente care nu fac parte din articol. Pentru publicațiile alese, există câte un *div* (sau mai multe) cu o clasă specifică articolului.

Prin urmare, în codul nostru adăugăm o variabilă care să identifice numele publicației pentru care realizăm extragerea. Acest lucru este posibil folosind expresii regulate, identificând textul aflat între *„www.”* şi *„.co”*. Apoi într-o instrucțiune *if*, folosind *find_all* din librăria BeautifulSoup, identificăm elementele *div* ce conțin paragrafele articolului. Mai rămâne doar să construim o listă ce conține paragrafele din *div*-urile identificate anterior, iar apoi să concatenăm textul din interiorul acestor paragrafe şi să ataşăm rezultatul la lista noastră de ştiri.

Din păcate, paginile web nu sunt construite pentru a face preluarea datelor mai uşoară (uneori chiar dimpotrivă). Astfel că, dacă ne uităm la textele preluate folosind metoda anterioară, vom observa multe paragrafe sau bucăți de text care nu au legătură cu articolul în sine. Exemple de astfel de paragrafe pot fi îndemnuri adresate uilizatorului să se aboneze la acea publicație, texte care anunță prezența unei reclame şi multe altele.

Pentru a elimina aceste porțiuni de text, vom construi o listă în care vom identifica toate paragrafele „inutile” aplicației noastre folosind aceleaşi metode descrise mai sus, iar înainte să construim textul articolului, vom adăuga o instrucțiune *if* în care vom verifica dacă paragraful pe care dorim să-l concatenăm la articol nu este în lista de paragrafe nefolositoare.

4.3.2. Sumarizarea ştirilor

Trăim într-un moment de plină ascensiune a informațiilor. În fiecare zi cantitatea de date textuale devine din ce în ce mai mare. Toate paginile web, toate blogurile, articolele de ştiri, actualizările de stare de pe diferite aplicații de socializare şi aşa mai departe, extind nivelul de date actual. Însă, acest lucru vine cu un cost. Toate aceste date sunt nestructurate şi necesită metode speciale pentru a extrage doar informațiile utile. O metoda foarte bună pentru a combate acest lucru o reprezintă indexarea, dar aceasta

implică pierderea formei inițiale a textului, lucru care pentru maşini nu este important, dar pentru noi, oamenii, face diferența.

De aceea, problema extragerii doar a informațiilor importante dintr-un text, păstrând în acelaşi timp sensul şi forma acestuia, devine din ce în ce mai evidentă. După cum vom vedea în cele ce urmează, această problemă *nu* este una trivială, ci reprezintă încă o problemă deschisă în domeniu.

Această problemă poate fi abordată în două moduri fundamentale: sumarizarea extractivă şi sumarizarea abstractivă.

Sumarizarea extractivă presupune extragerea cuvintelor şi a propozițiilor din textul inițial, şi construirea rezumatului fără a le modifica. Sumarizarea abstractivă îşi propune alcătuirea unui rezumat ce conține fraze şi cuvinte noi care să cuprindă sensul textului inițial. Prima metodă este mult mai uşor de implementat şi este de multe ori metoda aleasă atunci când avem de-a face cu astfel de probleme. Cea de-a doua metodă, deşi este mult mai greu de implementat şi, cu tehnologia pe care o avem în momentul de față, poate produce rezultate mai slabe decât contracandidata sa, rămâne metoda folosită de oameni pentru a rezuma un text (astfel, potențialul său este foarte mare). În această lucrarea, ne vom concentra numai pe metoda extractivă.

În continuare vom prezenta câțiva algoritmi de sumarizare.

Algoritmul Luhn, publicat în anul 1958, presupune alegerea propozițiilor importante în funcție de cuvintele cu frecvența cea mai mare şi distanța liniară dintre acestea.

Algoritmul LexRank este bazat pe grafuri similar cu algoritmul TextRank. Acest algoritm foloseşte cosinusul IDF-modificat drept măsură de similaritate între două propoziții, iar această similaritate va reprezenta costul drumurilor din graf. De asemenea, LexRank include şi un pas de post-procesare în urma căruia se asigură că propozițiile alese nu sunt prea asemănătoare între ele.

Algoritmul LSA (Latent Semantic Analysis), spre deosebire de ceilalți doi algoritmi prezenzați, abordează problema „înțelesului” unui cuvânt. Într-adevar, un cuvânt poate avea mai multe înțelesuri în funcție de contextul în care se află sau cuvinte diferite pot avea acelaşi sens (sinonime). Aceste lucruri pot pune în dificultate chiar şi o persoană. Soluția propusă de algoritm presupune scufundarea spațiului cuvintelor noastre într-un spațiu de dimensiune mai mică (numit şi spațiu latent) în care compararea a două cuvinte poate fi realizată cu mai mare uşurință.

Astfel, pentru a rezolva această problemă, LSA impune următoarele condiții:

- Documentele sunt reprezentate folosind modelul „bag of words” („sac de cuvinte”), care presupune asocierea fiecărui cuvânt cu numărul de apariții în document. Totuşi, utilizând acest model, vom pierde informațiile legate de ordinea cuvintelor în document;
- Conceptele sunt reprezentate de cuvinte care apar de obicei împreună în documente. De exemplu, cuvintele „mâncare”, „masă”, „chelner” apar preponderent în texte referitoare la un restaurant;
- Se presupune că orice cuvânt are un singur sens. Acest lucru este, evident, fals, însă ne permite să abordăm această problemă.

În lucrarea noastră, a fost ales algoritmul LSA deoarece în urma încercărilor repetate, au fost observate cele mai bune rezultate comparativ cu ceilalți algoritmi. Pentru a folosi acest algoritm, vom utiliza modulul Python „sumy”. Acest modul ofera implementări pentru toți cei trei algoritmi de mai sus.

Pentru a realiza rezumatul fiecărui articol de ştire, creăm un fişier nou, conţinând o metoda numită *make_summary(news_sites, language, sentence_count)*, unde *news_sites* reprezintă lista de adrese URL către fiecare feed RSS pe care ne dorim să îl analizăm, *language* reprezintă limba în care este scris textul pe care ne dorim să îl sumarizăm (*sumy* suportă mai multe limbi printre care engleză, franceză, cehă, chineză etc, dar din păcate nu şi română), iar *sentence_count* va fi numărul de propoziţii care alcătuiesc rezumatul nostru. Pentru această aplicaţie, vom folosi limba engleză şi un număr de trei propoziţii pentru fiecare rezumat.

În interiorul funcției, vom parcurge lista de adrese URL conținută în parametrul *news_sites*, iar pentru fiecare element, apelăm funcția *parse_rss(url)* care va parsa fiecare feed RSS aşa cum am descris la pasul anterior. Având lista rezultată în urma parsării, ne dorim să rezumăm doar textul articolului. În acest sens, vom parcurge această listă şi vom efectua sumarizarea şi salvarea în baza de date.

Pentru sumarizare, ne vom asigura înainte că textul pe care ne dorim să îl rezumăm are cel puțin 1500 de caractere, întrucât un text de dimensiuni mai mici ar fi suficient de scurt încât această operație să îşi piardă sensul.

Acum este momentul să folosim librăria *sumy*. Primul pas constă în crearea obiectului *parser* specificând textul articolului şi tokenizerul care va fi folosit pentru a despărți în cuvinte, respectiv propoziții articolul. Acest lucru se realizează cu următoarea linie de cod: `parser = PlaintextParser.from_string(text, Tokenizer(language))`, unde *text* reprezintă textul articolului, *Tokenizer(language)* este o clasă pusă la dispoziție de *sumy* ce primeşte la rândul său în constructor, limba în care este scris articolul (aceeaşi cu cea dată drept parametru funcției noastre).

Următorul pas este să creăm un obiect *summarizer* utilizând clasa *Summarizer* din interiorul modulului şi să îi pasăm în constructor un alt obiect de tip *Stemmer* care va avea care va conține la rândul său, limba articolului. Linia de cod care realizează cele descrise arata astfel: `summarizer=Summarizer(Stemmer(language))`. Inițializăm câmpul *stop_words* al stemmerului nostru folosind funcția *get_stop_word(language)* şi parcurgem tuplul rezultat în urma apelului *summarizer(parser.document, sentence_count)*, unde *summarizer* şi *parser* sunt obiectele descrise mai sus, iar *sentence_count* este cel de-al treilea parametru al funcției noastre *make_summary*. Fiecare element al acestei tuple reprezintă o propoziție a rezumatului nostru. Astfel, construim un nou string prin concatenarea acestor propoziții şi obținem textul sumarizat.

Deşi poate părea ciudat că este apelat un obiect asemeni unei funcții sau metode obişnuite, acest lucru este posibil, în Python, prin suprascrierea metodei magice *__call__* din interiorul clasei, lucru realizat deja de autorul librăriei.

Ultimul pas este să salvăm în baza de date o nouă înregistrare care să conțină toate informațiile preluate de parser (titlu, descriere, data articolului, adresa URL a articolului şi adresa URL a imaginii asociate acestuia) şi textul sumarizat.

4.3.3. Generarea automată de etichete

Cum tot procesul descris până acum este realizat în mod automat, apare nevoia unei ordonări a informațiilor. Un prim gând în această direcție ar fi posibilitatea de a accesa anumite articole în funcție de o temă, sau un termen. Cu alte cuvinte, ne dorim să avem la dispoziție un mecanism de căutare.

O soluție naivă la această problemă ar fi să căutăm fiecare cuvânt din textul nostru de căutare în textul, titlul şi descrierea articolelor sumarizate. Deşi în aparență rezolvă problema cu un minim de efort, această metodă se dovedeşte a fi foarte înceată pentru un volum de date foarte mare. Mai mult, această soluție nu ne oferă nici un fel de organizare a informațiilor din aplicația noastră.

Soluția aleasă de noi va fi asocierea de etichete (la care ne vom referi în continuare prin termenul de *„taguri”*).

Într-adevăr, prin intermediul tagurilor, putem deduce o anumită temă comună între articole, căutarea este cu mult uşurată datorită faptului că un tag poate fi asociat mai multor articole şi, prin urmare, vor fi mult mai puține taguri decât articole, astfel rezolvând problema volumelor mari de date.

Totuşi, ne lovim de un inconvenient. Automatizarea întregului proces nu ne permite să asociem manual taguri fiecărui articol (cel puțin nu fără un efort considerabil). Deci, avem nevoie să automatizăm şi generarea tagurilor împreună cu cea a rezumatelor.

Pentru a rezolva această problemă, vom crea o nouă funcție numită *get_tags(text)*, unde *text* reprezintă textul din care vrem sa extragem tagurile şi vom importa modulul Python *nltk* pentru a ne asigura toate utilitarele de care avem nevoie.

Ne propunem ca tagurile noastre să reprezinte cele mai importante bigrame şi trigrame din fiecare articol nesumarizat (alegem varianta nesumarizată pentru că ne oferă mult mai multe informații despre conținutul articolului). Bigramele şi trigramele reprezintă seturi de câte două, respectiv trei cuvinte consecutive dintr-un text. Vom presupune că dacă două, respectiv trei cuvinte consecutive se găsesc cu o frecvență suficient de mare în text, înseamnă că acea asociere deține informații importante despre conținutul acestuia.

Implementarea funcției noastre începe prin a inițializa variabila ce conține lista cu stopwords (cuvinte care au o frecvență ridicată în majoritatea textelor şi, astfel, nu dețin informații importante despre conținutul acestuia) specifice limbii engleze. Apoi, folosind modulul *nltk.collocations* creăm două obiecte de tipul *BigramAssocMeasures*, respectiv *TrigramAssocMeasures* care ne vor fi de folos în calcularea scorurilor. Vom numi aceste obiecte *bigram_measures* şi *trigram_measures*.

Definim o *colocație* drept o expresie formată din mai multe cuvinte care apar împreună în mod frecvent. Aplicând această definiție pentru problema noastră, vom încerca să găsim colocații de bigrame şi trigrame.

În continuare vom descrie algoritmul doar pentru bigrame, cazul trigramelor fiind analog. Pentru a găsi toate colocațiile din text, folosim următoarea linie de cod `finder_b=BigramCollocationFinder.from_words(word_tokenize(text))`. Obiectul obținut va conține toate colocațiile pentru bigramele din textul nostru. Pentru a păstra doar informațiile care contează, filtrăm datele astfel încât să fie luate în considerare doar colocațiile cu au o frecvență mai mare sau egală cu trei şi eliminăm orice cuvânt care se găseste în lista de stopwords declarată anterior sau a cărui lungime este mai mică de trei litere.

Pasul următor este să oferim un punctaj fiecărei colocații în funcție de importanța sa în text, iar acest lucru este posibil apelând metoda *score_ngrams* a obiectului *finder_b* declarat mai sus, pasând ca parametru măsura *bigram_measures.pmi*. PMI (Pointwise Mutual Information) este o măsură de asociere folosită în statistică ce determină discrepanța dintre probabilitatea ca două evenimente

să se întâmple simultan şi produsul probabilităților individuale (presupunând că cele două evenimente sunt independente).

În urma pasului anterior, am obținut o listă de bigrame ordonate în ordinea celor mai importante în funcție de punctaj. Alegem primele 15 elemente şi adăugăm la această listă şi lista trigramelor obținute în mod analog. De asemenea, sortăm lista nou formată în funcție de punctaj.

Dacă afişăm această listă, observăm că fiecare element este format dintr-o tuplă ce conține fiecare cuvânt din bigramul sau trigramul corespunzător şi punctajul asociat. Prin urmare, parcurgem lista şi formăm o nouă listă ale cărei elemente vor fi tuple ce conțin cuvintele bigramului sau trigramului concatenate într-un singur string şi punctajul asociat.

Returnăm lista nouă.

În acest moment, avem o cale de a extrage tagurile dintr-un text de dimensiuni mai mari (întrucât am ales ca frecvența bigramelor şi a trigramelor să fie mai mare de trei respectiv doi, într-un text aleator de dimensiuni mici, este o probabilitate mică de a găsi un tag care să îndeplinească aceste condiții), tot ce a rămas de făcut este să folosim această funcție pentru articolele noastre şi să le salvăm în baza de date.

Astfel, revenind în funcția *make_summary* discutată în secțiunea precedentă, imediat după ce am salvat în baza de date ştirea cu rezumatul asociat, vom apela funcția *get_tags* cu parametrul textul articolului integral (motivul pentru această alegere a fost prezentat mai sus).

Acum avem lista cu tagurile extrase. Din această listă, eliminăm tagurile care au rezultat din aceleaşi structuri, dar au fost colocate diferit. De exemplu, numele Mohamed bin Salam este un trigram valid şi care poate avea o importanță crescută în textul analizat, însă acesastă structură poate genera şi bigramele Mohamed bin şi bin Salam care sunt redundante).

Parcurgem lista obținută şi dacă tagul curent nu există în baza de date, îl adăugăm împreună cu punctajul său şi adăugăm şi o legătură între tag şi ştirea asociată. Dacă tagul există deja în baza de date, adăugăm doar legătura cu ştirea corespunzătoare.

În acest moment, avem salvate în baza de date articolul de ştire cu textul sumarizat şi o listă cu tagurile asociate acestuia. Trebuie precizat totuşi că datorită restricțiilor impuse pentru taguri, pot exista articole de ştire fără nici un tag asociat. Acest lucru este într-adevăr nefericit, iar soluționarea sa va fi introdusă într-o versiune viitoare a aplicației..

4.3.4. Programarea procesului

În secțiunile precedente, am discutat despre modul în care se desfăşoară întregul proces de preluare şi sumarizare automată a articolelor de ştiri. După cum am văzut, acest proces nu este deloc simplu şi necesită un timp de procesare considerabil pentru a genera răspunsul dorit. Mai mult, în starea actuală, generarea rezumatelor se realizează la încărcarea paginii ceea ce înseamnă că dacă deschidem de două ori aplicația simultan, vom observa că fiecare instanță va rula acelaşi proces, rezultând în articole duplicate.

Bineînțeles, acest lucru nu este de dorit, iar rezolvarea acestei probleme devine prioritară.

Analizând problema cu care ne confruntăm, ne dăm seama că problema rezidă în alegerea noastră de a lega acest proces de ciclul de viață al aplicației. Prin urmare, avem nevoie de o metodă care să ruleze acest proces automat şi independent de starea aplicației.

Soluția pentru această problemă este Celery. Celery ne oferă o modalitate foarte uşoară şi rapidă de a crea taskuri asincrone şi de a le programa la un interval prestabilit de timp.

Înainte de a crea taskul propriu-zis, trebuie să configurăm acest modul pentru a avea o integrare bună cu aplicația noastră. Primul pas ar fi să creăm un fişier *celery.py* în interiorul aplicației Django principale (cea cu aceelaşi nume ca al proiectului), iar în interiorul acestuia adăugăm un fragment de cod prestabilit[1], pe care îl adaptăm astfel încât să conțină informațiile corespunzătoare proiectului nostru. De asemenea, pentru a ne asigura că Celery va fi disponibil odată cu încărcarea proiectului nostru, adăugăm următorul import în interiorul fişierului *__init__.py* din acelaşi director: `from .celery import app as celery_app`. Prin *.celery* ne referim la fişierul creat anterior.

Celery foloseşte brokeri pentru transmiterea de mesaje între proiectul Django şi muncitorii săi (workers). În această lucrare, vom alege Redis pentru această funcționalitate. Pentru a-l utiliza, tot ce trebuie să facem este să adăugăm:

```
BROKER_URL = 'redis://localhost:6379'
CELERY_RESULT_BACKEND = 'redis://localhost:6379'
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
```

1 *Codul în sine poate fi găsit în documentația oficială: http://celery.readthedocs.io/en/latest/getting-started/next-steps.html#using-celery-in-your-application*

`CELERY_TIMEZONE = 'UTC'`

în fişierul *settings.py*. Aceste informații sunt suficiente pentru a lega Celery şi Redis între ele şi pentru a specifica tipul datelor ce vor fi interpretate.

Acum că avem totul configurat corespunzător, putem crea taskul care ne va porni procesul de sumarizare în mod asincron şi programat să se execute odată la 15 minute.

În acest sens, creăm un fişier *tasks.py* în acelaşi modul în care au fost definite şi fişierele ce conțin parserul, summarizerul şi generatorul de taguri, şi creăm o funcție în interiorul acestuia pe care o vom numi *get_news*. Această funcție va primi ca parametrii lista de feeduri RSS, limba în care sunt scrise articolele (şi care se va folosi pentru sumarizare şi tagare automată) şi numărul de propoziții care vor alcătui rezumatul articolelor. De asemenea, singurul lucru făcut de această funcție este să apeleze funcția *make_summary* descrisă într-o secțiune anterioară a lucrării cu parametrii de mai sus.

Pentru a transforma această funcție Python într-un task Celery, este suficient să adăugăm un decorator deasupra acesteia. Pentru un task normal se foloseşte decoratorul *@task(name='nume task')*, iar pentru un task care trebuie rulat periodic, decoratorul *@periodic_task(run_every=(crontab(minute='*/15')),name="get_news",ignore_result=True)*. În cazul nostru, vom folosi cea de-a doua variantă exemplificată deja cu parametrii corespunzători.

Un decorator în Python, reprezintă o funcție care primeşte ca parametru funcția decorată cu scopul de a-i extinde comportamentul fără a o modifica în mod direct. Cu alte cuvinte, este un wrapper peste funcția noastră ce ne permite să efectuăm anumite operații înainte şi/sau după rularea acesteia.

Odată definit taskul, tot ce a rămas de făcut este să îl rulăm. Din fericire, Celery face acest lucru uşor pentru noi, astfel că, este suficient să rulăm două comenzi în linia de comandă, una pentru a rula taskul în sine, iar cea de-a doua pentru a ne asigura că se rulează în intervalul precizat. Aceste comenzi sunt: `celery -A newssummy worker -l info`, respectiv `celery -A newssummy beat -l info`. Parametrul *-A* specifică numele proiectului, iar *-l* nivelul de la care să se salveze logurile (de la nivelul *info* în sus).

În producție, aceste procese trebuie întreținute de un tool (instrument) de monitorizare a proceselor precum *Supervizor*. Astfel, vom avea mult mai mult control asupra proceselor de fundal din aplicația noastră şi vom avea o acoperire mai bună în cazuri excepționale precum căderi ale rețelei electrice sau un crash al serverului. Detalierea acestei probleme depăşeşte însă scopul acestei lucrări, astfel că nu va fi tratată aici.

4.4. Analiza tehnică a aplicației

4.4.1. Sistemul de votare

Dezvoltarea unei aplicații la care utilizatorii să îşi dorească să se întoarcă, ar trebui să fie unul din principalele scopuri pentru orice tip de proiect. De aceea, ne propunem să transformăm aplicația noastră statică, care nu face decât să afişeze ştiri rezumate de pe internet, într-una interactivă în care utilizatorii îşi pot exprima opiniile în legătură cu ce este prezentat pe site sau cu ce se întâmplă în lume.

Aşa cum am discutat într-un capitol anterior, pentru exprimarea opiniei vis-a-vis de evenimentele reale, utilizatorul are la dispoziție pagina de bloguri, iar pentru a-şi exprima opinia despre conținutul prezentat în aplicație, vom introduce un sistem de votare.

Sistemul propus, după cum vom observa, este unul simplu, dar foarte eficient. Pe scurt, algoritmul pe care îl vom aborda constă în crearea unor căi unice fiecărui articol pentru votarea pozitivă, respectiv negativă a acestora. Unicitatea rutelor va fi dată de includerea cheii primare a articolului în compoziția acestora. Apoi, accesarea fiecărei dintre căile astfel create va apela o funcție de votare (pozitivă sau negativă, după caz) care va adăuga o nouă înregistrare într-o tabelă din baza de date care va conține id-urile articolelor şi ale userilor, împreună cu natura votului (pozitiv sau negativ). În final, în template vom afişa totalul acestor voturi pentru fiecare articol.

Să începem prin crearea căilor. Acestea sunt adăugate în fişierul *urls.py* din interiorul fiecărei aplicații Django, într-o listă denumită *urlpatterns*. Structura unei căi este *url(expresie_regulată, funcția_asociată, numele_rutei)*, unde *expresie_regulată* reprezintă un model care defineşte calea relativă (față de pagina părinte) unde se va găsi pagina în cauză, *funcția_asociată* este funcția sau metoda care se va apela atunci când este accesată adresa descrisă de expresia regulată anterioară, iar *numele_rutei* este un parametru opțional care ne permite să accesăm ruta din interiorul codului nostru, astfel evitând hard-codarea adresei. Aşa cum am spus, cel de-al treilea parametru nu este obligatoriu, dar este considerată o practică bună să îl folosim atunci când vrem să accesăm rute din interiorul codului.

Cele două rute corespunzătoare votării pozitive şi negative arată astfel: `url(r'^upvote/(?P<news_id>[0-9]+)/$',views.upvote,name='upvote')`, respectiv `url(r'^downvote/(?P<news_id>[0-9]+)/$', views.downvote, name='downvote')`. Expresia regulată folosită pentru construirea rutei este alcătuită din caracterul „^” care defineşte începutul unui string,

urmat de cuvântul *upvote* sau *downvote* în funcție de natura votului (pozitiv, respectiv negativ) şi de un „ / ”, apoi avem blocul *(?P<news_id>[0-9]+)* ce reprezintă id-ul articolului de ştire din baza de date şi încheiem prin caracterul „ / ” şi „ *$* ” care indică sfârşitul şirului de caractere.

Blocul *(?P<news_id>[0-9]+)* poartă numele de *named group* (grup denumit sau grup cu un nume asociat) şi va fi folosit pentru a indica faptul că modelul care va urma este legat de id-ul ştirilor. Blocul *[0-9]+* indică una sau mai multe cifre consecutive (cu alte cuvinte, un număr). Django va trata acest bloc drept un parametru care va trebui trimis funcției asociate alături de request (care este trimis în mod implicit), iar numele parametrului va fi acelaşi cu numele specificat între parantezele ascuțite „<>”.

Următorul parametru oferit în construcția rutelor noastre este funcția asociată. Aşa cum putem observa, ele sunt prezente în fişierul *views.py* şi poartă numele *upvote*, respectiv *downvote*. În continuare vom descrie aceste funcții.

Funcția *upvote* va primi drept parametrii requestul şi id-ul articolului care trebuie votat şi nu va returna nimic.

Primul pas va fi să verificăm tipul metodei care a dus la apelarea acestei funcții. Această informație poate fi accesată prin intermediul parametrului *request* verificând dacă *request.method == "GET"*. În caz afirmativ, păstrăm obiectul utilizatorului care a efectuat cererea de votare prin `user = request.user` şi obiectul articolului votat prin `news = News.objects.get(pk=news_id)`, unde *news_id* este numărul „prins” de blocul denumit din descrierea rutei.

Următorul pas este să adăugăm în tabela *UserNews* o nouă înregistrare în cazul în care este pentru prima dată când utilizatorul votează acea ştire sau să actualizăm acel câmp în caz contrar.

Acest lucru este realizat creând un obiect al tabelei *UserNews* care să aibă id-ul utilizatorului şi al articolului egal cu cele definite anterior. În cazul absenței unei astfel de înregistrări în baza de date, obiectul va primi valoarea *None*.

Astfel, dacă este pentru prima dată când utilizatorul votează acel articol (dacă obiectul are valoare *None*), vom incrementa valoarea celulei *vote_up* din tabela *News* cu o unitate şi vom adăuga o nouă înregistrare în tabela *UserNews* care va conține id-ul articolului, id-ul utilizatorului şi valoarea *1 (unu)* dacă votul este de natură pozitivă, respectiv *-1 (minus unu)* dacă este de natură negativă. Apoi se reîncarcă pagina pentru a afişa votul.

Dacă utilizatorul a votat deja acel articol în mod negativ, atunci se va decrementa cu o unitate valoarea celulei *vote_down* din tabela *News* şi se va incrementa celula

vote_up. Apoi se va actualiza timpul votului şi natura acestuia (din negativ, în pozitiv) în tabela *UserNews* şi se va reîncărca pagina pentru a afişa noul vot.

În mod analog se va defini şi funcția de *downvote*.

Ambele funcții vor putea fi apelate doar de utilizatori autentificați în aplicație. Pentru a specifica acest lucru, este suficient să adăugăm decoratorul *@login_required* din modulul *django.contrib.auth.decorators* deasupra definiției funcțiilor noastre. Rezultatul fiind că utilizatorul va fi redirecționat către pagina de autentificare atunci când va încerca să adauge un vot în mod anonim.

Afişarea voturilor totale şi a numărului acestora se va realiza în fişierul *HTML* care alcătuieşte pagina. Aceste operații sunt posibile datorită limbajului de template oferit de Django. Astfel, este posibil să verificăm dacă utilizatorul este autentificat sau nu printr-o instrucțiune precum `{% if request.user.is_anonymous %} cod_html_utilizator_neautentificat {% else %} cod_html_utilizator_autentificat {% endif %}` şi putem efectua operația de adunare prin `{{ n.vote_up| add:n.vote_down}}`, unde *n* este articolul din tabela *News*. Vom folosi prima instrucțiune pentru a afişa butoanele de votare drept inactive atunci când utilizatorul este deconectat, iar cea de-a doua pentru a afişa numărul de utilizatori care au votat acel articol.

Pentru a afişa totalul de voturi asociate articolului vom fi nevoiți să scădem numărul de voturi negative din numărul de voturi pozitive. Din nefericire, limbajul de template oferit de Django nu ne pune la dispoziție o modalitate de scădere a două variabile. Totuşi, putem trece peste acest inconvenient, definindu-ne propria funcție care să îndeplinească această cerință.

Aşadar, într-un nou folder numit *templatetags* din interiorul aplicației Django în care ne aflăm, creăm un fişier *custom_filters.py*, iar în interiorul acestuia, adăugăm o funcție numită *sub* care va primi doi parametri şi va returna diferența dintre aceştia. Pentru a indica totuşi că această funcție trebuie tratată drept un filtru pentru limbajul de template, va trebui să importăm clasa *Library* din modulul *django.template* şi să instanțiem un obiect de tipul acesteia, iar apoi apelăm metoda *filter* a acestui obiect pasând drept argumente numele filtrului aşa cum va fi folosit în template şi funcția care va efectua operația dorită.

Având filtrul definit, ne întoarcem în fişierul HTML care defineşte template-ul pentru pagina noastră şi adăugăm `{% load custom_filters %}`, la începutul acestuia, pentru a încărca filtrul definit de noi, iar mai jos, unde ne dorim să afişăm numărul total de voturi atribuite articolului, adăugăm variabila `{{ n.vote_up | sub:n.vote_down }}`.

Mai mult, tot procesul descris în această secțiune va fi adaptat şi pentru aplicația Django corespunzătoare paginii de bloguri.

4.4.2. Înregistrarea şi autentificarea utilizatorilor

Aşa cum am indicat până acum, rolul unui utilizator este foarte important în aplicația noastră. Mai mult, unii utilizatori vor avea mai multe drepturi decât ceilalți. Prin urmare, avem nevoie de o modalitate prin care să putem diferenția aceste tipuri.

Din fericire, Django ne pune la dispoziție, încă o dată, tot ce avem nevoie pentru a rezolva această problemă.

Sistemul de autentificare Django suportă toate cerințele de bază pentru un sistem de autentificare complet. Astfel, folosind acest framework, avem posibilitatea de a gestiona conturile de utilizatori, grupurile, pemisiunile, precum şi sesiunile bazate pe cookie-uri.

Este important de precizat faptul că Django include atât un sistem de autentificare, cât şi unul de autorizare. Astfel, autentificarea reprezintă acțiunea de a verifica validitatea credențialelor utilizatorului, iar autorizarea implică verificarea drepturilor (permisiunilor) asociate acestuia. În cele ce urmează, ne vom referi la ambele sisteme prin termenul de „autentificare”.

În Django, un utilizator nu este altceva decât un obiect al clasei *User* din modulul *django.contrib.auth.models*. Implicit, atributele acestui obiect sunt numele de utilizator (username), parola (password), email, prenumele (first_name) şi numele de familie (last_name).

Pentru înregistrare, vom crea un formular, care atunci când trimite o cerere de tip *POST*, verifică dacă numele şi emailul utilizatorului există deja în baza de date (în caz afirmativ, se va afişa un mesaj care să indice acest lucru), iar dacă nu există, creăm utilizatorul şi îl adăugăm în baza de date.

Pentru crearea unui utilizator, vom folosi metoda *create_user*, care primeşte drept argumente doar numele de utilizator, parola şi email-ul. În urma acestui apel, vom obține un obiect de tip *User* care conține aceste informații. După cum probabil ați observat, numele şi prenumele nu au fost incluse. Pentru a realiza acest lucru, este suficient să pasăm aceste informații, câmpurilor *first_name* şi *last_name* ale obiectului *user*. Ultimul pas este salvarea acestui utilizator în baza de date. Acest lucru se realizează apelând metoda *save()* a obiectului *user*.

De asemenea, vom trimite un email către adresa utilizatorului cu scopul de a anunța succesul creării contului său. Pentru a trimite acest mail, este necesară setarea informațiilor despre serverul care se ocupă de trimiterea acestor emailuri precum şi a altor date precum numele de utilizator şi parola pentru acel server, portul etc. Detaliile despre conținutul acestor câmpuri țin de soluția aleasă, iar descrierea lor nu este importantă pentru lucrarea de față. Ceea ce ne interesează totuşi, este modul în care este folosită această funcționalitate.

Având toate acestea în vedere, trimiterea email-ului de confirmare este posibilă folosind funcția *send_mail* din modulul *django.core.mail*. Această funcție primeşte drept parametrii, subiectul email-ului, conținutul, adresa emițătorului şi o listă cu persoanele care vor primi acest mail (în cazul nostru lista va conține un singur element: adresa utilizatorului tocmai creat).

Mai mult, există posibilitatea de a trimite email-ul înainte de adăugarea în baza de date a utilizatorului. Scopul fiind de a verifica validitatea adresei introduse. Astfel, înainte de salvarea obiectului *user* în baza de date, setăm valoarea câmpului *is_active=False* şi utilizând modulul Python *hashlib*, creăm un hash bazat pe numele de utilizator care va reprezenta cheia de activare a contului şi setăm o dată de expirare (vom alege două zile de la crearea contului).

Creăm în fişierul *models.py* o nouă clasă care extinde *models.Model* şi care va reprezenta o tabelă în relație *unu-la-unu* cu tabela *User*. Câmpurile acestei tabele vor fi cheia de activare, data de expirare, o coloană care indică dacă acea cheie a fost folosită sau nu şi cheia străină către utilizatorul căruia îi corespunde acea cheie de activare.

Revenind la funcția noastră de înregistrare, creăm un obiect al clasei tocmai create care va primi în constructor, obiectul utilizatorului ce va fi creat, cheia de activare corespunzătoare şi data de expirare a acesteia. Adăugăm aceste informații în baza de date folosind metoda *save()* şi trimitem email către adresa specificată de viitorul utilizator în care vom include un link unic către aplicația noastră care va conține cheia de activare.

Atunci când este accesată această adresă unică, se va apela o altă funcție din fişierul *views.py* care primeşte cheia de activare ca parametru şi care, după verificarea validității cheii (să nu fie expirată şi să nu fie folosită deja), precum şi verificarea ca utilizatorul să nu fie deja autenficat cu un cont existent, identificăm utilizatorul folosind cheia de activare primită ca parametru şi actualizăm câmpul *is_active=True* al utilizatorului. De asemenea, marcăm cheia drept „folosită” setând câmpul *used_key=True*.

În mod analog, se va implementa şi funcționalitatea de schimbare a parolei unui utilizator.

Acum că avem un utilizator înregistrat, ne dorim, în mod natural, să îi dăm posibilitatea de a se autentifica şi de-autentifica oricând îşi doreşte.

Din fericire, Django ne permite să implementăm acest comportament în doar doi paşi simpli. Astfel, pentru autentificare, creăm un formular şi un fişier *HTML* care vor conține câmpurile necesare pentru introducerea numelui de utilizator şi a parolei (bineînțeles, cu tot cu aspectul general al paginii), iar în fişierul *urls.py* al aplicației Django în care am lucrat până acum în această secțiune, adăugăm un nou element în lista *urlpatterns*. Acest nou element va arăta astfel: `url(r'^login/$', auth_views.login, {'template_name': 'register/login.html', 'authentication_form':LoginForm}, name='login')`, unde primul parametru reprezintă expresia regulată corespunzătoare adresei URL a paginii, cel de-al doilea parametru reprezintă funcția care va fi apelată şi care va realiza autentificarea utilizatorului, al treilea parametru conține un dicționar în care specificăm fişierul HTML creat anterior şi formularul de login corespunzător, iar ultimul parametru este numele asociat rutei, astfel încât să fie uşor referențiat din interiorul funcțiilor.

În linia de cod de mai sus, toți parametrii ne sunt cunoscuți cu excepția celui de-al doilea. Funcția *login* apelată este inclusă în Django, în modulul *django.contrib.auth.views*, redenumit de noi (folosind cuvântul cheie *„as”*) *auth_view* pentru a evita confuzia cu fişierul *views.py* prezent în aplicația Django curentă. Mai mult, era posibil să evităm al treilea parametru, caz în care Django ar fi folosit un template implicit pentru crearea paginii. Pentru a păstra totuşi un design consistent în proiect, am ales să implementăm un template propriu.

Analog, pentru implementarea funcționalității de de-autentificare, adăugăm în lista din fişierul *urls.py* elementul: `url(r'^logout/$', auth_views.logout, {'template_name': 'register/logout.html'}, name='logout')`, unde fişierul HTML indicat în parametrul al treilea, conține doar un mesaj de confirmare şi o legătură către pagina principală.

4.4.3. Paginarea şi căutarea

Prezentarea listei de elemente împărțită în mai multe pagini, comparativ cu afişarea întregii liste în aceeaşi pagină, nu aduce nici un beneficiu din punct de vedere funcțional.

Diferența apare doar la nivel de design. Într-adevăr, este mult mai uşor pentru utilizator să navigheze prin aplicație schimbând pagini, decât să fie nevoit să dea scroll

(să navigheze în josul paginii) până ajunge la articolul dorit. De exemplu, în primul caz, utilizatorul se va afla în orice moment la o distanță maximă cunoscută atât de partea de sus a aplicației, cât şi de cea de jos, astfel asigurându-ne că acesta are acces la toate meniurile şi funcționalitățile existente cu un efort cât mai mic. De asemenea, având tot conținutul pe o singură pagină, putem crea senzația de copleşire cu informații pentru utilizator, atunci când acesta navighează printre articole.

Având în minte toate aceste considerente, vom folosi clasa *Paginator* oferită de Django în modulul *django.core.paginator* pentru a crea acest element. Procesul este cât se poate de simplu, tot ce trebuie să facem este ca în fişierul *views.py* să salvăm într-o variabilă toate elementele pe care vrem să le afişăm pe pagină (vom considera cazul articolelor de ştiri), să luăm numărul paginii din metoda *GET* a requestului şi să instanțiem un obiect de tipul *Paginator* care va primi ca parametrii în constructor, lista de elemente ce vor fi paginate şi numărul de pagini. Apoi, într-o structură *try...except* vom apela metoda *page* a obiectului *Paginator* cu parametrul numărul paginii luată din requestul *GET* şi salvăm rezultatul în variabila ce va reprezenta toate articolele de ştire în interiorul template-ului *HTML*.

În template, vom adăuga o logică folosind limbajul de template Django prin care vom afişa drept active sau inactive butoanele „înapoi” sau „înainte” atunci când pagina curentă este prima sau ultima şi vom oferi un aspect special butonului ce reprezintă pagina curentă. Fiecare buton al paginării, va fi un element <*a*> în interiorul unei liste neordonate. Un astfel de element are următoarea structură `<a href="?page={{ i }}">{{ i }}</a></li>`, unde *i* este numărul paginii. Este important să observăm adresa atributului *href*, acest mod de scriere, va permite ca atunci când apăsăm butonul unei pagini, request-ul va conține un câmp numit *page* care va conține numărul paginii *i*.

Un alt element care uşurează utilizarea aplicației de către utilizator îl reprezintă căutarea. Spre deosebire de paginare, care a fost aleasă din considerente pur estetice, căutarea poate fi considerată o funcționalitate în plus.

Asemeni paginării, căutarea va reduce senzația de copleşire cu informații a utilizatorului şi, în plus, îi va permite să se întoarcă la un anumit articol oricând îşi doreşte, ceea ce creşte şansele ca acesta să se întoarcă în aplicație.

Pentru implementarea acestei funcționalități, este necesară adăugarea unui câmp de intrare în care va fi introdus textul căutării şi un buton care va porni această căutare. Vom adăuga aceste elemente în bara de navigare, astfel încât să fie disponibilă din orice punct al aplicației.

Apăsarea butonului va efectua un request de tip *GET* în interiorul căruia, vom avea textul introdus în câmpul text alăturat. Vom accesa această valoare prin metoda

request.GET.get('nume_textbox'), unde primul *get* reprezintă tipul requestului, iar cea de-a doua este o metodă care ne permite să accesăm un element în funcție de numele acestuia, *nume_textbox* este valoarea atributului *name* a elementului HTML în care introducem textul de căutare.

Dacă în câmpul de căutare este găsit un text la momentul requestului *GET*, atunci vom trata acest caz drept o încercare de căutare a unui articol. Pentru a realiza căutarea unei informații în baza de date, Django ne pune la dispoziție o metodă foarte simplă şi rapidă.

Primul pas constă în crearea unui vector de căutare care va conține coloanele din baza de date implicate în acest proces. În cazul nostru, vom alege coloanele care conțin titlul, descrierea şi textul articolului şi numele tagurilor. Acest vector va rezulta din concatenarea mai multor obiecte de tipul *SearchVector*.

Apoi, luăm toate datele din baza de date adnotăm aceste obiecte cu vectorul nostru de căutare pentru a indica ce coloane sunt implicate în acest proces şi filtrăm rezultatele oferind parametrului *search,* un obiect de tip *SearchQuery* care primeşte ca parametru textul căutării.

În final, adăugăm paginare pentru rezultatele acestei filtrări şi afişăm aceste date pe pagină.

4.4.4. Pagini modale

Am discutat până acum despre cum sunt sumarizate şi afişate toate articolele de ştiri pe pagina principală, însă, nu am spus nimic despre cum pot fi vizualizate de către utilizatori. Bineînțeles, conceptul este trivial şi nu necesită multă dezbatere, însă pentru a oferi un design mai plăcut aplicației, dar şi pentru a îmbunătăți viteza de afişare a articolelor selectate, am ales folosirea paginilor modale.

O pagină modală reprezintă o pagină care în loc să se încarce peste conținutul pe care îl avem deja pe pagină, se încarcă deasupra acestuia, într-o porțiune numită pop-up. Acest tip de componentă, îi va permite utilizatorului să deschidă un articol de ştire şi să îl închidă fără să se reîncarce pagina de pornire. Utilizând această metodă, nu numai că obținem un aspect mai plăcut, dar datorită faptului că pagina de pornire nu este reîncărcată atunci când revenim la ea de la un articol, vom crea impresia de rapiditate din perspectiva utilizatorului.

Pentru a crea o pagină modală, vom crea o rută în fişierul *urls.py* care va conține în calea sa id-ul articolului, apoi vom crea o funcție în fişierul *views.py* pe care o vom apela atunci când va fi accesată această rută.

Funcția va primi ca parametru cheia primara a articolului specificată în rută, iar cu această cheie, va selecta articolul de ştire şi tagurile asociate acestuia. De asemenea, vom extrage numele publicației de la care am preluat articolul din adresa URL. Apoi, pasăm toate acestea către template.

În template, vom folosi clase CSS corespunzătoare creării unei pagini modale precum *modal-head*, *modal-body*, *modal-title*, *modal-footer* s.a., puse la dispoziție de Bootstrap. Denumirile acestor clase sunt suficient de sugestive, astfel că vom presupune evident modul în care le-am folosit pentru a afişa informațiile extrase de funcția descrisa anterior.

Mai mult, pentru a oferi posibilitatea de a împărtăşi un articol şi cu alte persoane, vom adăuga în această pagină, un buton de „share" (distribuire) pe platforma Facebook. Acest lucru se realizează foarte uşor, fiind necesară doar crearea unui link a cărui legătură să fie `http://www.facebook.com/sharer/sharer.php?u=http://127.0.0.1:8000/shared/{{ news.id }}`, unde `{{ news.id }}` este o variabilă în limbajul de template Django şi reprezintă ID-ul articolului care va fi distribuit. Bineînțeles, adresa `127.0.0.1:8000`, va fi înlocuită cu numele domeniului atunci când vom publica această aplicație.

4.4.5. Adăugarea imaginilor

O imagine face cât o mie de cuvinte. Această frază de la începutul secolului XX este poate una dintre cele mai cunoscute din lume, iar cu ea în minte, ne vom orienta către pagina noastră de bloguri. Într-adevăr, un articol care conține doar text, nu atrage atenția şi, astfel, multe idei valoroase pot rămâne ignorate.

Pentru a evita acest impediment, vom permite adăugarea de imagini asociate articolelor de blog.

Modificările principale apar în fişierul *models.py* în care este descris modelul tabelei în care vor fi păstrate toate articolele de blog. Astfel, vom adăuga următoarele coloane:

- Coloana *blog_image* reprezentată printr-un obiect de tipul *models.ImageField()*. Acest câmp va primi ca parametrii calea directorului în care vor fi salvate imaginile pe server, o imagine implicită pentru cazurile în care utilizatorul alege să nu completeze acest câmp (acest parametru nu este obligatoriu, dar noi îl vom folosi pentru a păstra un aspect consistend în aplicație), doi parametrii *null* şi *blank* care descriu dacă adăugarea imaginii este obligatorie sau nu (noi vom alege valoarea *True* pentru ambele, astfel încât să se poată adăuga un articol fără

o imagine asociată) şi încă doi parametrii care indică numele coloanelor corespunzătoare înălțimii, respectiv lățimii imaginii.

- Coloana *height_field* este de tipul *models.IntegerField()* şi indică înălțimea imaginii.
- Coloana *width_field* este de tipul *models.IntegerField()* şi indică lățimea imaginii.

Acum că am pregătit tabela din baza de date pentru primirea imaginilor, ne mai rămâne doar să actualizăm formularul de adăugare a articolelor de blog. Astfel, în subclasa *Meta* a clasei formularului nostru, inițializăm câmpul *widgets* cu un dicționar care conține o singură pereche cheie-valoare: numele coloanei în care este găsită imaginea (*blog_image*) şi obiectul *forms.FileInput()*. Apoi adăugăm câmpul în fişierul de template într-un mod similar celorlalte câmpuri, iar în fişierul *views.py*, atunci când construim obiectul formularului nostru, specificăm şi parametrul *request.FILES* în constructor, pentru a avea acces la fişierele trimise în requestul *POST*.

4.5. Modelul bazei de date

4.5.1. Ce este ORM?

Să presupunem că dezvoltăm o aplicație care se va lega de o bază de date. În mod natural, ne dorim să adăugăm, modificăm şi extragem informații din acea bază de date direct din codul nostru. Pentru proiecte cu complexitate mică, acest lucru este realizabil fără prea mari probleme. Dar ce se întâmplă când lucrăm folosind paradigma programării orientate pe obiecte (POO) şi creăm un obiect *Utilizator* care conține un atribut *adresă* reprezentat tot printr-un obiect? Atunci când vom încerca să adăugăm informațiile câmpurilor obiectului nostru *Utilizator*, ne vom lovi de o problemă de compatibilitate.

Într-adevăr, într-o bază de date, datele sunt stocate în tabele sub formă de scalari simpli (tipuri numerice, text), iar noi încercăm să adăugăm un obiect. Acest lucru nu este posibil.

Pentru a rezolva această problemă, avem două posiblități: ori convertim câmpul *adresă* în grupuri de informații scalare înainte de a le adăuga în baza de date (şi bineînțeles, trebuie să avem o posibilitate de a recrea obiectul inițial plecând de la baza de date), ori rescriem tot codul astfel încât să corespundă restricțiilor impuse.

Un sistem ORM tratează prima variantă.

Prin urmare, ORM (Object-Relational Mapping) reprezintă o tehnică de programare prin care sunt convertite tipuri de date incompatibile folosind paradigma programării orientate pe obiecte. În alte cuvinte, un ORM este o librărie care încapsulează codul necesar manipulării bazei de date astfel încât programatorul să nu mai fie nevoit să scrie instrucțiuni SQL.

Avantajele unui astfel de sistem sunt:

- Timpul de dezvoltare a aplicațiilor scade deoarece ORM-ul rezolvă multe lucruri în mod automat (conexiunea la baza de date, construirea instrucțiunii SQL, executarea acesteia şi, dacă este cazul, returnarea rezultatului, pot fi restrânse la apelul unei singure metode). De asemenea, acest sistem ne permite să evităm repetarea codului şi ne constrânge în a folosi o arhitectură de tip MVC (ceea ce implică un cod mult mai „curat" şi mai uşor de întreținut).
- Un ORM oferă un plus de flexibilitate în dezvoltarea aplicației deoarece abstractizează sistemul de baze de date şi astfel, putem oricând să îl schimbăm. Mai mult, ne permite să folosim concepte din POO precum moştenirea.
- Previne greşelile de cod SQL.

Dezavantajele unui astfel de sistem sunt:

- Abstractizarea excesivă a bazei de date poate reprezenta un dezavantaj, în special pentru programatorii începători (sau fără cunoştințe de baze de date), deoarece este foarte uşor pentru ORM să construiască interogări foarte grele din punct de vedere computațional folosind doar câteva linii de cod (de exemplu, o interogare complexă inclusă într-un ciclu repetitiv) sau se pot defini relații între tabele care nu sunt normalizare corespunzător.
- Cum un ORM este doar o librărie, va fi necesară învățarea acesteia, iar înlocuirea cu un alt ORM va implica reluarea procesului de învățare.
- Deşi din punct de vedere al performanței, un sistem ORM este suficient de bun, un expert SQL va putea, în cele mai multe cazuri, să construiască interogări mult mai eficiente.

Exista multe exemple de astfel de sisteme ORM pentru majoritatea limbajelor de programare care suportă programarea orientată pe obiecte. Câteva exemple pot fi: Hibernate (Java), Propel (PHP), Django ORM şi SQLAlchemy (Python). În mod evident, pentru a gestiona baza de date din aplicația noastră, vom alege sistemul Django ORM.

4.5.2. Vizualizarea bazei de date

Pentru a ne fi mai uşor să înțelegem structura tabelelor din baza noastră de date şi motivele din spatele alegerilor făcute, ne este de ajutor să analizăm diagramele acestui model.

Este important de notat faptul că, atât pentru vizualizare, cât şi în restul discuției despre modelul bazei de date, vom trata numai tabelele care au un impact în aplicația noastră, întrucât folosind framework-ul de autentificare pus la dispoziție de Django, avem multe alte tabele care, deşi în mod normal sunt foarte utile, pentru simplitate, noi le-am evitat (de exemplu, tabele de permisiuni, de sesiuni etc).

Una dintre cele mai folosite metode de vizualizare a structurii unei baze de date o reprezintă diagrama entitate-relație (diagrama E/R). Aceasta a fost prima dată descrisă de Peter Chen în anul 1976 printr-o lucrare intitulată „The Entity-Relationship Model: Toward a Unified View of Data" („Modelul Entitate-Relație: către o vizualizare unificată a datelor).

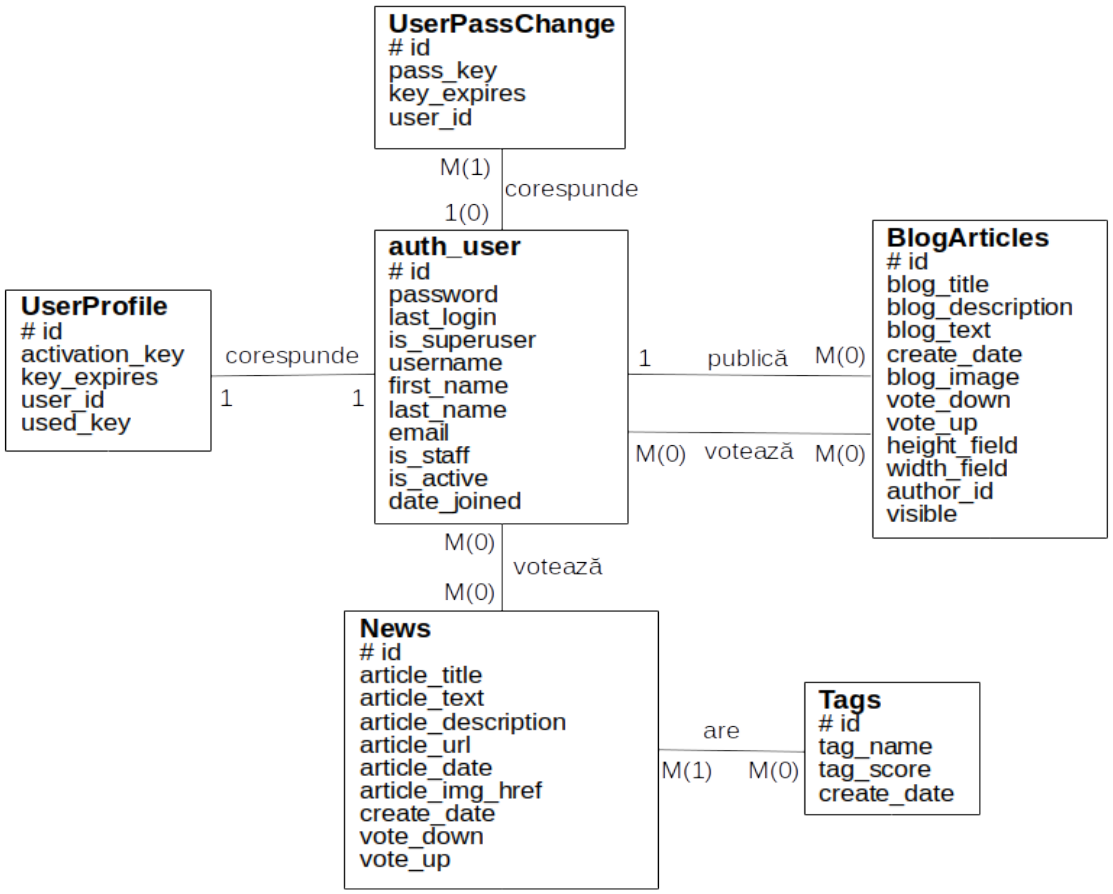

Figura 2. Diagrama E/R

După cum se poate deduce şi din nume, acest tip de diagramă ilustrează relațiile dintre entitățile modelului nostru. Atunci când vom vrea să construim modelul bazei de date, entitățile vor deveni tabele, iar relațiile vor deveni fie tabele de legătură (în cazul relațiilor de tip *mai mulți-la-mai mulți*), fie coloane speciale care referă cheia primară a celeilalte entități din relație.

O astfel de reprezentare devine foarte utilă în cazuri precum modelarea inițială a bazei de date (şansele de a implementa o logică greşită sau de a omite entități/relații importante din baza de date scad considerabil utilizând această diagramă) sau analiza problemelor apărute în logica unei baze de date deja existente.

În cazul aplicației noastre de sumarizare, diagrama entitate-relație arată precum în Figura 2.

Un alt tip de vizualizare a modelului bazei noastre de date îl reprezintă diagrama conceptuală. Acest tip de diagramă derivă din diagrama E/R, incluzând totuşi tabelele de legătură şi marcând cheile străine.

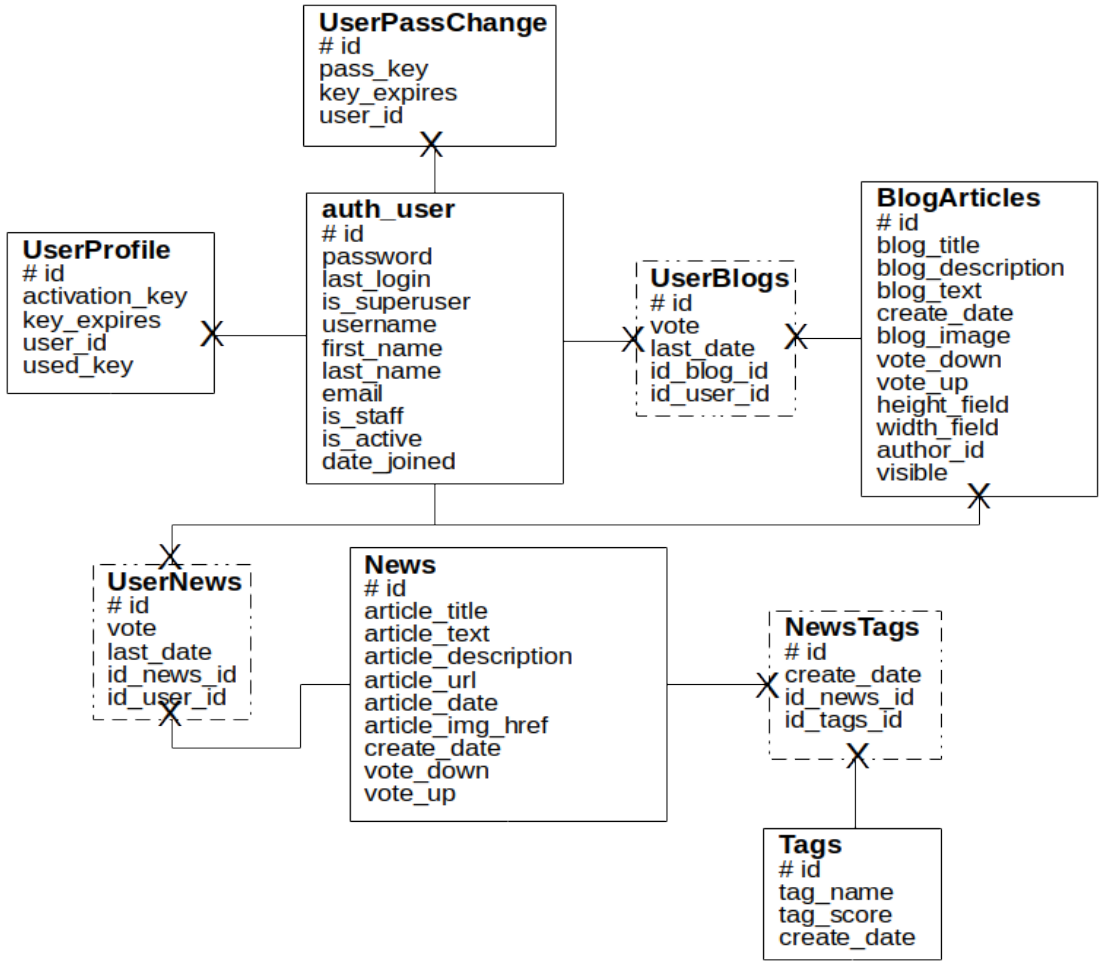

Figura 3. Diagrama conceptuală

Astfel, reprezentând prin linie punctată tabelele de legătură şi printr-un *X* faptul că în acea tabelă este inclusă cheia străină a celeilalte entități din relație, vom obține diagrama conceptuală din Figura 3.

Observăm din aceste reprezentări, faptul că tabela utilizatorilor va juca cel mai important rol în aplicația noastră, iar tabela de bloguri va avea două relații independente cu tabela utilizatorilor. Detaliile despre toate aceste decizii vor fi discutate în secțiunile care urmează.

4.5.3. Tabelele şi relațiile dintre acestea

Tabela AUTH_USER, deşi nu reprezintă funcționalitatea principală a aplicației, este totuşi cea mai importantă din punct de vedere al impactului asupra întregii scheme. După cum se poate observa şi din diagramele anterioare, exceptând tabela tagurilor, toate celelalte tabele sunt în relație cu aceasta.

Mai mult, această tabelă a fost creată în mod automat de către Django atunci când am folosit funcționalitatea de autentificare implicită.

Această tabelă conține atât informații specifice fiecărui utilizator cât şi informații generale despre permisiunile şi activitatea acestora. Coloanele tabelei sunt următoarele:

- *id*: cheia primară a tabelei;
- *password*: parola asociată contului;
- *username*: numele unic de utilizator;
- *first_name*, *last_name*: prenumele şi numele de familie al utilizatorului;
- *email*: adresa de email;
- *is_superuser*: un câmp boolean care indică dacă utilizatorul are toate drepturile asupra întregii aplicații;
- *is_staff*: tot un câmp boolean care indică dacă utilizatorul are dreptul să se autentifice în interfața de administrare a aplicației, dar nu are drepturi depline în aceasta (în proiectul nostru nu am inclus o astfel de interfață întrucât nu exista un concept de ierarhie care să o impună);
- *is_active*: indică dacă utilizatorul se poate autentifica în aplicație (am folosit această coloană pentru a opri accesul utilizatorilor noi până la accesarea link-ului de activare a contului trimis prin e-mail);

- *last_login*: amprentă temporală pentru când a avut loc ultima autentificare a utilizatorului în aplicație;
- *date_joined*: amprentă temporală pentru când a fost creat contul.

Constrângerile tabelei AUTH_USER sunt: UNIQUE (unic) pentru coloana *username*, PRIMARY KEY (cheie primară) pentru coloana *id*, şi NOT NULL pentru toate coloanele cu excepția coloanei *last_login*.

Tabelele USERPROFILE şi USERPASSCHANGE sunt foarte similare din punct de vedere al conținutului, dar diferite din punct de vedere al utilității şi relației acestora cu tabela utilizatorilor. Astfel, USERPROFILE este responsabilă cu stocarea cheii de activare a conturilor, iar USERPASSCHANGE va păstra cheia pentru schimbarea parolei.

Coloanele *activation_key* şi *pass_key* conțin cheile de activare, respectiv de schimbare a parolei, coloana *key_expires* indică data la care cheia va expira, *user_id* reprezintă cheia străină către tabela utilizatorilor, iar în cazul tabelei USERPROFILE, este inclusă şi coloana *used_key* care indică dacă acea cheie a fost utilizată sau nu pentru activarea contului utilizatorului.

După cum putem observa şi din diagrama entitate-relație, USERPROFILE se află în relație de *unu-la-unu* cu tabela AUTH_USER, iar USERPASSCHANGE şi AUTH_USER sunt în relație de *unu-la-mai mulți*.

Constrângerile impuse pentru coloanele acestor tabele sunt: PRIMARY KEY pentru coloana *id*, FOREIGN KEY (cheie străină) pentru coloana *user_id* şi NOT NULL pentru toate celelalte coloane. De asemenea, în cazul tabelei USERPROFILE, se va crea şi o constrângere UNIQUE aplicată coloanei *user_id* pentru a asigura relația de *unu-la-unu*.

Tabela NEWS va păstra toate ştirile sumarizate de algoritmul nostru şi câte voturi pozitive şi negative au fost înregistrate pentru fiecare articol.

Coloanele au o denumire sugestivă, astfel că *article_title*, *article_text*, *article_description*, *article_url*, *article_date*, *article_img_href* conțin informațiile pentru titlul, conținutul, descrierea, adresa url, data publicării, respectiv adresa url a imaginii articolului. Coloana *create_date* indică momentul de timp în care a fost creată fiecare înregistrare în baza de date, iar *vote_up* şi *vote_down* indică numărul de voturi pozitive, respectiv negative asociate articolului până în acel moment.

Din diagramele anterioare observăm că această tabelă se află în relație *mai mulți-la-mai mulți* cu tabelele AUTH_USER şi TAGS. Această relație ne va impune crearea a câte unui tabel de legătură pentru fiecare apariție a sa.

Astfel, relația dintre tabela utilizatorilor şi cea a ştirilor, va fi împărțită în două relații de *unu-la-mai mulți* între AUTH_USER şi USERNEWS, respectiv între NEWS şi USERNEWS. Tabela de legătură USERNEWS, va conține tipul votului fiecărui utilizator care a votat cel puțin o dată (*1* dacă este un vot pozitiv şi *-1* dacă este unul negativ), amprenta de timp la care a fost înregistrat ultimul vot (pentru articolul şi utilizatorul în cauză) şi cheile străine către tabelele utilizatorilor şi ştirilor.

În mod similar, relația dintre tabela NEWS şi tabela TAGS se va transforma în două relații *unu-la-mai mulți* între cele două tabele şi NEWSTAGS. Această tabelă de legătură conține data creării fiecărei asocieri dintre tag şi articol şi cheile străine către înregistrările corespunzătoare din tabelele de ştiri şi de taguri.

Constrângerile asupra coloanelor tabelei NEWS sunt: PRIMARY KEY pentru coloana *id* şi NOT NULL pentru toate celelalte coloane. Constrângerile tabelei USERNEWS sunt: PRIMARY KEY pentru coloana *id*, NOT NULL pentru toate celelalte coloane şi FOREIGN KEY pentru coloanele *id_user_id* şi *id_news_id*. În mod similar, tabela NEWSTAGS va avea constrângerile: PRIMARY KEY pentru coloana *id*, FOREIGN KEY pentru coloanele *id_news_id* şi *id_tags_id* şi NOT NULL pentru toate coloanele.

Tabela TAGS va păstra toate tagurile generate de algoritmul de generare prezentat într-o secțiune separată a acestei lucrări.

Coloanele sale vor indica numele etichetei (*tag_name*), punctajul asociat acesteia (*tag_score*) şi amprenta temporală la care a fost creată înregistrarea (*create_date*). Constrângerile asupra acestor coloane sunt: PRIMARY KEY pentru coloana *id* şi NOT NULL pentru toate celelalte coloane.

De asemenea, această tabelă se află în relație cu tabela NEWS, iar detaliile acestei relații au fost prezentate mai sus.

Ultima tabelă rămasă în modelul nostru este tabela BLOGARTICLES care conține toate articolele de blog create de utilizatorii aplicației.

Coloanele acestei tabele sunt: coloanele care păstrează informațiile despre titlul, conținutul, descrierea şi imaginea asociată articolului de blog (*blog_title*, *blog_text*, *blog_description*, *blog_image*), data publicării articolului (*create_date*), numărul de voturi pozitive şi negative (*vote_up*, *vote_down*), înălțimea şi lățimea imaginii asociate articolului (*height_field*, respectiv *width_field*), id-ul utilizatorului care a publicat articolul (*author_id*) şi o coloană boolean care indică dacă articolul este vizibil sau nu în aplicație (*visible*).

Din punct de vedere al relațiilor, tabela blogurilor este mult mai interesantă decât celelalte tabele ale bazei de date, deoarece se află în relație cu tabela utilizatorilor în două moduri diferite.

În primul rând, este în relație *mai mulți-la-mai mulți* pentru a lega voturile asociate fiecărui articol, de utilizatorii care au efectuat votarea. Această relație ne va determina să construim un tabel de legătură numit USERBLOGS în mod similar celui creat în urma relației dintre tabela utilizatorilor şi cea a ştirilor. Astfel, tabelul de legătură va conține informații despre natura votului (pozitiv sau negativ), data ultimei modificări a acestuia şi cheile străine care leagă cele două tabele din relație.

Cea de-a doua relație, este de tipul *unu-la-mai mulți* şi indică utilizatorul care a publicat articolul de blog (cu alte cuvinte, autorul articolului).

Constrângerile tabelei BLOGARTICLES sunt: PRIMARY KEY pentru coloana *id*, FOREIGN KEY pentru *author_id* şi NOT NULL pentru toate celelalte coloane. În cazul tabelei de legătură, vom avea PRIMARY KEY pentru coloana *id*, FOREIGN KEY pentru coloanele *id_blog_id* şi *id_user_id* şi NOT NULL în rest.

4.5.4. Modele şi migrări în Django

Acum că avem structura bazei de date bine definită, este timpul să o implementăm. Totuşi, construirea acesteia în mod direct folosind soluția SQL aleasă, deşi suficient de simplă la început, poate deveni greu de întreținut pe termen lung. Mai mult, dacă dezvoltatorul nu are suficientă experiență cu această tehnologie, soluțiile sale pot avea un impact negativ asupra modelului, putându-se ajunge la un punct în care revenirea la starea inițială să fie foarte costisitoare.

Pentru a evita o astfel de problemă, Django ne oferă o soluție proprie de ORM numită sugestiv Django ORM. Aşa cum am discutat mai sus, un ORM ne permite să efectuăm modificări la nivelul modelului de baze de date folosind paradigma programării orientate pe obiecte implementată într-un limbaj de programare cunoscut. În acest fel, dezvoltatorul se poate concentra doar pe limbajul de programare în care este construită aplicația.

În Django, orice tabelă este reprezentată de o clasă care extinde clasa *Model* din modulul *django.db.models*. Această clasă va fi adăugată în fişierul *models.py* din interiorul aplicației Django în care are cel mai mult sens să fie creată. Într-adevăr, putem importa această clasă şi din alte aplicații Django din interiorul proiectului, dar pentru a păstra un curs logic de-a lungul proiectului, este indicat să fie inclusă în aplicația în care modelul este accesat pentru prima dată.

Coloanele vor fi reprezentate prin atributele clasei corespunzătoare modelului. Django ne pune la dispoziție un număr de tipuri posibile pentru aceste coloane, toate reprezentând clase în modulul *django.db.models*, precum: *CharField* indică un câmp text de dimensiune mai mică de 255 de caractere, *TextField* indică un câmp text de dimensiuni mari, *IntegerField* reprezintă o coloană de tip numeric, *BooleanField* indică un câmp care poate avea doar două opțiuni (adevărat sau fals), *DateTimeField* indică un câmp care conține data şi timpul (similar, *DateField* va descrie o coloană ce conține doar data), şi multe altele.

Pentru a crea constrângerile, Django oferă un mix de câmpuri şi opțiuni de câmpuri care adresează această nevoie. Astfel, pentru a crea o cheie primară, este nevoie să creăm un câmp folosind una din clasele de mai sus (sau una similară), căreia să îi precizăm în constructor parametrul *primary_key=True* (dacă în modelul nostru nu există nici un astfel de argument, Django va crea automat un câmp de tipul *AutoField* care să păstreze cheia primară şi îl va numi *id*). În mod similar, pentru a adăuga constrângerile *unique* şi *not null*, vom preciza în constructorul câmpului nostru *unique=True*, respectiv *null=False*. Este de reținut totuşi faptul că, Django va adăuga constrângerea *not null* în mod implicit tuturor coloanelor, iar pentru a suprascrie acest comportament, este necesară precizarea în constructorul câmpului a parametrului *null=True*.

Constrângerea de cheie străină corespunde unui tip de câmp special, numit *ForeignKey* care primeşte în constructor clasa modelului pe care îl referă şi comportamentul în cazul ştergerii datelor (în general este aleasă opțiunea de ştergere în cascadă). Mai mult, precizarea acestui câmp va crea în mod automat o relație de tipul *unu-la-mai mulți* cu tabela referețiată.

Pentru crearea unei relații *unu-la-unu*, este necesar câmpul *OneToOneField* cu aceiaşi parametrii ca în cazul cheii străine, iar pentru relația *mai mulți-la-mai mulți*, avem la dispoziție câmpul *ManyToManyField* care necesită numai clasa tabelei pe care o referă. De asemenea, la crearea unei relații *unu-la-unu*, va fi creată şi o constrângere *unique* pentru coloana implicată, iar în cazul relației *mai mulți-la-mai mulți*, Django va crea în mod automat un tabel de legătură între cele două entități aflate în relație.

O altă observație în legătură cu crearea automată a tabelelor de către framework constă în denumirea acestora. În mod implicit, Django va numi tabelele ce vor fi create şi adăugate în baza de date folosind ORM-ul propriu după cum urmează: *nume_aplicație_nume_clasa_model*, unde *nume_aplicație* reprezintă numele aplicației Django în care a fost definit modelul, iar *nume_clasa_model* este numele modelului în sine (scris cu litere mici). Pentru a suprascrie acest comportament, şi a oferi tabelelor din baza de date un nume ales de noi, este suficient să creăm o subclasă numită *Meta* în interiorul clasei modelului țintă, iar în interiorul său să adăugăm atributul

db_table=”nume_tabel”, unde stringul *nume_tabel* reprezintă denumirea tabelului care va fi salvat în baza de date.

Din această ultimă observație, putem deduce imediat faptul că denumirile clasei care descrie modelul şi cea a tabelei salvate în baza de date, pot fi diferite, iar de-a lungul aplicației, orice tip de cerere către baza de date se va face prin intermediul claselor care descriu tabelele implicate. Django ORM îşi va da în mod automat seama către ce tabelă din baza de date să trimită cerințele şi cum să transforme cerința noastră în cod SQL.

Perfect. Tot ce mai rămâne de făcut este să propagăm aceste modificări către baza de date. Pentru a realiza acest lucru, Django introduce un sistem propriu numit *„migrări”*.

Migrările au drept scop propagarea automată a modificărilor de la modelul descris în aplicația Django, la tabela corespunzătoare din baza de date. Cu toate acestea, sistemul îi va ceda dezvoltatorului responsabilitatea de a crea şi de a executa migrările.

Pentru a crea migrările pentru orice modificare a modelelor, vom deschide terminalul în directorul întregului proiect (cel care conține toate aplicațiile şi fişierul *manage.py*) şi vom executa comanda `python manage.py makemigrations`. Odată create migrările, le vom aplica asupra bazei de date folosind comanda `python manage.py migrate`. Astfel, cu doar două comenzi, ne asigurăm că baza de date are structura reprezentată prin modelele noastre.

De asemenea, Django pune la dispoziție încă două comenzi speciale pentru a uşura lucrul cu migrările şi pentru a-i oferi dezvoltatorului o modalitate de a depista din timp eventualele erori în structura descrisă de prin modele. Prima astfel de comandă este *sqlmigrate* (rulată în mod similar cu cele două comenzi de mai sus). Ea va afişa comenzile SQL care urmează să fie executate la rularea comenzii *migrate*. Cea de-a doua comandă este *showmigrations* responsabilă cu afişarea tuturor migrărilor din proiect împreună cu statusul lor.

Pentru o înțelegere deplină a întregului concept de migrări, Django ne îndeamnă să ne gândim la acesta ca la un sistem de control al versiunilor[2] precum *Git*. Astfel, comanda *makemigrations* va crea fişiere speciale care vor conține toate modificările ce urmează să fie aduse bazei de date (scrise tot în Python, dar urmând un stil standardizat), analog comenzii *commit* din Git, iar comanda *migrate* va propaga modificările în baza de date, asemeni comenzii *push*.

Mai mult, datorită caracterului automat al migrărilor, Django ne asigură că schema rezultată va fi întotdeauna aceeaşi indiferent de câte ori am efectua migrările

2 *Vezi pagina din documentaţia oficială: https://docs.djangoproject.com/en/2.0/topics/migrations/*

(atenție totuşi că nu ne este asigurată şi calitatea migrărilor, de aceea există comenzile de gestionare *showmigrations* şi *sqlmigrate*). Astfel, mai mulți dezvoltatori pot lucra la acelaşi proiect şi pot rula migrările create de altcineva şi au garanția că baza lor de date locală va arăta la fel.

4.5.5. Alegerea sistemului de baze de date

Mai sus am stabilit ce entități vor fi adăugate în baza de date, ce relații vor fi între acestea şi modalitatea prin care vom transforma aceste modele în tabele ale bazei de date. A mai rămas un singur lucru de decis: ce bază de date vom folosi?

Într-adevăr, există multe sisteme de gestionare a bazelor de date, fiecare cu avantajele şi dezavantajele sale. Însă pentru aplicația noastră, nu ne interesează atât de mult calitățile generale ale acestora, ci, mai degrabă, compatibilitatea lor cu framework-ul Django. În acest fel, ne asigurăm că avem la dispoziție cât mai multe metode de gestionare automată oferite de Django.

Având în minte toate acestea, vom alege pentru aplicația noastră sistemul de gestionare PostgreSQL. PostgreSQL (sau simplu Postgres) este un sistem de gestionare al bazelor de date obiect-relație (ORDBMS), gratuit şi open-source, care pune accent pe extensibilitate şi conformitatea cu standardele.

Postgres îşi are originile în anul 1986 când Michael Stonebraker a luat decizia de a-şi dezvolta propriul sistem de gestionare al bazelor de date cu scopul de a adresa problemele sistemelor existente în acea perioadă. Mai mult, datorită impactului funcționalităților introduse în acest sistem de gestionare asupra domeniului, autorul a primit premiul Turing în anul 2014.

Aşa cum am precizat mai sus, deşi Django nu favorizează nici un sistem de gestionare al bazelor de date, datorită numărului impresionant de funcționalități puse la dispoziție de Postres, combinarea celor două devine tot mai interesantă.

Astfel, Postgres oferă cel mai bun suport pentru alterarea schemei (important în procesul de migrări) comparativ cu celelalte sisteme de gestionare suportate de Django, schimbarea tipului de date a unei coloane se poate realiza fără rescrierea întregului tabel, pentru aplicații care folosesc hărți, Django are suport numai pentru Postgres etc. Bineînțeles, nici PostgreSQL nu este perfect. Un inconvenient ar putea fi că adăugarea unei coloane care are setată o valoare implicită va rezulta într-o rescriere a întregului tabel ceea ce poate deveni o problemă atunci când tabelul este de dimensiuni foarte mari.

Conectarea unei baze de date de aplicația noastră se poate realiza foarte simplu din fişierul *settings.py* al aplicației principale, unde trebuie precizat următorul dicționar:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'newssummy',
        'USER': 'postgres',
        'PASSWORD': 'parola123',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

, unde *ENGINE* specifică sistemul de gestionare pe care îl vom folosi pentru crearea şi administrarea tabelelor din baza de date (observăm că valoarea acestuia este un string care indică un modul Django specific fiecărui sistem), *NAME* reprezintă numele bazei de date (deja existentă), *USER* este numele utilizatorului care se va conecta la baza de date şi în numele căruia vom efectua toate acțiunile, *PASSWORD* include parola utilizatorului bazei de date, iar *HOST* şi *PORT* indică adresa şi portul serverului în care se găseşte baza de date.

Trebuie să fim atenți totuşi, pentru a realiza conexiunea la baza de date, aceasta din urmă trebuie să existe deja. Django ne asigură crearea şi întreținerea entităților din baza de date, dar nu şi baza de date în sine.

4.6. Securitatea aplicației

4.6.1. Securitatea formularelor

Să presupunem că am dezvoltat o aplicație web pentru o bancă care are drept scop gestionarea conturilor bancare ale fiecărui client. În mod natural, aplicația noasră va avea un număr de formulare pentru introducerea informațiilor necesare creării unui nou cont, transferului banilor de la un cont la altul şi multe alte astfel de operațiuni. Fiecare formular se va găsi la o anumită adresă URL şi va executa un request de tip POST atunci când se vrea a fi înaintat spre procesare.

Totul arată bine până acum, iar într-o lume perfectă putem spune că aplicația este gata să fie lansată în producție.

Totuşi, nu trăim într-o astfel de lume. Prin urmare, ne gândim la următorul scenariu: o persoană oarecare dezvoltă o altă aplicație web care include într-unul dintre butoanele sau link-urile sale un request POST către unul dintre formularele din aplicația noastră. În acest fel, dacă un utilizator autentificat deja în aplicația noastră, accesează acel buton/link, va declanşa o cerere în aplicație care va fi tratată ca şi cum utilizatorul însuşi ar fi trimis-o.

Folosind această metodă, un site malițios poate crea conturi bancare şi trimite bani către un alt cont în mod automat, fără acordul sau ştiința victimei.

Mai mult, exemplul prezentat este bazat pe un caz real, iar banca în cauză este ING Direct[3]. Problema a fost însă investigată şi (mai mult ca sigur) rezolvată până în momentul scrierii acestei lucrări.

Un astfel de atac poartă numele de *Cross-Site Request Forgery* (*CSRF*) şi este definit ca un tip de atac prin care un site malițios poate provoca modificări într-un alt site în numele unui utilizator „de încredere" pentru acel server. Acest atac poate folosi cookie-urile sau IP-ul utilizatorului pentru a determina browserul să creadă că acțiunea este executată chiar de utilizator.

Alte modificări pe care un atacator le poate aduce unei aplicații folosind acest tip de atac sunt:

- De-autentificarea utilizatorului de pe aplicația țintă;
- Modificarea preferințelor victimei pe pagina respectivă;
- Publicarea de conținut pe pagina respectivă în numele utilizatorului victimă (comentarii, mesaje de status, etc).

Aşadar, atacurile CSRF sunt cu adevărat periculoase şi ne dorim să facem tot posibilul să protejăm utilizatorii aplicației noastre de astfel de atacuri. Pentru a face acest lucru, avem două posibilități: verificăm header-ul pentru a vedea de unde a fost trimisă cererea (în header vom avea informații despre adresa URI care a trimis acea cerere), iar dacă acea cerere nu vine dintr-o sursă de încredere (în general, este o sursă diferită de adresa aplicației noastre), atunci blocăm cererea. A doua posibilitate este să includem un câmp ascuns în formularul nostru care să conțină o cheie secretă pe care o vom verifica mai apoi înainte de a înainta formularul.

3 Mai multe informaţii despre cum putea fi efectuat un astfel de atac atât pentru ING Direct cât şi pentru alte aplicaţii web precum YouTube, The New York Times sau MetaFilter pot fi găsite în acest articol: https://people.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf secţiunea 3, paginile 4-8.

Avantajul primei metode constă în uşurința cu care poate fi implementat, însă unii utilizatori poate au ales să modifice setările browser-ului astfel încât să se omită acest header sau să fie modificat (aceste modificări pot apărea din dorința utilizatorului de a-şi proteja intimitatea). Cea de-a doua metodă pe de altă parte, nu se confruntă cu o asfel de problemă şi poate asigura o securitate mai mare decât prima metodă, însă pot apărea probleme în cazul în care un utilizator alege să salveze pagina HTML într-o altă locație (dacă atacatorul are acces la acea locație, poate vedea cheia de protecție şi o poate folosi pentru un atac CSRF).

Django foloseşte cea de-a doua metodă pentru a asigura protecția formularelor împotriva unor astfel de atacuri. Implementarea acesteia implică următoarele acțiuni: crearea unui cookie CSRF bazat pe o cheie secretă generată în mod aleatoriu şi a cărei cheie se schimbă la fiecare autentificare a utilizatorului, includerea unui câmp ascuns în requestul POST care va conține valoarea secretă generată anterior, orice request HTTP care nu foloseşte una din metodele GET, HEAD, TRACE sau OPTIONS şi care nu conține cookie-ul CSRF şi câmpul ascuns „*csrfmiddlewaretoken*" discutate anterior va rezulta în blocarea requestului şi returnarea codului de eroare 403. Mai mult, pentru a evita cazurile în care atacatorul modifica cookie-urile utilizatorului pentru a putea demara atacul, Django va verifica adresa celui care a pornit requestul (acest lucru este necesar deoarece protocolul HTTPS nu oferă o astfel de protecție).

Pentru a asigura protecția formularelor din aplicația noastră împotriva unor atacuri CSRF, este suficient să adăugăm instrucțiunea în limbajul de template specific Django `{% csrf_token %}` imediat după tag-ul HTML `<form>`. Atenție totuşi la faptul că această protecție nu trebuie folosită pentru formulare care trimit requestul către o adresă externă aplicației noastre întrucât acea adresă va putea avea acces la cheia secretă (token-ul) utilizatorului, deci se compromite securitatea aplicației.

4.6.2. Alte elemente de securitate

Din moment ce aplicația noastră suportă conectarea mai multor utilizatori şi oferă funcționalități diferite în funcție de starea acestora (autentificați sau nu), se ridică problema limitării accesului utilizatorilor la anumite pagini pentru care nu sunt autorizați. De exemplu, un utilizator neautentificat nu trebuie să vadă pagina de adăugare de bloguri întrucât adăugarea unui articol din acea pagină implică o legătură cu tabela utilizatorilor. În mod similar, butoanele sistemului de votare ar trebui dezactivate atunci când utilizatorul nu este autentificat.

Cum ne-am obişnuit, Django ne oferă două modalități simple de a controla accesul în aplicație. Pentru a limita accesul pentru întreaga pagină (vezi exemplul 1 de

mai sus), este suficient să importăm decoratorul *login_required* din modulul *django.contrib.auth.decorators* şi să îl adăugăm deasupra funcției view (din fişierul *views.py*) corespunzătoare afişării paginii țintă. Folosind acest decorator, Django va redirecționa orice utilizator neautentificat care încearcă să acceseze acea pagină, către pagina de login.

Pentru exemplul 2, nu avem nevoie sa restricționăm accesul la întreaga pagină, ci doar la anumite funcționalități. Astfel, ne vom orienta atenția către template-ul care descrie aspectul paginii (şi conține elementele pe care ne dorim să le restricționăm). Folosind sintaxa `{% if request.user.is_anonymous %} elemente_dezactivate {%else%} elemente_activate {%endif%}`, unde *elemente_dezactivate* reprezintă codul HTML ce trebuie afişat pentru utilizatorii neautentificați, iar *elemente_activate* codul HTML pentru utilizatorii autentificați.

Este interesant de observat că, folosind limbajul de template pus la dispoziție de Django, avem acces la acelaşi obiect *request* ca cel din funcția de view. Astfel, legătura dintre backend şi frontend este mult mai strânsă, ceea ce se traduce prin mult mai mult control asupra funcționalităților aplicației pe care o dezvoltăm.

Având toate acestea, paginile restricționate din aplicația noastră sunt: pagina de adăugare, de editare şi de ştergere a blogurilor, pagina de vizualizare a blogurilor postate de un utilizator, funcțiile sistemului de votare (chiar dacă dezactivăm butonul, adresa URL care apelează funcția rămâne disponibilă). Funcționalitățile restricționate din template sunt: butoanele sistemului de votare (au fost dezactivate pentru utilizatori neautentificați) şi meniul vertical din dreapta bării de navigare (pentru utilizatori neautentificați va afişa opțiuni de autentificare şi înregistrare, iar pentru ceilalți, va afişa legături către pagina „blogurile mele” şi log out.

Un alt aspect important în legătură cu securitatea aplicației, îl reprezintă salvarea parolelor. Într-adevăr, este considerată a fi o practică rea şi este vehement descurajată de comunitatea dezvoltatorilor salvarea unei parole în baza de date fără nici o metodă de criptare sau hashing. Folosind sistemul de autentificare oferit de Django, ne asigurăm că parolele vor fi salvate sub formă de hash.

Algoritmul folosit în mod implicit este PBKDF2 (Password-Based Key Derivation Function 2) împreună cu o funcție hash SHA256. Această metodă ar trebui să fie suficientă pentru a asigura o securitate de nivel suficient de înalt pentru majoritatea aplicațiilor. Cu toate acestea, Django permite schimbarea algoritmului de hashing, adăugând un nou prim element în lista PASSWORD_HASHERS din fişierul *settings.py* (Django va alege primul element al listei drept algoritmul de hashing). Mai mult, toți algorimii fiind păstrați într-o listă, este posibil ca primul element al listei să fie utilizat pentru stocarea parolelor noi, iar ceilalți algoritmi pentru verificarea parolelor

salvate deja cu alți algoritmi. Cu alte cuvinte, această metodă ne permite să schimbăm algoritmul de hashing şi să păstrăm parolele securizate cu algoritmul vechi, iar Django va şti să le verifice.

De asemenea, Django oferă, în mod implicit, câteva modalități de verificare a validității parolei înainte de a fi creat un cont. Printre aceste validări se numără: verificarea dacă parola este foarte asemănătoare cu numele de utilizator, verificarea ca parola să fie formată din cel puțin 8 caractere, verificarea dacă toate caracterele sunt alfanumerice şi verificarea dacă parola nu este una foarte comună[4]. Mai mult, framework-ul ne permite crearea propriilor clase de validare sau modificarea proprietăților celor existente.

Toate clasele de validare sunt incluse într-o listă numită *AUTH_PASSWORD_VALIDATORS* din fişierul *settings.py* sub forma unor dicționare cu un singur element care are cheia *„NAME”* şi valoarea un string care conține calea către clasa de validare. Toate clasele implicite sunt definite în modulul *django.contrib.auth.password_validation*.

4 *Lista implicită folosită pentru verificarea dacă parola este una comună este creată de Mark Burnett şi poate fi găsită la adresa https://xato.net/passwords/more-top-worst-passwords/*

5. Îmbunătățiri ulterioare

În capitolele anterioare am prezentat toate tehnilogiile, funcționalitățile şi caracteristicile aplicației dezvoltate. Însă, aşa cum era de aşteptat, orice proiect are loc de îmbunătățiri, iar în cazul în care ne dorim să tratăm această aplicație drept un produs de comercializat, adăugarea anumitor funcționalități devine obligatorie pentru asigurarea succesului pe termen lung.

Astfel, în acest capitol ne vom uita la câteva îmbunătățiri şi funcționalități viitoare pe care ne-am dori să le includem în aplicația noastră, dar care, din diferite motive, nu au putut fi prezente la momentul scrierii acestei lucrări.

Prin îmbunătățiri, vom înțelege modificări aduse proceselor şi componentelor deja existente în aplicație cu scopul de a spori experiența utilizatorului şi de a asigura o mai bună stabilitate a aplicației.

Câteva îmbunătățiri pe care ne dorim să le aducem acestui proiect pot fi:

- Implementarea unui algoritm de etichetare automată mult mai eficient care să poată identifica în mod „corect” şi taguri formate dintr-un singur cuvânt (întrucât eliminarea acestora din algoritm poate afecta calitatea rezultatului). De asemenea, noua versiune a algoritmului ar trebui să fie capabilă să genereze taguri pentru orice articol (ne amintim că în implementarea discutată în capitolul patru, nu avem garanția creării a cel puțin unui tag pentru fiecare articol de ştire);
- O altă îmbunătățire foarte importantă vizează sistemul de autentificare. Mai exact, salvarea parolelor. Într-adevăr, parolele sunt salvate într-un mod foarte sigur, iar „spargerea” hashului unei parole ar necesita mult mai multă putere de procesare decât ar fi profitabil. Cu toate acestea, nu îl opreşte nimic pe atacator să încerce pur şi simplu parole la întâmplare până o află pe cea corectă. Din acest motiv, este necesară implementarea unor validări mult mai stricte asupra parolelor şi implementarea unui sistem care să împiedice introducerea greşită a parolei, în mod consecutiv, de mai mult de un număr n de ori (unde n este un număr ales de noi). O validare adițională ar putea fi obligarea utilizatorului de a folosi cel puțin o literă mare, o literă mică şi o cifră în crearea parolei. Iar pentru cazul în care parola este introdusă greşit de mai multe ori consecutiv, să fie obligat să introducă un cod captcha;

- Modificarea sistemului de votare astfel încât să folosească Ajax atunci când se înregistrează un nou vot. Utilizând această tehnologie, utilizatorul nu va mai experimenta reîncărcarea paginii de fiecare dată când votează un articol;
- Implementarea posibilității ca utilizatorul să anuleze un vot;
- Preluarea articolelor de ştiri de la mai multe publicații şi preluarea ştirilor din mai multe categorii.

Din punct de vedere al funcționalităților adiționale pe care ne dorim să le includem într-o versiune viitoare a aplicației, enumerăm următoarele:

- Adăugarea unei pagini de profil pentru fiecare utilizator, în care acesta să poată vedea toate articolele de ştiri sau de blog pe care le-a votat de-a lungul timpului (şi bineînțeles, posibilitatea de a modifica votul). Mai mult, tot din această interfață să putem oferi posibilitatea de a-şi schimba parola, numele de familie sau prenumele;
- Dacă îndeplinim primul punct din secțiunea de îmbunătățiri (cel care implică dezvoltarea unui algoritm de etichetare mai eficient), am putea adăuga posibilitatea de a genera taguri pornind de la textul introdus de utilizator în pagina de sumarizare a propriului text;
- Pentru a oferi utilizatorului cele mai importante ştiri ale momentului, va trebui să implementăm o metodă de a acorda fiecărui articol un anumit punctaj (invizibil din punct de vedere al utilizatorului) prin intermediul căruia să putem sorta articolele într-o modalitate mult mai relevantă importanței acestora. Câțiva factori relevanți în construirea unui astfel de punctaj pot fi: data publicării articolului, numărul de voturi pozitive şi negative oferite de utilizatori, subiectul articolelor (dacă există suficient de multe articole despre un anumit eveniment publicate în aceeaşi perioadă de timp, putem presupune că acel subiect tratează evenimente cu un impact ridicat în lume);
- Adăugarea unor metode de a sortare a articolelor. De exemplu, sortare în funcție de cele mai noi articole, de numărul de voturi, de un punctajul calculat prin metoda descrisă la punctul anterior. Mai mult, sortarea în funcție de acest punctaj ar putea deveni sortarea implicită a articolelor;
- O altă funcționalitate care poate nu are un impact atât de mare precum cele de mai sus, dar care poate îmbunătăți considerabil calitatea conținutului din aplicație o reprezintă adăugarea unei metode de raportare a conținutului necorespunzător. Cu alte cuvinte, utilizatorii să poată raporta un rezumat care nu a fost generat într-un mod corespunzător (de exemplu, nu are sens atunci când este citit), un articol de blog care nu respectă scopul inițial al acestei pagini (de

exemplu, este un articol despre viața personală a autorului, fără relevanță pentru evenimentele de actualitate), sau un articol de blog care promovează valori greşite (de exemplu, instigare la ură, extremism, rasism etc).

6. Concluzii

Internetul va continua să cuprindă tot mai mult din lumea în care trăim, iar informațiile cuprinse de acesta sunt atât de numeroase încât este aproape imposibil să ținem pasul cu tot ce se întâmplă în jurul nostru.

Într-adevăr, pentru mulți, internetul poate părea copleşitor, misterios, poate chiar înfricoşător. Însă acesta nu este decât o ustensilă, un mijloc de comunicare, o legătură între noi şi restul lumii. Cei care îşi perfecționează abilitatea de a utiliza această ustensilă vor avea toate mijloacele necesare spre a cunoaşte lumea din jur.

Aplicația noastră îşi propune să uşureze acest proces de informare oferind acces la rezumate ale articolelor scrise de publicații renumite, la opiniile altor persoane vis-a-vis de diferinte evenimente din lume sau posibilitatea de a rezuma propriul text (pentru cazurile în care informația dorită nu există pe site).

Mai mult, aşa cum am văzut de-a lungul lucrării, impactul utilizatorului asupra calității informației nu este deloc neglijat, ci dimpotrivă am permis votarea tuturor articolelor (atât de ştiri, cât şi de blog), am inclus un top al celor mai importante trei ştiri din ultimele şapte zile şi vom avea în vedere extinderea acestor permisiuni.

Păstrând tema aplicației, în continuare, vom *rezuma* principalele puncte atinse în această lucrare.

Astfel, am început discuția cu o scurtă introducere în care am văzut de ce este importantă crearea unei astfel de aplicații şi cum poate afecta într-un mod pozitiv viața oamenilor (pe care i-am numit mai târziu „utilizatori”). Apoi, am prezentat principalele tehnologii care au făcut posibilă dezvoltarea acestei aplicații.

Având stabilite un motiv şi un set de tehnologii care să ne ajute, ne-am îndreptat atenția în capitolul trei către funcționalitatea aplicației noastre din punctul de vedere al utilizatorului. Astfel, realizăm analiza funcțională prin două metode diferite cu scopul de a sabili prioritatea implementării funcționalităților şi calitatea acestora (punctele slabe, punctele tari etc), iar după, parcurgem fiecare element din proiect din punctul de vedere al utilizatorului.

În capitolul patru începem discuția despre cum au fost implementate funcționalitățile discutate în capitolul anterior. Discuția a început prin prezentarea arhitecturii care va fi folosită în dezvoltarea aplicației, urmată de o scurtă introducere a framework-ului Django.

Apoi, ne am rezervat o secțiune specială pentru a discuta despre funcționalitatea principală a aplicației noastre: algoritmul de sumarizare a articolelor şi generarea automată a etichetelor. De asemenea, tot în această secțiune am discutat şi despre cum putem executa întregul proces într-un mod independent de aplicație şi cum îl putem programa să ruleze la un interval de timp prestabilit.

În continuare, după încă o secțiune în care prezentăm diferite alte funcționalități importante ale aplicației, dar cu un impact mult mai mic decât cele discutate anterior (de exemplu căutarea articolelor, votarea acestora, paginare etc) am început discuția despre modelarea bazei de date. Astfel, am realizat diagramele modelului bazei de date şi am ținut o discuție pe baza acestora, iar apoi am văzut cum putem folosi Django pentru a uşura întreaga interacțiune cu baza de date.

Încheiem capitolul patru cu o discuție pe tema securității aplicației noastre, unde vom prezenta principalele modalități prin care Django ne asigură siguranța proiectului.

Nu în ultimul rând, capitolul cinci conține o scurtă listă cu îmbunătățiri şi funcționalități pe care ne dorim să le implementăm în versiuni viitoare ale aplicației.

Bibliografie

[1] Guido van Rossum, Python Dev Team, *Python 3.6 Tutorial*, https://docs.python.org/3/tutorial/, disponibil în data 20.05.2018

[2] William Zeller, Edward W. Felten, *Cross-Site Request Forgeries: Exploitation and Prevention*, https://people.eecs.berkeley.edu/~daw/teaching/cs261-f11/reading/csrf.pdf, secțiunea 3, paginile 4-8, disponibil în data de 20.05.2108.

[3] *Documentația oficială Django*, https://docs.djangoproject.com, disponibil în data de 20.05.2108

[4] *Documentația oficială Celery*, http://docs.celeryproject.org/en/latest/index.html, disponibilă în data 20.05.2018

[5] *Text summarization in Python: Extractive vs. Abstractive techniques revisited*, https://rare-technologies.com/text-summarization-in-python-extractive-vs-abstractive-techniques-revisited/, disponibilă în data 20.05.2018

[6] *Internet*, https://en.wikipedia.org/wiki/Internet, disponibilă în data 20.05.2018