**UNIVERSITATEA DIN BUCURESTI**

**FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ**

**BAZE DE DATE ŞI TEHNOLOGII WEB**

**LUCRARE DE DIZERTAŢIE**

**UTILIZAREA ALGORITMILOR DE SUMARIZARE AUTOMATĂ ÎNTR-O APLICAŢIE DE PREZENTARE A ŞTIRILOR**

**Coordonator ştiinţific: Student:**

**Conf. Dr. Popescu Marius Făşie Ilie Mădălin**

**BUCUREŞTI**

**2018**

Cuprins

# [1. Introducere 4](#__RefHeading___Toc109_4293233207)

# [2. Tehnologii utilizate 6](#__RefHeading___Toc215_4293233207)

[2.1. Python 6](#__RefHeading___Toc313_4293233207)

[2.2. Django 6](#__RefHeading___Toc315_4293233207)

[2.3. HTML şi CSS 8](#__RefHeading___Toc317_4293233207)

[2.4. JavaScript 8](#__RefHeading___Toc319_4293233207)

[2.5. Redis 9](#__RefHeading___Toc1760_1466436559)

[2.6. Celery 9](#__RefHeading___Toc1764_1466436559)

[2.7. Sublime Text 3 10](#__RefHeading___Toc3479_3360707616)

# [3. Descrierea aplicaţiei 12](#__RefHeading___Toc217_4293233207)

[3.1. Audienţa aplicaţiei 12](#__RefHeading___Toc730_1346346633)

[3.2. Analiza functională folosind metodele SWOT şi MoSCoW 12](#__RefHeading___Toc876_1346346633)

[3.3. Scenarii de utilizare 14](#__RefHeading___Toc1301_7137530)

[3.3.1. Pagina de pornire 14](#__RefHeading___Toc2622_1512321347)

[3.3.2. Pagina Blog 16](#__RefHeading___Toc2624_1512321347)

[3.3.3. Paginile utilizatorului 17](#__RefHeading___Toc2626_1512321347)

[3.3.4. Pagina de sumarizare a propriului text şi pagina de contact 17](#__RefHeading___Toc2628_1512321347)

[3.3.5. Diagrama UML 18](#__RefHeading___Toc5460_3658878738)

[4.1. Ce este MVC? 20](#__RefHeading___Toc1303_7137530)

[4.2. Introducere în Django 20](#__RefHeading___Toc1305_7137530)

[4.2.1. Structura unui proiect 20](#__RefHeading___Toc1307_7137530)

[4.2.2. Construirea view-urilor 22](#__RefHeading___Toc1309_7137530)

[4.3. Preluarea automată a ştirilor 23](#__RefHeading___Toc1311_7137530)

[4.3.1. Parsarea ştirilor 23](#__RefHeading___Toc1702_721487483)

[4.3.2. Sumarizarea ştirilor 25](#__RefHeading___Toc1871_1466436559)

[4.3.3. Generarea automată de etichete 28](#__RefHeading___Toc1873_1466436559)

[4.3.4. Programarea procesului 31](#__RefHeading___Toc2494_1412244252)

[4.4. Analiza tehnică a aplicaţiei 33](#__RefHeading___Toc2496_1412244252)

[4.4.1. Sistemul de votare 33](#__RefHeading___Toc2498_1412244252)

[4.4.2. Înregistrarea şi autentificarea utilizatorilor 36](#__RefHeading___Toc2714_1512321347)

[4.4.3. Paginarea şi căutarea 38](#__RefHeading___Toc3469_3360707616)

[4.4.4. Pagini modale 40](#__RefHeading___Toc2718_1512321347)

[4.4.5. Adăugarea imaginilor 41](#__RefHeading___Toc3477_386202915)

[4.5. Modelul bazei de date 42](#__RefHeading___Toc1315_7137530)

[4.5.1. Ce este ORM? 42](#__RefHeading___Toc3451_386202915)

[4.5.2. Vizualizarea bazei de date 44](#__RefHeading___Toc3479_386202915)

[4.5.3. Tabelele şi relaţiile dintre acestea 46](#__RefHeading___Toc3481_386202915)

[4.5.4. Modele şi migrări în Django 49](#__RefHeading___Toc3663_386202915)

[4.5.5. Alegerea sistemului de baze de date 52](#__RefHeading___Toc3893_386202915)

[4.6. Securitatea aplicaţiei 53](#__RefHeading___Toc3419_386202915)

[4.6.1. Securitatea formularelor 53](#__RefHeading___Toc3665_386202915)

[4.6.2. Alte elemente de securitate 55](#__RefHeading___Toc3667_386202915)

# [5. Îmbunătăţiri ulterioare 58](#__RefHeading___Toc221_4293233207)

# [6. Concluzii 61](#__RefHeading___Toc223_4293233207)

# [Bibliografie 63](#__RefHeading___Toc5340_3658878738)

# **1. Introducere**

Istoria internetului începe în anii 1950 odată cu începerea dezvoltării calculatoarelor. În anii următori, începe să se contureze conceptul de „reţea pe arie lungă”, rezultând în formarea proiectului *ARPANET*, precursorul internetului. În 1969 a fost trimis primul mesaj prin această reţea. 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 viaţă servicii precum poşta electronică (e-mail), mesageria instant, realizarea de apeluri telefonice prin internet, apeluri video, forumuri, bloguri și multe altele. Din punct de vedere istoric, aceasta creștere este aproape instantanee.

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

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

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

Mai mult, pe lângă rezumarea articolelor, aplicaţia îşi doreşte crearea unei comunităţi de utilizatori. Într-adevăr, cei mai buni critici sunt oamenii pasionaţi 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 perfecţiona atât algoritmul de sumarizare cât şi alte aspecte şi funcţionalităţi din aplicaţie.

În capitolele care urmează, vom analiza în detaliu această aplicaţie 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 înţeles a limbajului de programare Python combinată cu metodele specifice aplicaţiilor web şi arhitectura MVC incluse în Django, ne oferă un mediu de dezvoltare foarte stabil şi puternic.

Următorul capitol are drept scop prezentarea aplicaţiei din punctul de vedere al unui utilizator normal. După cum vom vedea în acest capitol, aplicaţia 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 aplicaţie sau să scrie propriile articole într-o secţiune 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 înţelegere deplină a deciziilor luate în dezvoltarea aplicaţiei. Odată trecuţi de partea introductivă a acestui capitol, vom aborda şi descrie în detaliu câteva funcţionalităţi care stau la baza aplicaţiei. Exemple de astfel de funcţionalităţi 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 îmbunătăţirilor care nu au putut fi prezente în versiunea aceasta a aplicaţiei, 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 prezenţa semnului „ ; ”, tipul variabilelor este stabilit de interpretator la momentul rulării şi aşa mai departe.

Potrivit creatorului, dezvoltarea limbajului a pornit din dorinţa sa de a avea un „hobby” pe parcursul vacanţei 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 funcţională, programarea procedurală sau programarea imperativă. Aplicaţiile sale sunt, de asemenea, numeroase, el fiind folosit în domenii precum inteligenţa artificială, robotică, automatizare, jocuri, aplicaţii 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 dispoziţie modalităţi de construire a interfeţelor grafice (GUI), de lucru cu baze de date relaţionale, de manipulare a expresiilor regulate, teste unitare, networking şi multe altele.

Important de reţinut 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 faţă, 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 aplicaţii 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 aplicaţiei 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 comunităţii şi utilităţile puse la dispoziţie 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 împărţit în mai multe module de sine stătătoare, denumite „aplicaţii”. Avantajul unei astfel de structuri este acela că se pot copia aplicaţii de la un proiect la altul, fără a fi nevoie de modificări suplimentare. Singurul lucru necesar fiind specificarea denumirii noii aplicaţii în variabila INSTALLED\_APPS din fișierul settings.py al aplicaţiei principale. Pentru a evita orice fel de confuzie, pe parcursul lucrării, prin „aplicaţie” ne vom referi la întregul proiect, urmând să fie specificate explicit cazurile în care ne referim la o aplicaţie Django.

Pentru a facilita dezvoltarea de aplicaţii web dinamice, Django include propriul său limbaj de template. Acest limbaj poate fi inclus în documentele HTML ale aplicaţiei şi permite utilizarea instrucţiunilor 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 consistenţă sporită a aplicaţiei.

Un alt supliment oferit de Django este propriul ORM (Object-Relational Mapping) care uşurează foarte mult lucrul cu baza de date în aplicaţie. Un ORM permite automatizarea transferului de date între tabelele unei baze de date relaţionale folosind cod Python în loc de cod SQL. Două avantaje importante ale utilizării unui ORM sunt posibilitatea de a dezvolta aplicaţia 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 producţie PostgreSQL, nu trebuie decât să specificăm informaţiile 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.

Deși nomenclatura unei aplicaţii 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 relaţională, 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* conţine toate rutele aplicaţiei folosindu-se de expresii regulate şi alcătuind „controller-ul” aplicaţiei.

## **2.3. HTML şi CSS**

Bineînţeles, o aplicaţie 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 întreţinute 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 înţeles, atunci când dezvoltăm aplicaţii mai mari, documentele HTML şi CSS pot deveni destul de complicate. Pentru a permite dezvoltatorilor să se concentreze mai mult pe aspectele tehnice ale aplicaţiei şi nu pe aspect, a fost dezvoltat framework-ul Bootstrap. Aflat acum la versiunea 4, Bootstrap ofera o mulţime de componente HTML şi CSS care pot fi combinate într-o mulţime de moduri diferite pentru a obţine o interfaţă 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, deși 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 popularităţii sale, au început să apară framework-uri care permit dezvoltarea părţii serverului (server-side) folosind tot JavaScript. Exemple de astfel de framework-uri pot fi: Node.js, Angual.js, ReactJS, Meteor.js și 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 operaţiile 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, hărţi (map), seturi şi seturi sortate.

Printre cele mai utilizate aplicaţii ale ale acestei tehnologii se numără: caching (fiind mult mai rapid decât bazele de date tradiţionale sau folosirea soluţiei *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 execuţia job-urilor de fundal aflate într-o coadă.

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

## 2.6. Celery

Atunci când dezvoltăm o aplicaţie 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 funcţionalităţi. Nu putem renunţa nici la acest proces, nici la dorinţa de a avea o aplicaţie 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 soluţie pentru aceste probleme. Celery este o coadă de joburi asincrone bazată pe trimiterea de mesaje distribuite.

După un setup iniţial în care vom specifica unde se găseşte fişierul *settings.py* al aplicaţiei şi aplicaţiile Django care pot folosi acest utilitar, ne îndreptăm atenţia către funcţia care conţine procesul pe care vrem să îl rulăm asincron. Ceea ce ne dorim, de fapt, este să transformăm funcţia noastră într-un task Celery. Acest lucru este posibil adaugând deasupra definiţiei funcţiei decoratorul *@task(name=”task name”)*, unde *name* reprezintă numele taskului.

Am rezolvat problema rulării unei funcţii asincron, dar încă mai trebuie tratat cazul rulării unui task periodic. Celery ne oferă o soluţie 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* primește un obiect *crontab* cu un interval de repetiţie (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\_aplicaţie worker -l info*, unde *nume\_aplicaţie* 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\_aplicaţie beat -l info*, unde *nume\_aplicaţie* 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 aplicaţie 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 și HTML care au o sintaxă suficient de ușor de urmărit şi înţeles. Cu toate acestea, ne vor fi de folos funcţionalităţi precum colorarea textului şi oferirea de sugestii.

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

# **3. Descrierea aplicaţiei**

## **3.1. Audienţa aplicaţiei**

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 părţi intrinseci ale societăţii. Cei care deţin cele mai multe informaţii şi cunoştinţe, sunt capabil să ocupe un loc mai bun pe scara socială cu un efort mult mai redus. Informaţiile ne permit să luăm deciziile optime pentru dezvoltarea noastră şi a comunităţii 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 situaţia financiară a unei alte ţări poate nu ne afectează în mod direct, dar ne poate oferi o idee despre sustenabilitatea unor investiţii în acea zonă. Astfel, putem spune că informaţia în sine nu deţine o putere impresionantă, dar modul în care ea este folosită, poate schimba totul.

Prin urmare, lucrarea de faţă 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 aplicaţiei şi conturarea unui flux logic al acţiunilor, ne permite să dezvoltăm aplicaţia într-un mod în care toate componentele sunt bine legate între ele şi oferă o experienţă intuitivă şi plăcută utilizatorului.

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

Această metodă poate fi folosită nu numai pentru întreaga aplicaţie, ci şi pentru părți ale acesteia. Scopul fiind acela de a decide dacă o anumită funcţionalitate merită să fie introdusă sau nu, care sunt direcţiile ce trebuie urmate pentru implementarea ei, la ce limitări ne putem aştepta şi aşa mai departe. Astfel, analiza SWOT a aplicaţiei 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 esenţa acestora; * Posibilitatea de a rezuma propriul text; * Design responsive * Posibilitatea de adaugare a propriilor articole sau de a citi articolele publicate de alţi utilizatori; * Votarea articolelor considerate interesante, încurajând conţinutul de calitate. | * Articolele de blog pot avea o calitate inferioară; * Algoritmul nu este perfect, astfel unele articole care conţin 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. |
| **Oportunităţi (O)** | **Pericole (T)** |
| * Pot fi introduse reclame prin intermediul unor platforme cunoscute pentru a monetiza aplicaţia; * Lipsa unei concurenţe imporante şi folosind o campanie de publicitate adecvată, ar putea obţine un monopol în piaţă. | * Trebuie respectată politica site-urilor sursă cu privire la preluarea automată a conţinutului acestora. |

O altă metodă de analiză a aplicaţiei 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 părţile implicate în proiect, care sunt specificaţiile 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 funcţionare a aplicaţiei şi că se vor respecta limitările de timp impuse.

Metoda MoSCoW aplicată întregii noastre aplicaţii 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 conţinutului 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 interacţiunile cu aceasta să fie intuitive. Proiectul de faţă a fost dezvoltat cu aceste caracteristici în minte.

Astfel, pagina de pornire va conţine cele mai importante informaţii disponibile, anume lista ştirilor sumarizare, în ordine descrescătoare a datei în care a fost publicat articolul. Spunem „cele mai importante informaţii” deoarece aceasta este tema aplicaţiei, 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 aplicaţiei, dar şi pentru a oferi utilizatorului minumul de informaţii 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 conţine imaginea, titlul, data publicării şi rezumatul articolului, dar şi tagurile (etichetele) generate automat pentru acel articol şi opţiunile 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 toţi utilizatorii autentificaţi, astfel creând o imagine asupra calităţii 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 (diferenţa 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 secţiune ce conţine 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 conţine atât legături către celelalte pagini ale aplicaţiei, cât şi un câmp text prin intermediul căruia putem căuta anumite ştiri în aplicaţie în funcţie de conţinut (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 opţiuni specifice utilizatorului (dacă utilizatorul nu este autentificat, se vor afişa opțiuni de autentificare şi înregistrare, iar dacă este deja autentificat se vor afişa opţiunea 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 aplicaţiei rămâne totuşi în colţul din stânga).

### 3.3.2. Pagina Blog

Blogurile au scopul de a crea o comunitate asociată aplicaţiei 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 comunităţi vine din dorinţa ca această aplicaţie să nu fie o simplă aplicaţie statică de citit articole. Dimpotrivă, atât sistemul de votare, cât şi blogurile sau funcţia de sumarizare a propriului text, constituie o modalitate minimală de a transforma această aplicaţie, 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 colţul dreapta-sus. În acest moment, utilizatorul este rugat să introducă un titlu, o descriere şi un conţinut 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, aplicaţia va adăuga o imagine standard cu textul „None” (inexistent) indicând absenţa 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 funcţionalitate, se va deschide meniul vertical din dreapta bării de navigare şi se va alege opţiunea „My blogs”. În acest moment, utilizatorul va putea vedea toate articolele care îi aparţin, 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 aplicaţie, 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 aplicaţie 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 aplicaţiei noastre. Acest lucu se va observa pe tot parcursul lucrării.

Pentru a interacţiona cu contul său, utilizatorul are la dispoziţie un meniu dropdown (meniu vertical) numit „My account”. Atunci când acesta nu este autentificat în aplicaţie, va avea la dispoziţie opţiunile „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, opţiunea de autentificare. Această opţiune necesită numele de utilizator şi parola alese la pasul anterior. De asemenea, pagina de „Log in” prezintă şi opţiunea de „Forgot my password” (mi-am uitat parola) care va trimite un link pe mail-ul utilizatorului conţinând un link unic de resetare a parolei.

După autentificare, textul meniului dropdown va afişa textul „Welcome” urmat de prenumele utilizatorului. Bineînţeles, şi opţiunile din acest meniu s-au schimbat, ele devenind „My blogs” (blogurile mele) şi „Log out” (de-autentificare). Funcţionalitatea opţiunii 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 aplicaţiei le reprezintă paginile „Summarize your text” (sumarizaţi-vă propriul text) şi „Contact”. Aceste pagini nu afectează conţinutul de informaţii prezent pe aplicaţie, însă oferă utilizatorului un plus de flexibilitate atunci când nu poate găsi informaţiile 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 propoziţii. Pagina conţine două câmpuri, unul numeric corespunzător numărului de propoziţii care vor alcătui rezumatul (în mod implicit sunt alese trei propoziţii) şi unul text ce va conţine textul care va fi sumarizat. Odată introduse aceste informaţii, 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 opţiune poate fi benefică proiectului pentru că putem primi păreri despre ce am putea îmbunătăţi la el, ce funcţionalităţi sunt cele mai utile, ce probleme există, bug-uri etc. Aceste păreri pot transforma această aplicaţie 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

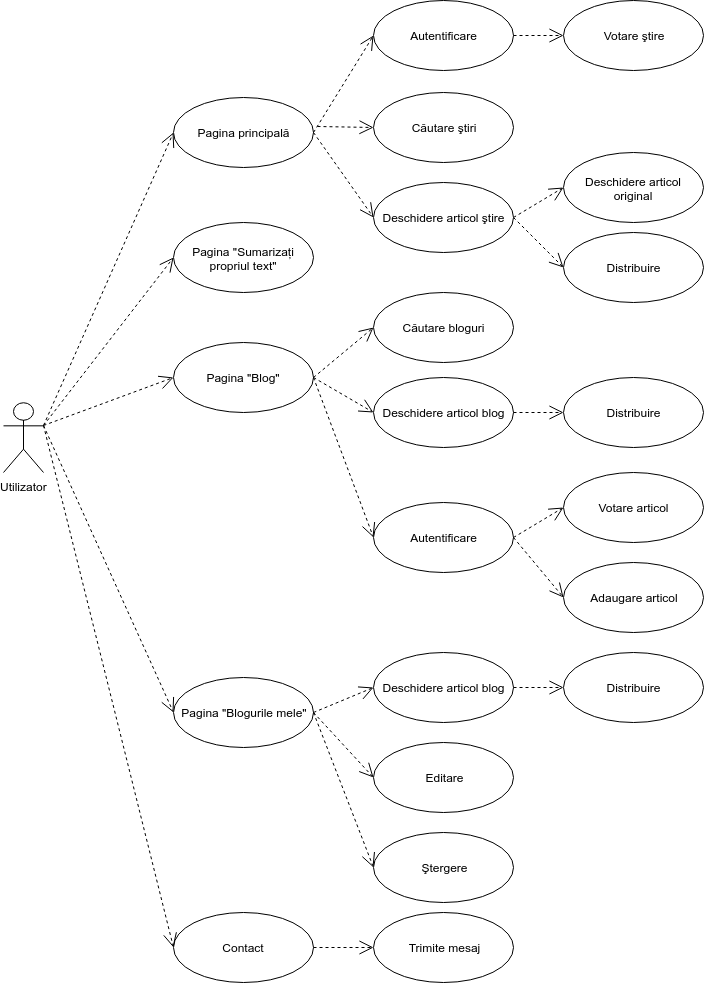
Vom încheia acest capitol prezentând diagrama UML corespunzătoare aplicaţiei noastre (Figura 1).

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

Cu toate acestea, datorită versatilităţii 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 îmbunătăţirilor şi funcţionalităţilor aduse de-a lungul timpului, diagramele UML pot conţine 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 funcţionalităţile aplicaţiei noastre.

  
Figura 1. Diagrama E/R

**4. Arhitectura aplicaţiei**

## 4.1. Ce este MVC?

Modelul arhitectural ales pentru lucrarea de faţă este Model-View-Controller. Acest *design pattern* îşi are originea în dezvoltarea aplicaţiilor desktop, dar a fost introdus şi în dezvoltarea aplicaţiilor web.

După cum spune şi numele, acest pattern împarte aplicaţia în trei componente principale:

* Model – responsabil pentru stocarea datelor şi impunerea unor reguli şi restricţii asupra lor.
* View – poate fi orice reprezentare a datelor din model. De exemplu, o pagina web, o interfaţă 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 aplicaţie simultan, permite gruparea logică a acţiunilor dintr-un controller sau a view-urilor, modificările de cod se pot face cu uşurinţă, acelaşi model poate avea mai multe view-uri.

Dezavantajele unui astfel de model sunt greutatea de a naviga prin cod datorită straturilor de abstractizare suplimentare şi împărţirea aplicaţiei în trei componente va constrânge dezvoltatorul să găsească soluţii 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 conţine un fişier numit *manage.py* care are acelaşi rol precum comanda *django-admin*, dar are grijă şi de câteva setări în locul nostru şi un folder cu acelaşi nume cu al proiectului. În interiorul acestui folder vom găsi alte patru fişiere.

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 fișierul *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*, așa 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* conţine toate funcţiile şi clasele care definesc informaţiile ce vor fi prezentate utilizatorului. Altfel spus, în acest fişier vom avea funcţiile 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 aplicaţie Django are inclusă o interfaţă de administrare numită *admin* şi descrisă în fişierul *admin.py*. Folosind această interfaţă, putem gestiona datele din baza de date, utilizatorii aplicaţiei 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 aplicaţie Django nu conţine un fişier *urls.py*. Acest lucru ne indică faptul că nu suntem obligaţi să avem legături de orice fel cu alte părţi ale proiectului şi putem folosi această aplicaţie doar pentru prezentarea de date sau doar pentru a dezvolta o anumită funcţionalitate cu scopul de a fi integrată în alte aplicaţii, 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 aplicaţiei şi să creăm toate fişierele HTML în acel director.

Totuşi pentru Django există un singur director „templates”, iar tot conţinutul folderelelor „templates” din toate aplicaţiile 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 aplicaţiei. Pentru a evita acest inconvenient, putem crea un alt director cu acelaşi nume cu cel al aplicaţiei Django în care ne aflăm având astfel pentru fiecare aplicaţie structura: *templates/numeAplicaţie/fişier.html*.

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

Acest limbaj de template este evidenţiat în document prin semnele „{% %}” pentru instrucţiuni şi „{{ }}” pentru variabile şi se bazează pe blocuri de instrucţiuni. Blocurile pot fi standard (de exemplu blocul unei instrucţiuni *if*) sau putem defini propriile noastre blocuri ({% block numeBloc %} {% endblock %}). Exemple de instrucţiuni 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 lipsește 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 și *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 „inspectaţi 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*. Diferenţa dintre cele două este că *find* returnează primul element care îndeplineşte condiţia oferită ca parametru, iar *find\_all* returnează o listă cu toate elementele care îndeplinesc această condiţie.

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 funcţie de care să căutăm, încercăm să găsim un element părinte care să cuprindă toate aceste *p*-uri, dar care are cât mai puţine elemente care nu fac parte din articol. Pentru publicaţiile 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 publicaţiei pentru care realizăm extragerea. Acest lucru este posibil folosind expresii regulate, identificând textul aflat între *„www.”* şi *„.co”*. Apoi într-o instrucţiune *if*, folosind *find\_all* din librăria BeautifulSoup, identificăm elementele *div* ce conţin paragrafele articolului. Mai rămâne doar să construim o listă ce conţine 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 bucăţi 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 publicaţie, texte care anunţă prezenţa unei reclame şi multe altele.

Pentru a elimina aceste porţiuni de text, vom construi o listă în care vom identifica toate paragrafele „inutile” aplicaţiei noastre folosind aceleaşi metode descrise mai sus, iar înainte să construim textul articolului, vom adăuga o instrucţiune *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 informaţiilor. 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 aplicaţii 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 informaţiile utile. O metoda foarte bună pentru a combate acest lucru o reprezintă indexarea, dar aceasta implică pierderea formei iniţiale a textului, lucru care pentru maşini nu este important, dar pentru noi, oamenii, face diferenţa.

De aceea, problema extragerii doar a informaţiilor 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 propoziţiilor din textul iniţial, și construirea rezumatului fără a le modifica. Sumarizarea abstractivă îşi propune alcătuirea unui rezumat ce conţine fraze şi cuvinte noi care să cuprindă sensul textului iniţial. 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 faţă, poate produce rezultate mai slabe decât contracandidata sa, rămâne metoda folosită de oameni pentru a rezuma un text (astfel, potenţialul său este foarte mare). În această lucrarea, ne vom concentra numai pe metoda extractivă.

În continuare vom prezenta câţiva algoritmi de sumarizare.

Algoritmul Luhn, publicat în anul 1958, presupune alegerea propoziţiilor importante în funcţie de cuvintele cu frecvenţa cea mai mare și distanţa 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 propoziţii, 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 propoziţiile alese nu sunt prea asemănătoare între ele.

Algoritmul LSA (Latent Semantic Analysis), spre deosebire de ceilalți doi algoritmi prezenzaţi, abordează problema „înţelesului” unui cuvânt. Într-adevar, un cuvânt poate avea mai multe înţelesuri în funcţie de contextul în care se află sau cuvinte diferite pot avea acelaşi sens (sinonime). Aceste lucruri pot pune în dificultate chiar și o persoană. Soluţia propusă de algoritm presupune scufundarea spaţiului cuvintelor noastre într-un spaţiu de dimensiune mai mică (numit şi spaţiu latent) în care compararea a două cuvinte poate fi realizată cu mai mare uşurinţă.

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

* Documentele sunt reprezentate folosind modelul „bag of words” („sac de cuvinte”), care presupune asocierea fiecărui cuvânt cu numărul de apariţii în document. Totuşi, utilizând acest model, vom pierde informaţiile 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 ceilalţi algoritmi. Pentru a folosi acest algoritm, vom utiliza modulul Python „sumy”. Acest modul ofera implementări pentru toţi 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 funcţiei, vom parcurge lista de adrese URL conţinută în parametrul *news\_sites*, iar pentru fiecare element, apelăm funcţia *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 puţin 1500 de caractere, întrucât un text de dimensiuni mai mici ar fi suficient de scurt încât această operaţie 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 despărţi în cuvinte, respectiv propoziţii 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 dispoziţie de *sumy* ce primeşte la rândul său în constructor, limba în care este scris articolul (aceeași cu cea dată drept parametru funcţiei 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 conţine la rândul său, limba articolului. Linia de cod care realizează cele descrise arata astfel: summarizer=Summarizer(Stemmer(language)). Iniţializăm câmpul *stop\_words* al stemmerului nostru folosind funcţia *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 funcţiei noastre *make\_summary*. Fiecare element al acestei tuple reprezintă o propoziţie a rezumatului nostru. Astfel, construim un nou string prin concatenarea acestor propoziţii şi obţinem textul sumarizat.

Deşi poate părea ciudat că este apelat un obiect asemeni unei funcţii 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 conţină toate informaţiile 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 informaţiilor. Un prim gând în această direcţie ar fi posibilitatea de a accesa anumite articole în funcţie de o temă, sau un termen. Cu alte cuvinte, ne dorim să avem la dispoziţie un mecanism de căutare.

O soluţie naivă la această problemă ar fi să căutăm fiecare cuvânt din textul nostru de căutare în textul, titlul şi descrierea articolelor sumarizate. Deşi în aparenţă 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 soluţie nu ne oferă nici un fel de organizare a informaţiilor din aplicaţia noastră.

Soluţia 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 puţine 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 puţin 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 funcţie 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 și trigrame din fiecare articol nesumarizat (alegem varianta nesumarizată pentru că ne oferă mult mai multe informaţii despre conţinutul 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 frecvenţă suficient de mare în text, înseamnă că acea asociere deţine informaţii importante despre conţinutul acestuia.

Implementarea funcţiei noastre începe prin a iniţializa variabila ce conţine lista cu stopwords (cuvinte care au o frecvenţă ridicată în majoritatea textelor şi, astfel, nu deţin informaţii importante despre conţinutul 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 *colocaţie* drept o expresie formată din mai multe cuvinte care apar împreună în mod frecvent. Aplicând această definiţie pentru problema noastră, vom încerca să găsim colocaţii de bigrame şi trigrame.

În continuare vom descrie algoritmul doar pentru bigrame, cazul trigramelor fiind analog. Pentru a găsi toate colocaţiile din text, folosim următoarea linie de cod finder\_b=BigramCollocationFinder.from\_words(word\_tokenize(text)). Obiectul obţinut va conţine toate colocaţiile pentru bigramele din textul nostru. Pentru a păstra doar informaţiile care contează, filtrăm datele astfel încât să fie luate în considerare doar colocaţiile cu au o frecvenţă 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 colocaţii în funcţie de importanţa 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 discrepanţa dintre probabilitatea ca două evenimente să se întâmple simultan şi produsul probabilităţilor individuale (presupunând că cele două evenimente sunt independente).

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

Dacă afişăm această listă, observăm că fiecare element este format dintr-o tuplă ce conţine 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 conţin 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 frecvenţa 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 condiţii), tot ce a rămas de făcut este să folosim această funcţie pentru articolele noastre şi să le salvăm în baza de date.

Astfel, revenind în funcţia *make\_summary* discutată în secţiunea precedentă, imediat după ce am salvat în baza de date ştirea cu rezumatul asociat, vom apela funcţia *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 importanţă crescută în textul analizat, însă acesastă structură poate genera şi bigramele Mohamed bin şi bin Salam care sunt redundante).

Parcurgem lista obţinută 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 restricţiilor impuse pentru taguri, pot exista articole de ştire fără nici un tag asociat. Acest lucru este într-adevăr nefericit, iar soluţionarea sa va fi introdusă într-o versiune viitoare a aplicaţiei.

### 4.3.4. Programarea procesului

În secţiunile 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 aplicaţia simultan, vom observa că fiecare instanţă va rula acelaşi proces, rezultând în articole duplicate.

Bineînţeles, 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 viaţă al aplicaţiei. Prin urmare, avem nevoie de o metodă care să ruleze acest proces automat şi independent de starea aplicaţiei.

Soluţia 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 aplicaţia noastră. Primul pas ar fi să creăm un fişier *celery.py* în interiorul aplicaţiei Django principale (cea cu aceelaşi nume ca al proiectului), iar în interiorul acestuia adăugăm un fragment de cod prestabilit[[1]](#footnote-2), pe care îl adaptăm astfel încât să conţină informaţiile 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 funcţionalitate. 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'

CELERY\_TIMEZONE = 'UTC'

în fişierul *settings.py*. Aceste informaţii 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 conţin parserul, summarizerul şi generatorul de taguri, şi creăm o funcţie în interiorul acestuia pe care o vom numi *get\_news*. Această funcţie 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 propoziţii care vor alcătui rezumatul articolelor. De asemenea, singurul lucru făcut de această funcţie este să apeleze funcţia *make\_summary* descrisă într-o secţiune anterioară a lucrării cu parametrii de mai sus.

Pentru a transforma această funcţie 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 funcţie care primeşte ca parametru funcţia decorată cu scopul de a-i extinde comportamentul fără a o modifica în mod direct. Cu alte cuvinte, este un wrapper peste funcţia noastră ce ne permite să efectuăm anumite operaţii 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 producţie, aceste procese trebuie întreţinute de un tool (instrument) de monitorizare a proceselor precum *Supervizor*. Astfel, vom avea mult mai mult control asupra proceselor de fundal din aplicaţia noastră şi vom avea o acoperire mai bună în cazuri excepționale precum căderi ale reţelei 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 aplicaţiei

### 4.4.1. Sistemul de votare

Dezvoltarea unei aplicaţii 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 aplicaţia 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 dispoziţie pagina de bloguri, iar pentru a-şi exprima opinia despre conţinutul prezentat în aplicaţie, 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 compoziţia acestora. Apoi, accesarea fiecărei dintre căile astfel create va apela o funcţie de votare (pozitivă sau negativă, după caz) care va adăuga o nouă înregistrare într-o tabelă din baza de date care va conţine 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 aplicaţii Django, într-o listă denumită *urlpatterns*. Structura unei căi este *url(expresie\_regulată, funcţia\_asociată, numele\_rutei)*, unde *expresie\_regulată* reprezintă un model care defineşte calea relativă (faţă de pagina părinte) unde se va găsi pagina în cauză, *funcţia\_asociată* este funcţia sau metoda care se va apela atunci când este accesată adresa descrisă de expresia regulată anterioară, iar *numele\_rutei* este un parametru opţional 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 funcţie 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 funcţiei asociate alături de request (care este trimis în mod implicit), iar numele parametrului va fi acelaşi cu numele specificat între parantezele ascuţite „*<>*”.

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

Funcţia *upvote* va primi drept parametrii requestulşi 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 funcţii. Această informaţie 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.userşi 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 absenţei 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 conţine 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 funcţia de *downvote*.

Ambele funcţii vor putea fi apelate doar de utilizatori autentificaţi în aplicaţie. Pentru a specifica acest lucru, este suficient să adăugăm decoratorul *@login\_required* din modulul *django.contrib.auth.decorators* deasupra definiţiei funcţiilor noastre. Rezultatul fiind că utilizatorul va fi redirecţionat 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 operaţii sunt posibile datorită limbajului de template oferit de Django. Astfel, este posibil să verificăm dacă utilizatorul este autentificat sau nu printr-o instrucţiune precum {% if request.user.is\_anonymous %} cod\_html\_utilizator\_neautentificat {% else %} cod\_html\_utilizator\_autentificat {% endif %} şi putem efectua operaţia de adunare prin {{ n.vote\_up| add:n.vote\_down}}, unde *n* este articolul din tabela *News*. Vom folosi prima instrucţiune 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 nevoiţi să scădem numărul de voturi negative din numărul de voturi pozitive. Din nefericire, limbajul de template oferit de Django nu ne pune la dispoziţie o modalitate de scădere a două variabile. Totuşi, putem trece peste acest inconvenient, definindu-ne propria funcţie care să îndeplinească această cerinţă.

Aşadar, într-un nou folder numit *templatetags* din interiorul aplicaţiei Django în care ne aflăm, creăm un fişier *custom\_filters.py*, iar în interiorul acestuia, adăugăm o funcţie numită *sub* care va primi doi parametri şi va returna diferenţa dintre aceştia. Pentru a indica totuşi că această funcţie trebuie tratată drept un filtru pentru limbajul de template, va trebui să importăm clasa *Library* din modulul *django.template* şi să instanţiem 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 funcţia care va efectua operaţia 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 secţiune va fi adaptat şi pentru aplicaţia 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 aplicaţia noastră. Mai mult, unii utilizatori vor avea mai multe drepturi decât ceilalţi. Prin urmare, avem nevoie de o modalitate prin care să putem diferenţia aceste tipuri.

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

Sistemul de autentificare Django suportă toate cerinţele 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 acţiunea de a verifica validitatea credenţialelor 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 obţine un obiect de tip *User* care conţine aceste informaţii. După cum probabil aţi observat, numele și prenumele nu au fost incluse. Pentru a realiza acest lucru, este suficient să pasăm aceste informaţii, 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 anunţa succesul creării contului său. Pentru a trimite acest mail, este necesară setarea informaţiilor 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 conţinutul acestor câmpuri ţin de soluţia aleasă, iar descrierea lor nu este importantă pentru lucrarea de faţă. Ceea ce ne interesează totuşi, este modul în care este folosită această funcţionalitate.

Având toate acestea în vedere, trimiterea email-ului de confirmare este posibilă folosind funcţia *send\_mail* din modulul *django.core.mail*. Această funcţie primeşte drept parametrii, subiectul email-ului, conţinutul, adresa emiţătorului şi o listă cu persoanele care vor primi acest mail (în cazul nostru lista va conţine 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 relaţie *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 funcţia 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 informaţii în baza de date folosind metoda *save()* şi trimitem email către adresa specificată de viitorul utilizator în care vom include un link unic către aplicaţia noastră care va conţine cheia de activare.

Atunci când este accesată această adresă unică, se va apela o altă funcţie din fişierul *views.py* care primeşte cheia de activare ca parametru şi care, după verificarea validităţii 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 funcţionalitatea 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 conţine câmpurile necesare pentru introducerea numelui de utilizator şi a parolei (bineînţeles, cu tot cu aspectul general al paginii), iar în fişierul *urls.py* al aplicaţiei Django în care am lucrat până acum în această secţiune, 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 funcţia care va fi apelată şi care va realiza autentificarea utilizatorului, al treilea parametru conţine un dicţionar în care specificăm fişierul HTML creat anterior şi formularul de login corespunzător, iar ultimul parametru este numele asociat rutei, astfel încât să fie uşor referenţiat din interiorul funcţiilor.

În linia de cod de mai sus, toţi parametrii ne sunt cunoscuţi cu excepţia celui de-al doilea. Funcţia *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 aplicaţia 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 funcţionalităţii 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, conţine doar un mesaj de confirmare şi o legătură către pagina principală.

### 4.4.3. Paginarea şi căutarea

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

Diferenţa 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 distanţă maximă cunoscută atât de partea de sus a aplicaţiei, cât şi de cea de jos, astfel asigurându-ne că acesta are acces la toate meniurile şi funcţionalităţile existente cu un efort cât mai mic. De asemenea, având tot conţinutul pe o singură pagină, putem crea senzaţia de copleşire cu informaţii 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 instanţiem 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 conţine un câmp numit *page* care va conţine numărul paginii *i*.

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

Asemeni paginării, căutarea va reduce senzaţia de copleşire cu informaţii 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 aplicaţie.

Pentru implementarea acestei funcţionalităţi, 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 aplicaţiei.

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 funcţie 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. Bineînţeles, conceptul este trivial şi nu necesită multă dezbatere, însă pentru a oferi un design mai plăcut aplicaţiei, dar şi pentru a îmbunătăţi viteza de afişare a articolelor selectate, am ales folosirea paginilor modale.

O pagină modală reprezintă o pagină care în loc să se încarce peste conţinutul pe care îl avem deja pe pagină, se încarcă deasupra acestuia, într-o porţiune 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 obţinem 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 conţine în calea sa id-ul articolului, apoi vom crea o funcţie în fişierul *views.py* pe care o vom apela atunci când va fi accesată această rută.

Funcţia 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 publicaţiei 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 dispoziţie de Bootstrap. Denumirile acestor clase sunt suficient de sugestive, astfel că vom presupune evident modul în care le-am folosit pentru a afişa informaţiile extrase de funcţia 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 aplicaţie.

### 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 conţine doar text, nu atrage atenţia 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 aplicaţie), 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 înălţimii, respectiv lăţimii imaginii.
* Coloana *height\_field* este de tipul *models.IntegerField()* şi indică înălţimea imaginii.
* Coloana *width\_field* este de tipul *models.IntegerField()* şi indică lăţimea 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, iniţializăm câmpul *widgets* cu un dicţionar care conţine 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) și 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 posiblităţi: ori convertim câmpul *adresă* în grupuri de informaţii scalare înainte de a le adăuga în baza de date (şi bineînţeles, trebuie să avem o posibilitate de a recrea obiectul iniţial plecând de la baza de date), ori rescriem tot codul astfel încât să corespundă restricţiilor 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 instrucţiuni SQL.

Avantajele unui astfel de sistem sunt:

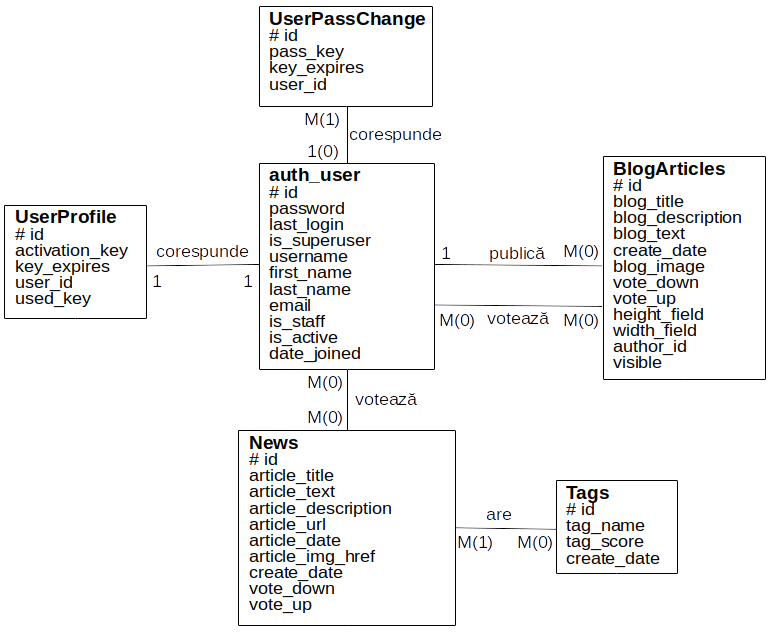
* Timpul de dezvoltare a aplicaţiilor scade deoarece ORM-ul rezolvă multe lucruri în mod automat (conexiunea la baza de date, construirea instrucţiunii SQL, executarea acesteia şi, dacă este cazul, returnarea rezultatului, pot fi restrânse la apelul unei singure metode). De asemenea, acest sistem ne 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 întreţinut).
* Un ORM oferă un plus de flexibilitate în dezvoltarea aplicaţiei 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 greșelile 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 cunoştinţe de baze de date), deoarece este foarte uşor pentru ORM să construiască interogări foarte grele din punct de vedere computaţional folosind doar câteva linii de cod (de exemplu, o interogare complexă inclusă într-un ciclu repetitiv) sau se pot defini relaţii între tabele care nu sunt normalizare corespunzător.
* Cum un ORM este doar o librărie, va fi necesară învăţarea acesteia, iar înlocuirea cu un alt ORM va implica reluarea procesului de învăţare.
* Deşi din punct de vedere al performanţei, 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 aplicaţia noastră, vom alege sistemul Django ORM.

### 4.5.2. Vizualizarea bazei de date

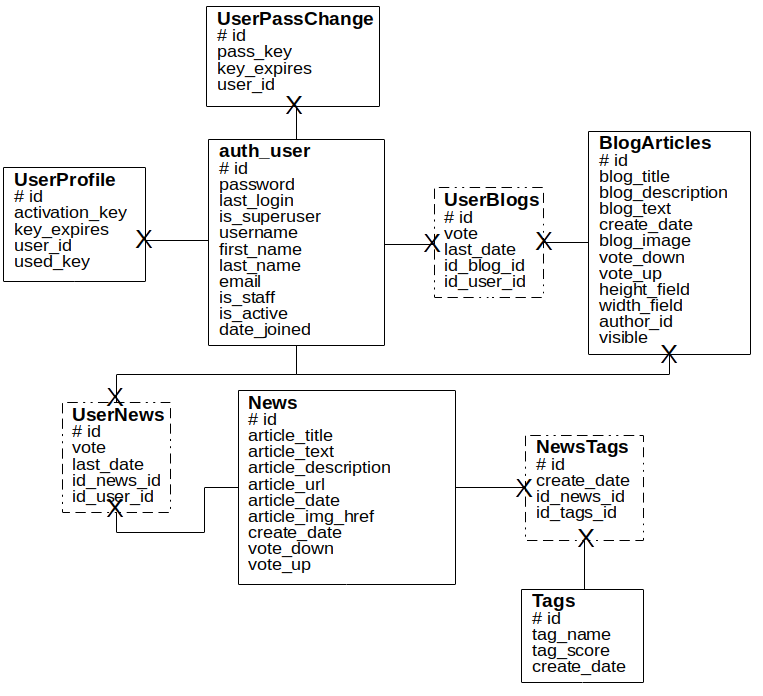
  
Figura 2. Diagrama E/R

Pentru a ne fi mai uşor să înţelegem 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 discuţiei despre modelul bazei de date, vom trata numai tabelele care au un impact în aplicaţia noastră, întrucât folosind framework-ul de autentificare pus la dispoziţie 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-relaţie (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-Relaţie: către o vizualizare unificată a datelor).

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

  
Figura 3. Diagrama conceptuală

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

În cazul aplicaţiei noastre de sumarizare, diagrama entitate-relaţie 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.

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

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

### 4.5.3. Tabelele şi relaţiile dintre acestea

Tabela AUTH\_USER, deşi nu reprezintă funcţionalitatea principală a aplicaţiei, 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 relaţie cu aceasta.

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

Această tabelă conţine atât informaţii specifice fiecărui utilizator cât şi informaţii 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 aplicaţii;
* *is\_staff*: tot un câmp boolean care indică dacă utilizatorul are dreptul să se autentifice în interfaţa de administrare a aplicaţiei, dar nu are drepturi depline în aceasta (în proiectul nostru nu am inclus o astfel de interfaţă întrucât nu exista un concept de ierarhie care să o impună);
* *is\_active*: indică dacă utilizatorul se poate autentifica în aplicaţie (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 aplicaţie;
* *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 excepţia coloanei *last\_login*.

Tabelele USERPROFILE şi USERPASSCHANGE sunt foarte similare din punct de vedere al conţinutului, dar diferite din punct de vedere al utilităţii şi relaţiei 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* conţin 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-relaţie, USERPROFILE se află în relaţie de *unu-la-unu* cu tabela AUTH\_USER, iar USERPASSCHANGE şi AUTH\_USER sunt în relaţie de *unu-la-mai mulţi*.

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 relaţia 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* conţin informaţiile pentru titlul, conţinutul, 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 relaţie *mai mulţi-la-mai mulţi* cu tabelele AUTH\_USER şi TAGS. Această relaţie ne va impune crearea a câte unui tabel de legătură pentru fiecare apariţie a sa.

Astfel, relaţia dintre tabela utilizatorilor şi cea a ştirilor, va fi împărţită în două relaţii de *unu-la-mai mulţi* între AUTH\_USER şi USERNEWS, respectiv între NEWS şi USERNEWS. Tabela de legătură USERNEWS, va conţine tipul votului fiecărui utilizator care a votat cel puţin 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, relaţia dintre tabela NEWS şi tabela TAGS se va transforma în două relaţii *unu-la-mai mulţi* între cele două tabele şi NEWSTAGS. Această tabelă de legătură conţine 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 NULLpentru 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 secţiune 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 relaţie cu tabela NEWS, iar detaliile acestei relaţii au fost prezentate mai sus.

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

Coloanele acestei tabele sunt: coloanele care păstrează informaţiile despre titlul, conţinutul, 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*), înălţimea şi lăţimea 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 aplicaţie (*visible*).

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

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

Cea de-a doua relaţie, este de tipul *unu-la-mai mulţi* 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 soluţia SQL aleasă, deşi suficient de simplă la început, poate deveni greu de întreţinut pe termen lung. Mai mult, dacă dezvoltatorul nu are suficientă experienţă cu această tehnologie, soluţiile sale pot avea un impact negativ asupra modelului, putându-se ajunge la un punct în care revenirea la starea iniţială să fie foarte costisitoare.

Pentru a evita o astfel de problemă, Django ne oferă o soluţie 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 aplicaţia.

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 aplicaţiei Django în care are cel mai mult sens să fie creată. Într-adevăr, putem importa această clasă şi din alte aplicaţii Django din interiorul proiectului, dar pentru a păstra un curs logic de-a lungul proiectului, este indicat să fie inclusă în aplicaţia în care modelul este accesat pentru prima dată.

Coloanele vor fi reprezentate prin atributele clasei corespunzătoare modelului. Django ne pune la dispoziţie 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 opţiuni (adevărat sau fals), *DateTimeField* indică un câmp care conţine data şi timpul (similar, *DateField* va descrie o coloană ce conţine doar data), şi multe altele.

Pentru a crea constrângerile, Django oferă un mix de câmpuri şi opţiuni 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 reţinut 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 opţiunea de ştergere în cascadă). Mai mult, precizarea acestui câmp va crea în mod automat o relaţie de tipul *unu-la-mai mulţi* cu tabela refereţiată.

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

O altă observaţie 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\_aplicaţie\_nume\_clasa\_model*, unde *nume\_aplicaţie* reprezintă numele aplicaţiei 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 ţintă, 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 observaţie, 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 aplicaţiei, 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 cerinţele şi cum să transforme cerinţa 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 aplicaţia 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 conţine toate aplicaţiile 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 înţelegere deplină a întregului concept de migrări, Django ne îndeamnă să ne gândim la acesta ca la un sistem de control al versiunilor[[2]](#footnote-3) precum *Git*. Astfel, comanda *makemigrations* va crea fişiere speciale care vor conţine 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 (atenţie 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 garanţia că baza lor de date locală va arăta la fel.

### 4.5.5. Alegerea sistemului de baze de date

Mai sus am stabilit ce entităţi vor fi adăugate în baza de date, ce relaţii 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 aplicaţia noastră, nu ne interesează atât de mult calităţile generale ale acestora, ci, mai degrabă, compatibilitatea lor cu framework-ul Django. În acest fel, ne asigurăm că avem la dispoziţie cât mai multe metode de gestionare automată oferite de Django.

Având în minte toate acestea, vom alege pentru aplicaţia noastră sistemul de gestionare PostgreSQL. PostgreSQL (sau simplu Postgres) este un sistem de gestionare al bazelor de date obiect-relaţie (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 funcţionalităţilor 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 funcţionalităţi puse la dispoziţie 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 aplicaţii care folosesc hărţi, 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 aplicaţia noastră se poate realiza foarte simplu din fişierul *settings.py* al aplicaţiei principale, unde trebuie precizat următorul dicţionar:

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 acţiunile, *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 atenţi totuşi, pentru a realiza conexiunea la baza de date, aceasta din urmă trebuie să existe deja. Django ne asigură crearea şi întreţinerea entităţilor din baza de date, dar nu şi baza de date în sine.

## 4.6. Securitatea aplicaţiei

### 4.6.1. Securitatea formularelor

Să presupunem că am dezvoltat o aplicaţie web pentru o bancă care are drept scop gestionarea conturilor bancare ale fiecărui client. În mod natural, aplicaţia noasră va avea un număr de formulare pentru introducerea informaţiilor necesare creării unui nou cont, transferului banilor de la un cont la altul şi multe alte astfel de operaţiuni. 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 aplicaţia este gata să fie lansată în producţie.

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

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

Mai mult, exemplul prezentat este bazat pe un caz real, iar banca în cauză este ING Direct[[3]](#footnote-4). 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 maliţios 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 acţiunea este executată chiar de utilizator.

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

* De-autentificarea utilizatorului de pe aplicaţia ţintă;
* Modificarea preferinţelor victimei pe pagina respectivă;
* Publicarea de conţinut 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 aplicaţiei noastre de astfel de atacuri. Pentru a face acest lucru, avem două posibilităţi: verificăm header-ul pentru a vedea de unde a fost trimisă cererea (în header vom avea informaţii 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 aplicaţiei noastre), atunci blocăm cererea. A doua posibilitate este să includem un câmp ascuns în formularul nostru care să conţină o cheie secretă pe care o vom verifica mai apoi înainte de a înainta formularul.

Avantajul primei metode constă în uşurinţa 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 dorinţa 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 locaţie (dacă atacatorul are acces la acea locaţie, poate vedea cheia de protecţie şi o poate folosi pentru un atac CSRF).

Django foloseşte cea de-a doua metodă pentru a asigura protecţia formularelor împotriva unor astfel de atacuri. Implementarea acesteia implică următoarele acţiuni: 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 conţine valoarea secretă generată anterior, orice request HTTP care nu foloseşte una din metodele GET, HEAD, TRACE sau OPTIONS şi care nu conţine 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 protecţie).

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

### 4.6.2. Alte elemente de securitate

Din moment ce aplicaţia noastră suportă conectarea mai multor utilizatori şi oferă funcţionalităţi diferite în funcţie de starea acestora (autentificaţi sau nu), se ridică problema limitării accesului utilizatorilor la anumite pagini pentru care nu sunt autorizaţi. 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 modalităţi simple de a controla accesul în aplicaţie. 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 funcţiei view (din fişierul *views.py*) corespunzătoare afişării paginii ţintă. Folosind acest decorator, Django va redirecţiona orice utilizator neautentificat care încearcă să acceseze acea pagină, către pagina de login.

Pentru exemplul 2, nu avem nevoie sa restricţionăm accesul la întreaga pagină, ci doar la anumite funcţionalităţi. Astfel, ne vom orienta atenţia către template-ul care descrie aspectul paginii (şi conţine elementele pe care ne dorim să le restricţionăm). Folosind sintaxa {% if request.user.is\_anonymous %} elemente\_dezactivate {%else%} elemente\_activate {%endif%}, unde *elemente\_dezactivate* reprezintă codul HTML ce trebuie afişat pentru utilizatorii neautentificaţi, iar *elemente\_activate* codul HTML pentru utilizatorii autentificaţi.

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

Având toate acestea, paginile restricţionate din aplicaţia noastră sunt: pagina de adăugare, de editare şi de ştergere a blogurilor, pagina de vizualizare a blogurilor postate de un utilizator, funcţiile sistemului de votare (chiar dacă dezactivăm butonul, adresa URL care apelează funcţia rămâne disponibilă). Funcţionalităţile restricţionate din template sunt: butoanele sistemului de votare (au fost dezactivate pentru utilizatori neautentificaţi) şi meniul vertical din dreapta bării de navigare (pentru utilizatori neautentificaţi va afişa opţiuni de autentificare şi înregistrare, iar pentru ceilalţi, va afişa legături către pagina „blogurile mele” şi log out.

Un alt aspect important în legătură cu securitatea aplicaţiei, 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 funcţie hash SHA256. Această metodă ar trebui să fie suficientă pentru a asigura o securitate de nivel suficient de înalt pentru majoritatea aplicaţiilor. 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, toţi algorimii fiind păstraţi într-o listă, este posibil ca primul element al listei să fie utilizat pentru stocarea parolelor noi, iar ceilalţi algoritmi pentru verificarea parolelor salvate deja cu alţi 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 modalităţi de verificare a validităţii 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 puţin 8 caractere, verificarea dacă toate caracterele sunt alfanumerice şi verificarea dacă parola nu este una foarte comună[[4]](#footnote-5). Mai mult, framework-ul ne permite crearea propriilor clase de validare sau modificarea proprietăţilor celor existente.

Toate clasele de validare sunt incluse într-o listă numită *AUTH\_PASSWORD\_VALIDATORS* din fişierul *settings.py* sub forma unor dicţionare cu un singur element care are cheia *„NAME”* și valoarea un string care conţine calea către clasa de validare. Toate clasele implicite sunt definite în modulul *django.contrib.auth.password\_validation*.

# **5. Îmbunătăţiri ulterioare**

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

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

Prin îmbunătăţiri, vom înţelege modificări aduse proceselor şi componentelor deja existente în aplicaţie cu scopul de a spori experienţa utilizatorului şi de a asigura o mai bună stabilitate a aplicaţiei.

Câteva îmbunătăţiri 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 garanţia creării a cel puţin unui tag pentru fiecare articol de ştire);
* O altă îmbunătăţire 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 adiţională ar putea fi obligarea utilizatorului de a folosi cel puţin 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 posibilităţii ca utilizatorul să anuleze un vot;
* Preluarea articolelor de ştiri de la mai multe publicaţii şi preluarea ştirilor din mai multe categorii.

Din punct de vedere al funcţionalităţilor adiţionale pe care ne dorim să le includem într-o versiune viitoare a aplicaţiei, 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 bineînţeles, posibilitatea de a modifica votul). Mai mult, tot din această interfaţă să putem oferi posibilitatea de a-şi schimba parola, numele de familie sau prenumele;
* Dacă îndeplinim primul punct din secţiunea de îmbunătăţiri (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 importanţei acestora. Câţiva factori relevanţi 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 funcţie de cele mai noi articole, de numărul de voturi, de un punctajul calculat prin metoda descrisă la punctul anterior. Mai mult, sortarea în funcţie de acest punctaj ar putea deveni sortarea implicită a articolelor;
* O altă funcţionalitate care poate nu are un impact atât de mare precum cele de mai sus, dar care poate îmbunătăţi considerabil calitatea conţinutului din aplicaţie o reprezintă adăugarea unei metode de raportare a conţinutului 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 iniţial al acestei pagini (de exemplu, este un articol despre viața personală a autorului, fără relevanţă 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 informaţiile cuprinse de acesta sunt atât de numeroase încât este aproape imposibil să ţinem pasul cu tot ce se întâmplă în jurul nostru.

Într-adevăr, pentru mulţi, 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 perfecţionează abilitatea de a utiliza această ustensilă vor avea toate mijloacele necesare spre a cunoaşte lumea din jur.

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

Mai mult, aşa cum am văzut de-a lungul lucrării, impactul utilizatorului asupra calităţii informaţiei 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 aplicaţiei, în continuare, vom *rezuma* principalele puncte atinse în această lucrare.

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

Având stabilite un motiv şi un set de tehnologii care să ne ajute, ne-am îndreptat atenţia în capitolul trei către funcţionalitatea aplicaţiei noastre din punctul de vedere al utilizatorului. Astfel, realizăm analiza funcţională prin două metode diferite cu scopul de a sabili prioritatea implementării funcţionalităţilor 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 discuţia despre cum au fost implementate funcţionalităţile discutate în capitolul anterior. Discuţia a început prin prezentarea arhitecturii care va fi folosită în dezvoltarea aplicaţiei, urmată de o scurtă introducere a framework-ului Django.

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

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

Încheiem capitolul patru cu o discuţie pe tema securităţii aplicaţiei noastre, unde vom prezenta principalele modalităţi prin care Django ne asigură siguranţa proiectului.

Nu în ultimul rând, capitolul cinci conţine o scurtă listă cu îmbunătăţiri şi funcţionalităţi pe care ne dorim să le implementăm în versiuni viitoare ale aplicaţiei.

# 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, secţiunea 3, paginile 4-8, disponibil în data de 20.05.2108.

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

[4] *Documentaţia 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

1. *Codul în sine poate fi găsit în documentaţia oficială: http://celery.readthedocs.io/en/latest/getting-started/next-steps.html#using-celery-in-your-application* [↑](#footnote-ref-2)
2. *Vezi pagina din documentaţia oficială: https://docs.djangoproject.com/en/2.0/topics/migrations/* [↑](#footnote-ref-3)
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. [↑](#footnote-ref-4)
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/* [↑](#footnote-ref-5)