### Proiectarea bazei de date

Baza de date este o necesitate pentru ca aplicația să funcționeze corespunzător. Conține informațiile corespunzătoare care trebuie să fie afișate în frontend. Aplicația dispune de o bază de date care conține 15 tabele: auth\_permission, auth\_group\_permissions, auth\_group, auth\_user\_user\_permissions, auth\_user, auth\_user\_groups, django\_admin\_log, django\_content\_type, django\_session, django\_migrations, catalog\_subscription, catalog\_seminar, contacts\_contact, trainings\_training, instructors\_instructor. Dar din aceste 15 tabele doar 6 sunt de o importanță deosebită pentru proiect și anume: auth\_user, contacts\_contact, catalog\_subscription, catalog\_seminar, trainings\_training și instructors\_instructor. Toate tabelele au fost create prin Django ORM folosind PostgreSQL.

“ORM vine de la Object-Relational Mapper și este o bibliotecă de coduri care automatizează transferul datelor stocate în tabelele de baze de date relaționale în obiecte care sunt utilizate frecvent în codul aplicației.” [34]

“ORM-urile oferă o abstractizare de nivel înalt pe o bază de date relațională care permite unui dezvoltator să scrie cod Python în loc de SQL pentru a crea, citi, actualiza și șterge date și scheme din baza de date. Dezvoltatorii pot folosi limbajul de programare cu care sunt confortabili pentru a lucra cu o bază de date în loc să scrie instrucțiuni SQL sau proceduri stocate.” [34]

“Framework-ul Django vine cu propriul modul ORM încorporat, denumit în general Django ORM sau Django’s ORM. Django ORM funcționează bine pentru operațiuni de baze de date cu complexitate simplă și medie.” [34]

Pentru configurarea bazei de date se folosește fișierul **settings.py**:

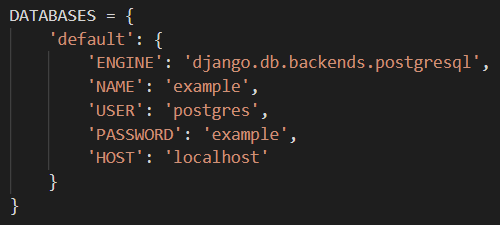


Figura 3.9: Configurarea bazei de date

În ordine se ia:

* **ENGINE** – se precizează ce sistem de baze de date se dorește a fi folosit. În cazul de față se utilizează PostgreSQL.
* **NAME** – se pune numele bazei de date
* **USER** – numele utilizatorului care se dorește pentru baza de date
* **PASSWORD** – se introduce parola bazei de date
* **HOST** – se inserează host-ul care poate fi mașina locală sau orice altă adresă de rețea

În ceea ce privește constrângerile asupra bazei de date există următoarele:

* **PRIMARY KEY** – cheia primară conferă unicitate tabelei și nu poate avea valoarea NULL. Pentru Django aceasta se generează automat atunci când este creată tabela în baza de date.
* **FOREIGN KEY** – cheia străină, care nu este altceva decât o legătură între 2 tabele
* **UNIQUE** – atestă faptul că în același tabel nu vor exista valori duplicat. La Django aceasta se realizează prin setarea câmpului **unique** la “True”
* **NOT NULL** – nici o coloană din tabel nu va putea avea valoarea NULL. Este realizată implicit. Pentru dezactivare se setează câmpul **blank** pe “True”.

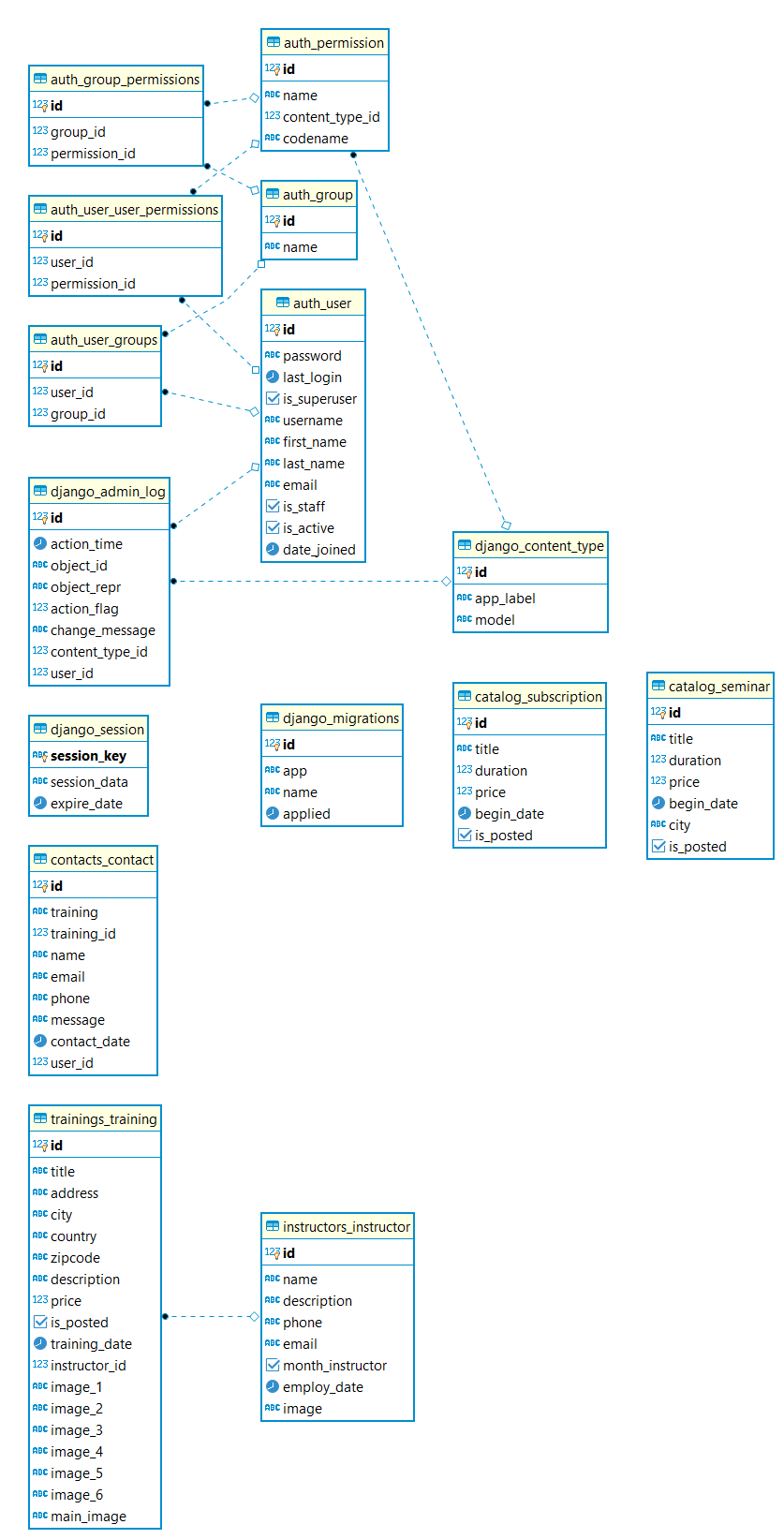


Figura 3.10: Diagrama bazei de date

Tabelele au fost create pe baza execuției comenzilor **python manage.py makemigrations** și **python manage.py migrate**. Tabelele: contacts\_contact, catalog\_subscription, catalog\_seminar, trainings\_training și instructors\_instructor au fost create cu aceleași comenzi dar pornind de la niște modele scrise în Python.

“Un model este sursa unică definitvă de informații despre date. Conține câmpurile și comportamentele esențiale ale datelor solicitate. În general, fiecare model mapează o singură tabelă de bază de date.” [36]

“Regulile elementare: ” [36]

* “Fiecare model este o clasă Python care subclasează **django.db.models.Model**.” [36]
* “Fiecare atribut al modelului reprezintă un câmp al bazei de date.” [36]
* “Django oferă API-ul de acces la baza de date generată automat.” [36]

“După ce au fost definite modelele, Django trebuie să știe că acestea vor fi folosite. Acest lucru se face prin editarea fișierului de setări **settings.p**y și modificarea în **INSTALLED\_APPS** în așa fel încât să fie adăugat numele modulului care conține **models.py**.” [36]

“Cea mai importantă parte a unui model – și singura parte necesară a unui model – este lista câmpurilor bazei de date pe care le definește. Câmpurile sunt specificate de atributele clasei. Trebuie avut grijă să nu se aleagă nume de câmp care intră în conflict cu API-ul modelelor, cum ar fi curățarea, salvarea sau ștergerea.” [36]

“Comanda **python manage.py makemigrations** îi precizează lui Django că au fost făcute unele modifcări la modele și că modificările să fie stocate ca migrare. ” [37]

“Migrațiile reprezintă modul în care Django stochează modficările la modele (și astfel schema bazei de date) – sunt fișiere pe disc. ” [37]

“Comanda **python manage.py migrate** preia toate migrațiile care nu au fost aplicate (Django urmărește care sunt aplicate folosind un tabel din baza de date numit **django\_migrations**) și le execută asupra bazei de date, sincronizând modificările care au fost făcute asupra modelelor cu schema bazei de date. ” [37]

“Migrațiile sunt foarte puternice și permit schimbarea modelelor în timp, fără a fi nevoie să se șteargă baza de date sau tabelele și să se facă unele noi – se specializează în modernizarea bazei de date în direct, fără a pierde date. Este important de reținut trei pași pentru a face schimbări de model: ” [36]

* “Se schimbă modelele (în **models.py**). ” [37]
* “Se execută **python manage.py makemigrations** pentru a crea migrații pentru acele modificări. ” [37]
* “Se execută **python manage.py migrate** pentru a aplica aceste modificări la baza de date. ” [37]

“Motivul pentru care există comenzi separate pentru efectuarea și aplicarea migrațiilor este acela că se efectuează migrații către sistemul de control al versiunii și sunt livrate cu aplicația; nu numai că facilitează dezvoltarea, ci sunt utilizate și de alți dezvoltatori și în producție. ” [37]

**Tabela auth\_user**: aici sunt stocați toți utilizatorii indiferent de rolul pe care-l au. Atât cei care sunt angajați, cât și cei care nu sunt angajați vor fi regăsiți în tabelă. Fiecare utilizator are un ID propriu pe baza căruia va putea fi identificat mai ușor.

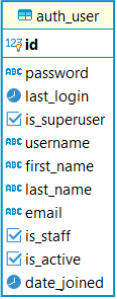


Figura 3.11: Tabela auth\_user

Această tabelă nu are un model anume, ci este creată cu numele și câmpurile de mai sus automat când se rulează comenzile **python manage.py makemigrations** și **python manage.py migrate** prima dată.

**Tabela contacts\_contact**: stochează toate cererile de informații suplimentare sau feedback pe care un utilizator le efectuează asupra unui antrenament. Utilizatorul este identificat pe baza ID-ului său în baza de date.

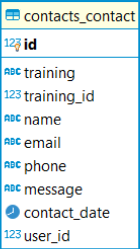


Figura 3.12: Tabela contacts\_contact

Modelul pe baza căruia a fost obținută tabela contacts\_contact este următorul:

# Modelul pentru tabela contacts\_contact

class Contact(models.Model):

    training = models.CharField(max\_length=200)

    training\_id = models.IntegerField()

    name = models.CharField(max\_length=200)

    email = models.CharField(max\_length=100)

    phone = models.CharField(max\_length=100)

    message = models.TextField(blank=True)

    contact\_date = models.DateTimeField(default=datetime.now, blank=True)

    user\_id = models.IntegerField(blank=True) # se ia ID-ul utilizatorului care este autentificat pentru a fi afișate antrenamentele asupra cărora s-au efectuat cereri de informații suplimentare sau feedback

**Tabela instructors\_instructor**: conține instructorii care vor susține antrenamentele. Există legătură între această tabelă și tabela trainings\_training cu scopul de a specifica în mod clar fiecare instructor ce antrenamente are de susținut.

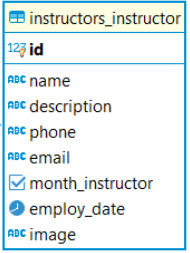
****

Figura 3.13: Tabela instructors\_instructor

Modelul pe baza căruia a fost obținută tabela instructors\_instructor este următorul:

# Modelul pentru tabela instructors\_instructor

class Instructor(models.Model):

    name = models.CharField(max\_length=200, unique=True)

    image = models.ImageField(upload\_to='images/%Y/%m/%d/')

    description = models.TextField(blank=True, unique=True)

    phone = models.CharField(max\_length=20, unique=True)

    email = models.CharField(max\_length=50, unique=True)

    month\_instructor = models.BooleanField(max\_length=False)

    employ\_date = models.DateTimeField(default=datetime.now, blank=True)

**Tabela trainings\_training**: conține toate antrenamentele care vor fi desfășurate. Există legătură între această tabelă și tabela instructors\_instructor cu scopul de a se cunoaște fiecare antrenament cărui instructor îi este atribuit.

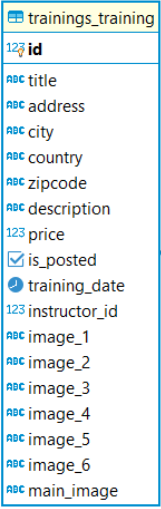


Figura 3.14: Tabela trainings\_training

Modelul pe baza căruia a fost obținută tabela trainings\_training este următorul:

# Modelul pentru tabela trainings\_training

class Training(models.Model):

    instructor = models.ForeignKey(Instructor, on\_delete=models.DO\_NOTHING)

    title = models.CharField(max\_length=200, unique=True)

    address = models.CharField(max\_length=200, unique=True)

    city = models.CharField(max\_length=100, unique=True)

    country = models.CharField(max\_length=100, unique=True)

    zipcode = models.CharField(max\_length=20, unique=True)

    description = models.TextField(blank=True, unique=True)

    price = models.IntegerField()

    main\_image = models.ImageField(upload\_to='images/%Y/%m/%d/')

    image\_1 = models.ImageField(upload\_to='images/%Y/%m/%d/', blank=True)

    image\_2 = models.ImageField(upload\_to='images/%Y/%m/%d/', blank=True)

    image\_3 = models.ImageField(upload\_to='images/%Y/%m/%d/', blank=True)

    image\_4 = models.ImageField(upload\_to='images/%Y/%m/%d/', blank=True)

    image\_5 = models.ImageField(upload\_to='images/%Y/%m/%d/', blank=True)

    image\_6 = models.ImageField(upload\_to='images/%Y/%m/%d/', blank=True)

    is\_posted = models.BooleanField(default=True)

    training\_date = models.DateTimeField(default=datetime.now,blank=True)

**Tabela catalog\_subscription**: conține toate abonamentele disponibile, iar scopul este acela de a-i oferi utilizatorului o gamă cât mai largă din care să aleagă.

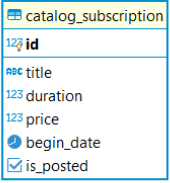


Figura 3.15: Tabela catalog\_subscription

Modelul pe baza căruia a fost obținută tabela catalog\_subscription este următorul:

# Modelul de bază pentru tabelele catalog\_subscription și catalog\_seminar

class Catalog(models.Model):

    title = models.CharField(max\_length=200, unique=True)

    duration = models.IntegerField()

    price = models.IntegerField()

    is\_posted = models.BooleanField(default=True)

    # Se face modelul Catalog model de bază

    class Meta:

        abstract = True

# Modelul pentru tabela catalog\_subscription folosind moștenirea

class Subscription(Catalog):

    begin\_date = models.DateTimeField(default=datetime.now, blank=True)

**Tabela catalog\_seminar**: conține toate seminariile disponibile, iar scopul este acela de a-i oferi utilizatorului o diversitate cât mai mare din care să selecteze.

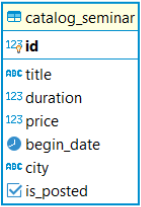


Figura 3.16: Tabela catalog\_seminar

Modelul pe baza căruia a fost obținută tabela catalog\_seminar este următorul:

# Modelul de bază pentru tabelele catalog\_subscription și catalog\_seminar

class Catalog(models.Model):

    title = models.CharField(max\_length=200, unique=True)

    duration = models.IntegerField()

    price = models.IntegerField()

    is\_posted = models.BooleanField(default=True)

    # Se face modelul Catalog model de bază

    class Meta:

        abstract = True

# Modelul pentru tabela catalog\_seminar folosind moștenirea

class Seminar(Catalog):

    begin\_date = models.DateTimeField(default=datetime.now, blank=True)

    city = models.CharField(max\_length=100, unique=True)

După cum se observă tabelele catalog\_subscription și catalog\_seminar au fost obținute folosind moștenirea de model.

“Moștenirea modelului în Django funcționează aproape identic cu modul în care funcționează moștenirea normală a clasei în Python. Aceasta înseamnă următorul lucru: clasa de bază ar trebui să subclasese **django.db.models.Model**.” [36]

“Singura decizie care trebuie luată este dacă se dorește ca modelele părinte să fie modele la propriu (cu propriile tabele de bază de date) sau dacă părinții sunt doar deținători de informații comune care vor fi vizibile doar prin modele copii.” [36]

În cazul de față s-a optat pentru a doua variantă și anume modelele părinți au doar informații comune care pot fi văzute prin modele copii. Varianta aceasta poartă numele: Clase de Bază Abstracte.

“Clasele de bază abstracte sunt utile atunci când se dorește introducerea câtorva informații comune într-o serie de alte modele. Se scrie clasa de bază și se pune **abtrasct=True** în clasa **Meta**. Acest model nu va fi apoi utilizat pentru crearea niciunui tabel de bază de date. În schimb, atunci când este utilizat cu scopul de clasă de bază pentru alte modele, câmpurile sale vor fi adăugate la cele din clasa copil.” [36]

Ca să putem folosi modelele s-a folosit următorul import : from django.db import models.

Pentru a minimiza cât mai mult posibilitatea obținerii de date eronate în baza de date s-a folosit principiul normalizării bazelor de date.[38] Mai departe sunt enunțate formele normale în care se încadrează baza de date.

Prima formă normală (FN1) precizează că nu trebuie să existe grupuri repetitive astfel încât fiecare câmp din baza de date să aibă doar o valoare atomică, adică să nu mai poată fi descompusă.[38] Trebuie și ca fiecare înregistrare să poată fi identificată printr-o cheie primară.[39] Baza de date a proiectului se încadrează în această formă pentru că fiecare tabelă are unicitatea oferită prin prezența unei chei primare și fiecare câmp capătă doar o valoare atomică.

A doua formă normală (FN2) precizează că o bază de date trebuie să fie în FN1 [38] și că toate elementele unei tabele să fie dependente funcțional de totalitatea cheii primare. Dacă unul sau mai multe elemente sunt dependente funcțional parțial de cheia primară, atunci este nevoie de separarea în tabele diferite.[39] Din moment ce baza de date a proiectului are toate tabelele cu cheia primară formată doar dintr-un singur atribut se poate spune că aceasta este automat în FN2.

A treia formă normală (FN3) precizează că o bază de date trebuie să fie în FN2 și să nu existe dependențe funcționale între atributele non-chei[38], adică orice atribut care nu face parte din cheie trebuie să depindă direct de cheia primară.[40] Acest lucru implică ca în baza de date să existe informații din mai multe domenii, ceea ce în cazul bazei de date a proiectului nu se aplică.

Forma normală Boyce-Codd (FNBC) este o variantă mai restrictivă a FN3, și anume atributele depind în întregime de o cheie. [39]

### Proiectarea aplicației

Pentru proiectarea aplicației și pentru a face mai ușor procesul de înțelegere, s-au realizat mai multe diagrame UML care explică conceptual logica ce stă la baza fiecărei interfețe. Se începe cu diagrama paginii *Home*:

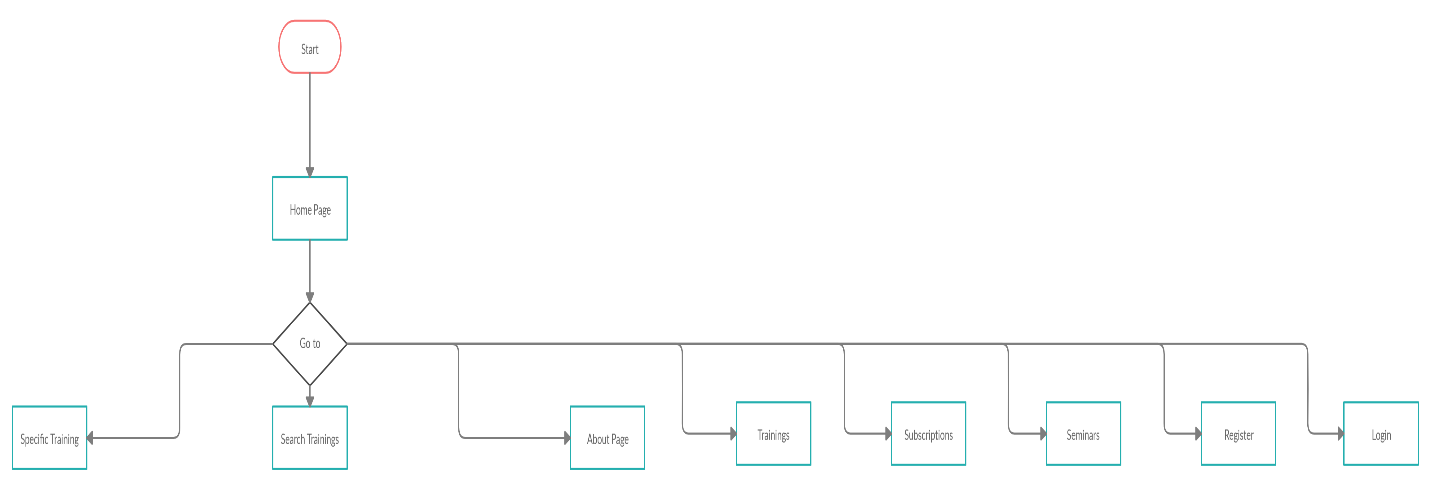


Figura 3.17: Diagrama UML a paginii Home

Diagrama de mai sus prezintă faptul că odată ce utilizatorul se află în pagina principală, acesta are la dispoziție mai multe opțiuni din care să aleagă și anume: poate să meargă direct la un antrenament specific, poate să caute un antrenament în particular, poate să viziteze paginile *About*, *Trainings*, *Subscriptions* sau *Seminars*, precum și să se înregistreze sau să se autentifice.

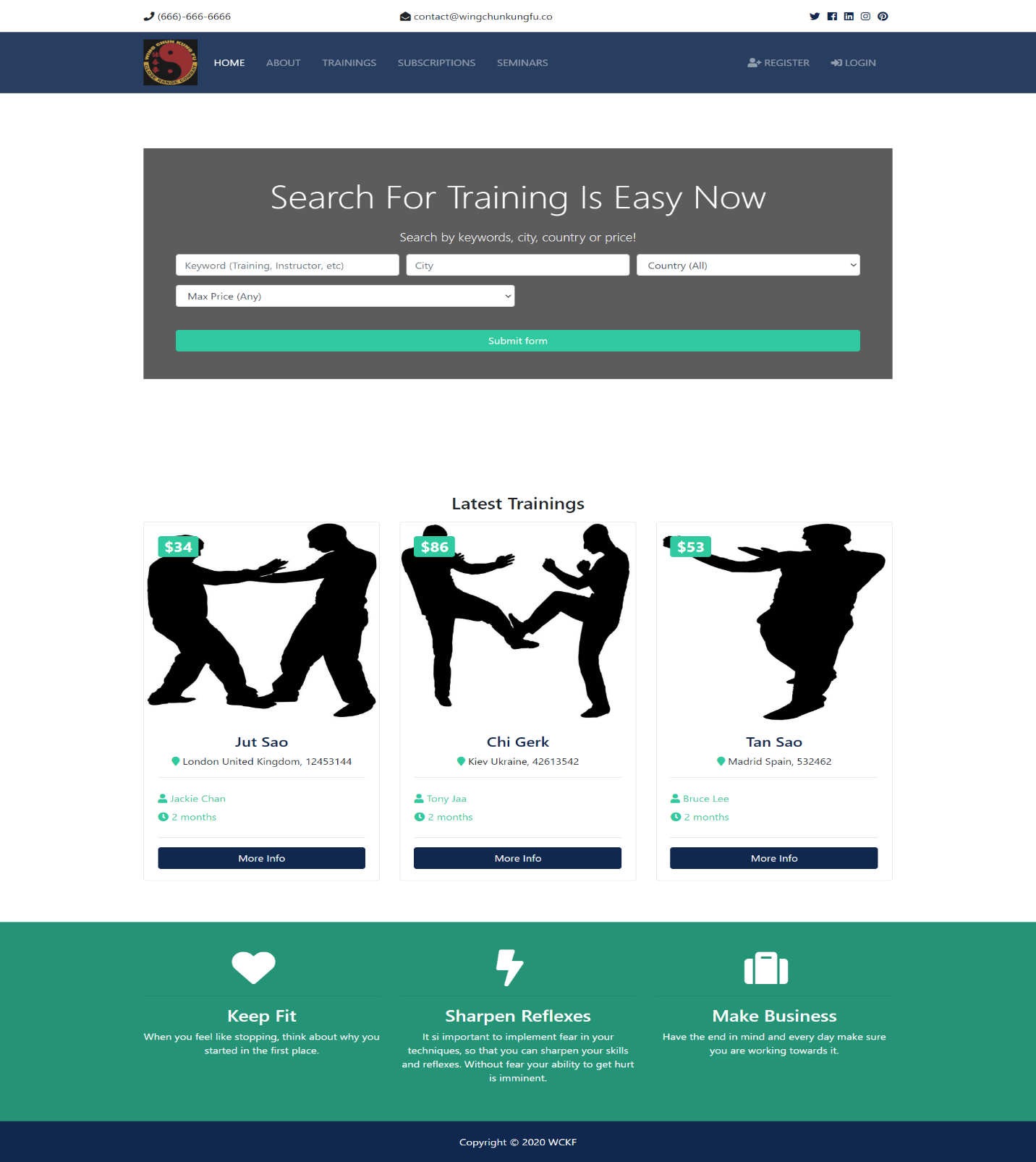


Figura 3.18: Interfața paginii Home

După cum se observă utilizatorul poate comuta între pagini alegând între *About*, *Trainings*, *Subscriptions*, *Seminars*, precum și să se înregistreze prin apăsarea butonului *Register*, să se autentifice apăsând butonul *Login*, să caute un antrenament introducând informațiile dorite și apăsând butonul Search sau să accese direct unul apăsând butonul *More info*.

Pentru pagina *Home*:

# Metoda index pentru pagina Home

def index(request):

# Se iau toate antrenamentele din baza de date ordonate descrescator (-) după câmpul training\_date

# și filtrate după câmpul  is\_posted și sunt afișate ultimele 3 cele mai recente

    trainings = Training.objects.order\_by('-training\_date').filter(is\_posted=True)[:3]

# Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'trainings' : trainings,

        'country\_choices' : country\_choices,

        'price\_choices' : price\_choices

    }

    # Se returnează pagina Home

    return render(request,'pages/index.html', context)

Metoda *index* ia ca parametru o cerere (*request*) care “este un obiect de tip HttpRequest și conține informații despre cerere.” [41]

Aceeași metoda se regăsește și “în fișierul **urls.py**, cu mențiunea că nu se apelează funcția **view.index**, ci se enumeră doar o referință la aceasta. Django apelează apoi funcția atunci când intră o solicitare de potrivire și trece obiectul HttpRequest ca paramatru.” [41]

Se creează variabila *trainings* care este un QuerySet și este setat pe numele modelului și anume *Training*. Folosind metoda *objects* se iau toate informațiile din baza de date și vor ordonate descrescător (-) pe baza câmpului *training\_date* și afișate doar dacă au câmpul *is\_posted* bifat în zona administratorului. Vor fi afișate ultimele 3 cele mai recente.

Dicționarul *context* care se bazează pe perechi cheie-valoare funcționează în felul următor: cheile vor fi folosite în pagina HTML a interfeței pentru a putea fi afișate în frontend, iar valorile sunt de fapt înregistrările din baza de date care au fost prelucrate mai devreme.

Se returnează cu funcția *render* în felul următor: “funcția render ia obiectul cerere (*request*) ca prim argument, un template ca al doilea argument și un dicționar ca al treilea argument opțional. Întoarce un obiect HttpResponse al template-ului cu contextul dat. ” [42]

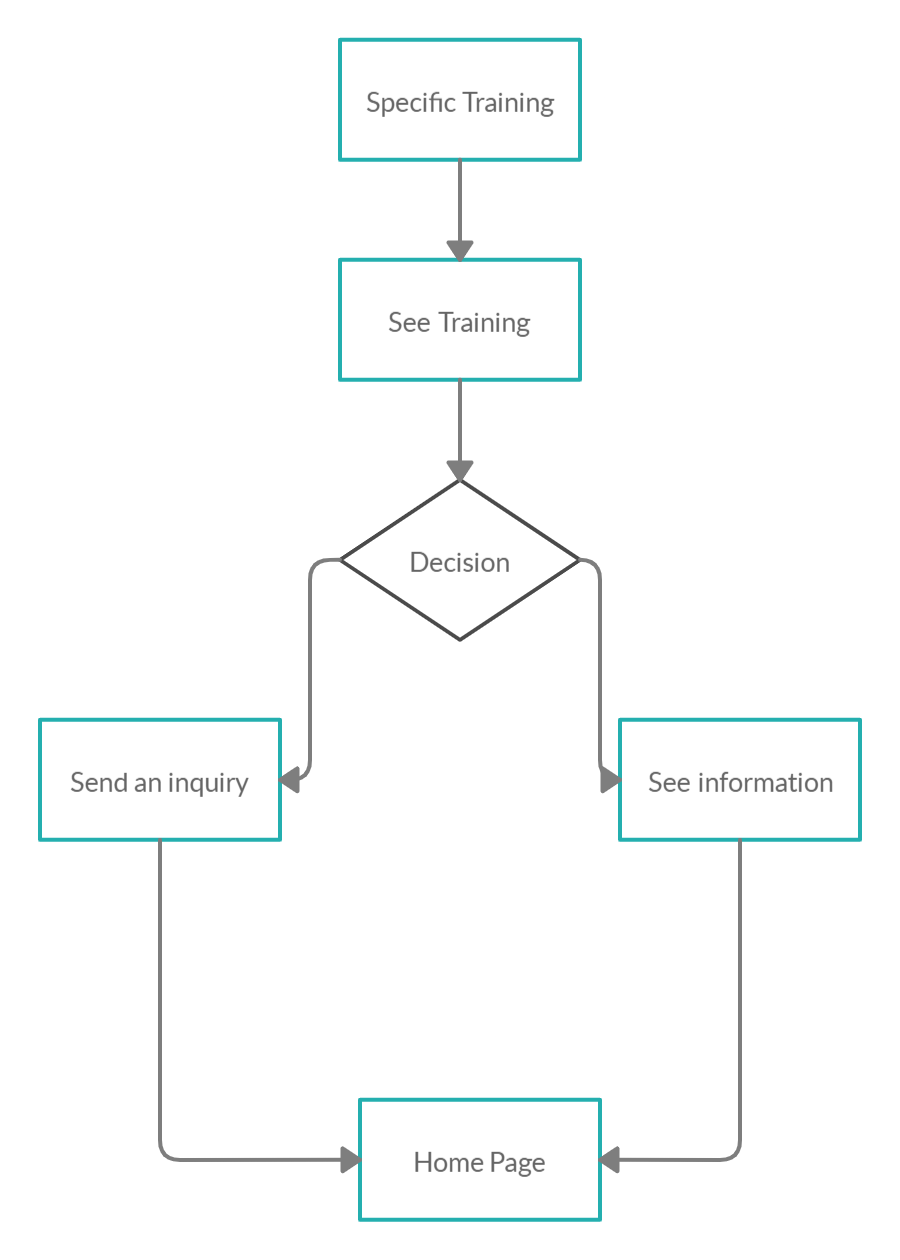


Figura 3.19: Diagrama UML a paginii unui antrenament specific

Diagrama arată faptul că utilizatorul care a accesat pagina unui antrenament specific poate să trimită o cerere de informații suplimentare despre antrenament sau să ofere feedback instructorului care a susținut acel antrenament, precum și afișarea informațiilor adecvate despre antrenament, iar la final se întoarce la pagina *Home*.

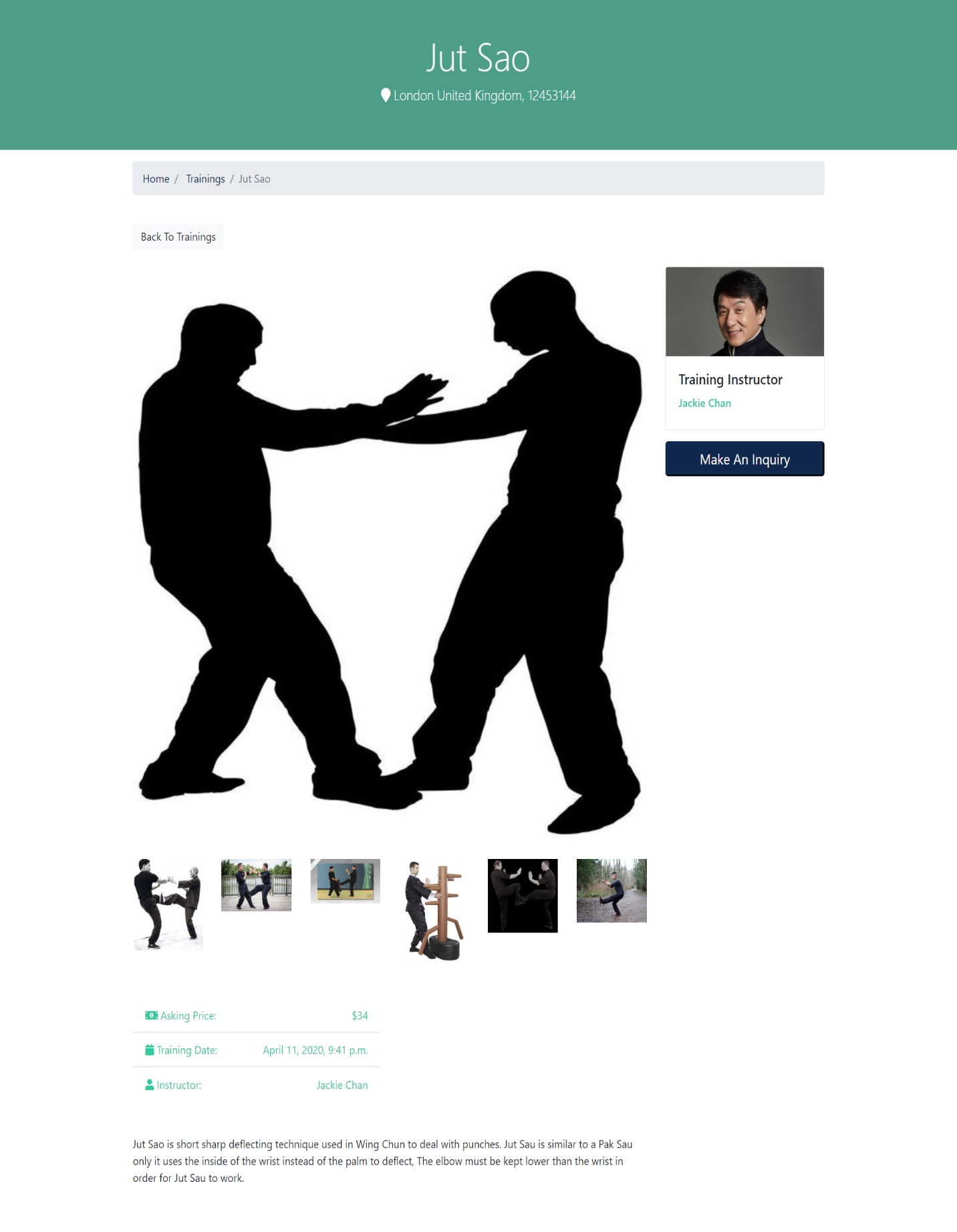


Figura 3.20: Interfața paginii unui antrenament specific

După cum se observă în imagine utilizatorul poate face o cerere de informații suplimentare sau oferi feedback prin apăsarea butonului *Make an inquiry*, vizualiza informații, să se întoarcă la pagina *Home* prin apăsarea butonului cu același sau să meargă înapoi la antrenamente prin apăsarea butonului *Back To Trainings*.

Pentru pagina unui antrenament specific:

# Metoda pentru un antrenament specific

def training(request, training\_id):

# Se verifică faptul că antrenamentul există pe baza prezenței id-ului său în baza de date există, altfel se afișează o eroare Http404

    training = get\_object\_or\_404(Training, pk=training\_id)

    # Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'training': training

    }

    # Se returnează pagina antrenamentului specific

    return render(request,'trainings/training.html', context)

“Funcția *get\_object\_or\_404 ()* ia un model Django ca prim argument și un număr arbitrar de argumente de cuvinte cheie, pe care le trece funcției *get ()* a managerului modelului. Afișează Http404 dacă obiectul nu există. ” [42]

“Un Manager este interfața prin care sunt furnizate operațiuni de interogare a bazelor de date modelelor Django. Există cel puțin un Manager pentru fiecare model dintr-o aplicație Django.”[43]

“În mod implicit, Django adaugă un Manager cu numele *objects* la fiecare clasă de model Django.” [43]

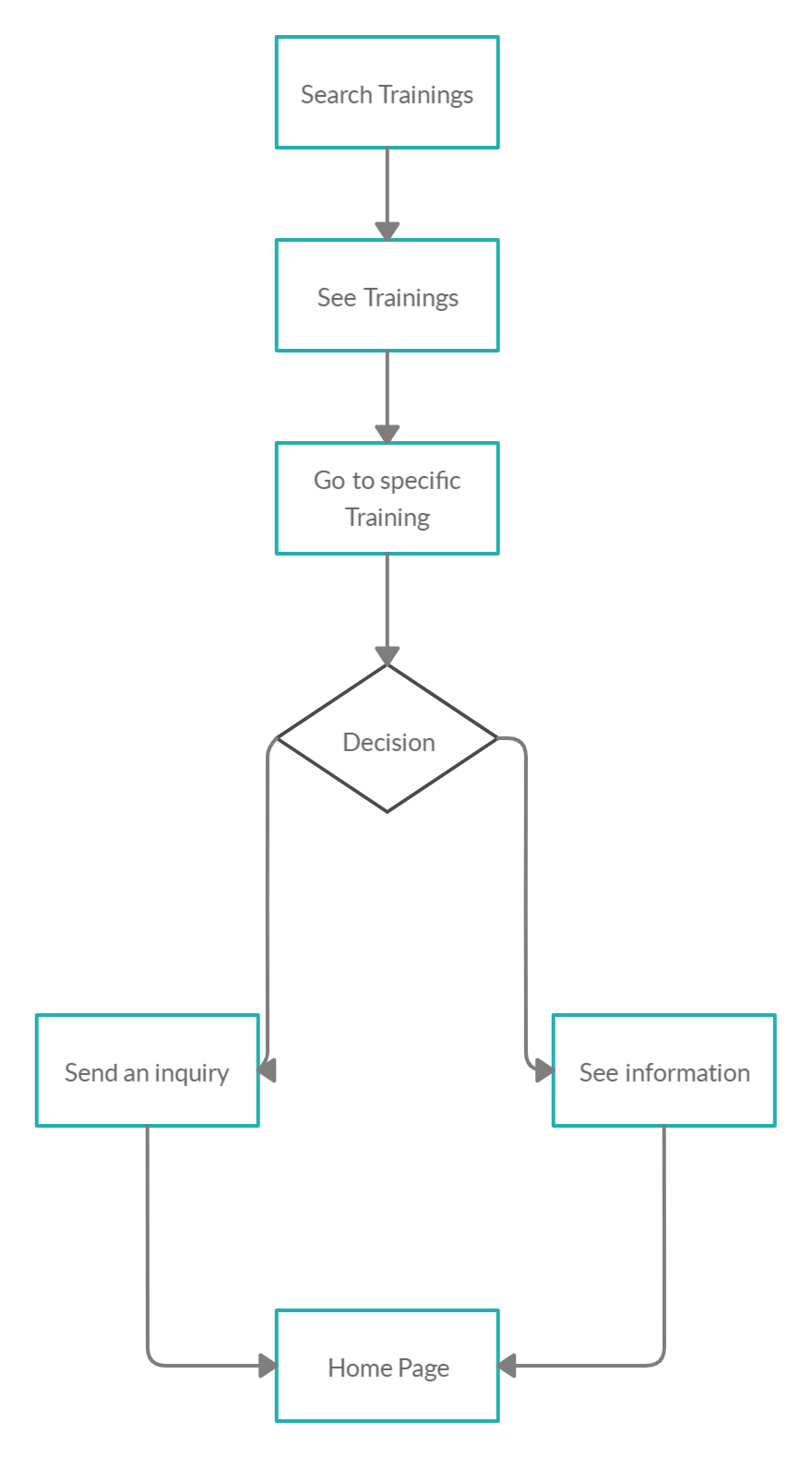


Figura 3.21: Diagrama UML a căutării unui antrenament specific

Diagrama indică faptul că un utilizator care a căutat și a accesat un antrenament specific poate ori să vizualizeze informațiile corespunzătoare antrenamentului respectiv, ori să ofere feedback sau să solicite informații adiționale despre antrenament. La sfârșit se poate întoarce la pagina *Home*.

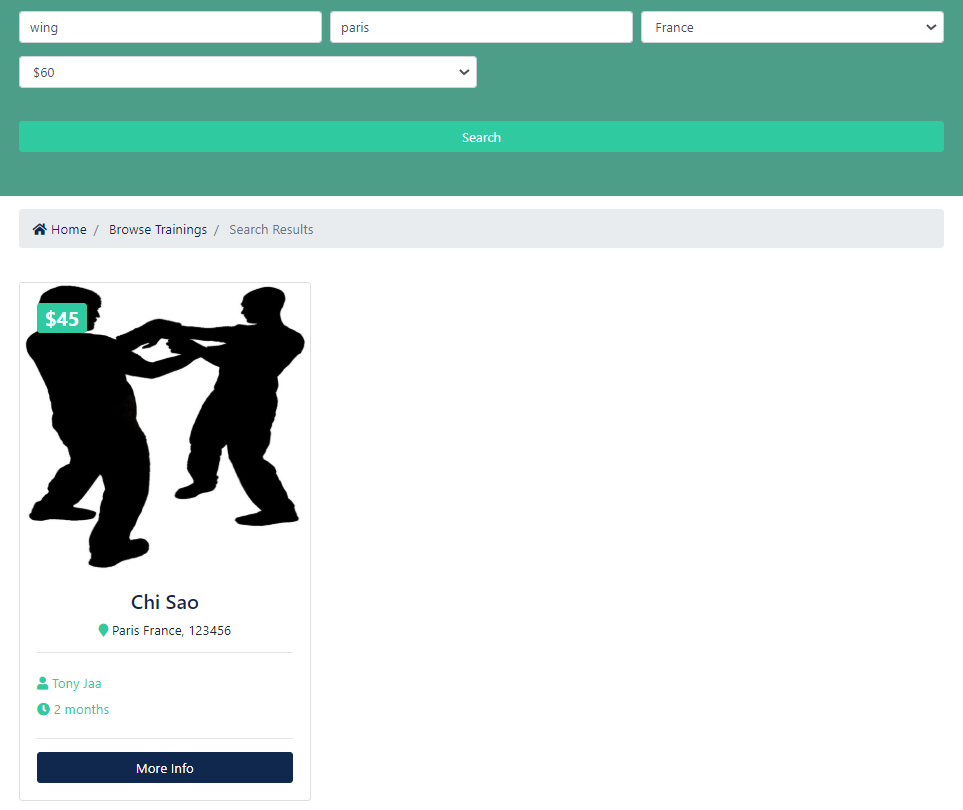


Figura 3.22: Interfața căutării unui antrenament specific

Așa cum se observă utilizatorul poate să vizualizeze pe baza informațiilor introduse antrenamentul dorit pe care poate să-l acceseze prin apăsarea butonului *More info*. De asemenea, se poate duce la toate antrenamentele prin intermediul butonului *Browse Trainings* sau la pagina *Home* prin intermediul butonului cu același nume.

Pentru pagina de căutare:

# Metoda de căutare

def search(request):

    # Se stochează în variabila de tip QuerySet toate antrenamentele din baza de date ordonate descrescător dupa câmpul training\_date

    queryset\_trainings = Training.objects.order\_by('-training\_date')

    # Căutare după cuvinte cheie

    if 'keywords' in request.GET: # dacă apare 'keywords' în URL

        keywords = request.GET['keywords'] # se memorează conținutul

        if keywords: # dacă există cuvinte cheie

            queryset\_trainings = queryset\_trainings.filter(description\_\_icontains=keywords) # se caută cuvintele cheie în întregul paragraf

    # Căutare după oraș

    if 'city' in request.GET: # dacă apare 'city' în URL

        city = request.GET['city'] # se memorează conținutul

        if city: # dacă există un oraș

            queryset\_trainings = queryset\_trainings.filter(city\_\_iexact=city) # se caută exact orașul respectiv

    # Căutare după țară

    if 'country' in request.GET: # dacă apare 'country' în URL

        country = request.GET['country'] # se memorează conținutul

        if country: # dacă există o țară

            queryset\_trainings = queryset\_trainings.filter(country\_\_iexact=country) # se caută exact țara respectivă

    # Căutare după preț

    if 'price' in request.GET: # dacă apare 'price' în URL

        price = request.GET['price'] # se memorează conținutul

        if price: # dacă există un preț

            queryset\_trainings = queryset\_trainings.filter(price\_\_lte=price) # se caută prețurile care sunt mai mici sau egale cu valoarea specificată

    # Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'country\_choices': country\_choices,

        'price\_choices': price\_choices,

        'trainings': queryset\_trainings,

        'values': request.GET

    }

     # Se returnează pagina de căutare

    return render(request,'trainings/search.html',context)

La căutarea după cuvinte cheie se folosește opțiunea *icontains* care “nu ține cont dacă sunt introduse litere mari sau mici” [44], iar căutarea în sine se face pe unul sau multe cuvinte cheie.

La căutarea după oraș și țară se utilizează opțiunea *iexact* care “nu ține cont dacă sunt introduse litere mari sau mici” [44], dar căutarea se face în așa fel trebuie să fie o potrivire exactă cu valorile din dicționar.

La căutarea după preț se utilizează opțiunea *lte* care “înseamă mai mic sau egal.” [44]

# Dicționar pentru prețuri

price\_choices = {

  '10':'$10',

  '20':'$20',

  '30':'$30',

  '40':'$40',

  '50':'$50',

  '60':'$60',

  '70':'$70',

  '80':'$80',

  '90':'$90',

  '100':'$100+', }

# Dicționar pentru țări

country\_choices = {

        'Austria': 'Austria',

        'Belgium': 'Belgium',

        'Bulgaria': 'Bulgaria',

        'Croatia': 'Croatia',

        'Cyprus': 'Cyprus',

        'Czech Republic': 'Czech Republic',

        'Denmark': 'Denmark',

        'Estonia': 'Estonia',

        'Finland': 'Finland',

        'France': 'France',

        'Germany': 'Germany',

        'Greece': 'Greece ',

        'Hungary': 'Hungary',

        'Italy': 'Italy',

        'Ireland': 'Ireland',

        'Latvia': 'Latvia',

        'Lithuania': 'Lithuania',

        'Luxembourg': 'Luxembourg',

        'Malta': 'Malta',

        'Netherlands': 'Netherlands',

        'Poland': 'Poland',

        'Portugal': 'Portugal',

        'Romania': 'Romania',

        'Slovakia': 'Slovakia',

        'Slovenia': 'Slovenia',

        'Spain': 'Spain',

        'Sweden': 'Sweden',

        'United Kingdom': 'United Kingdom',}

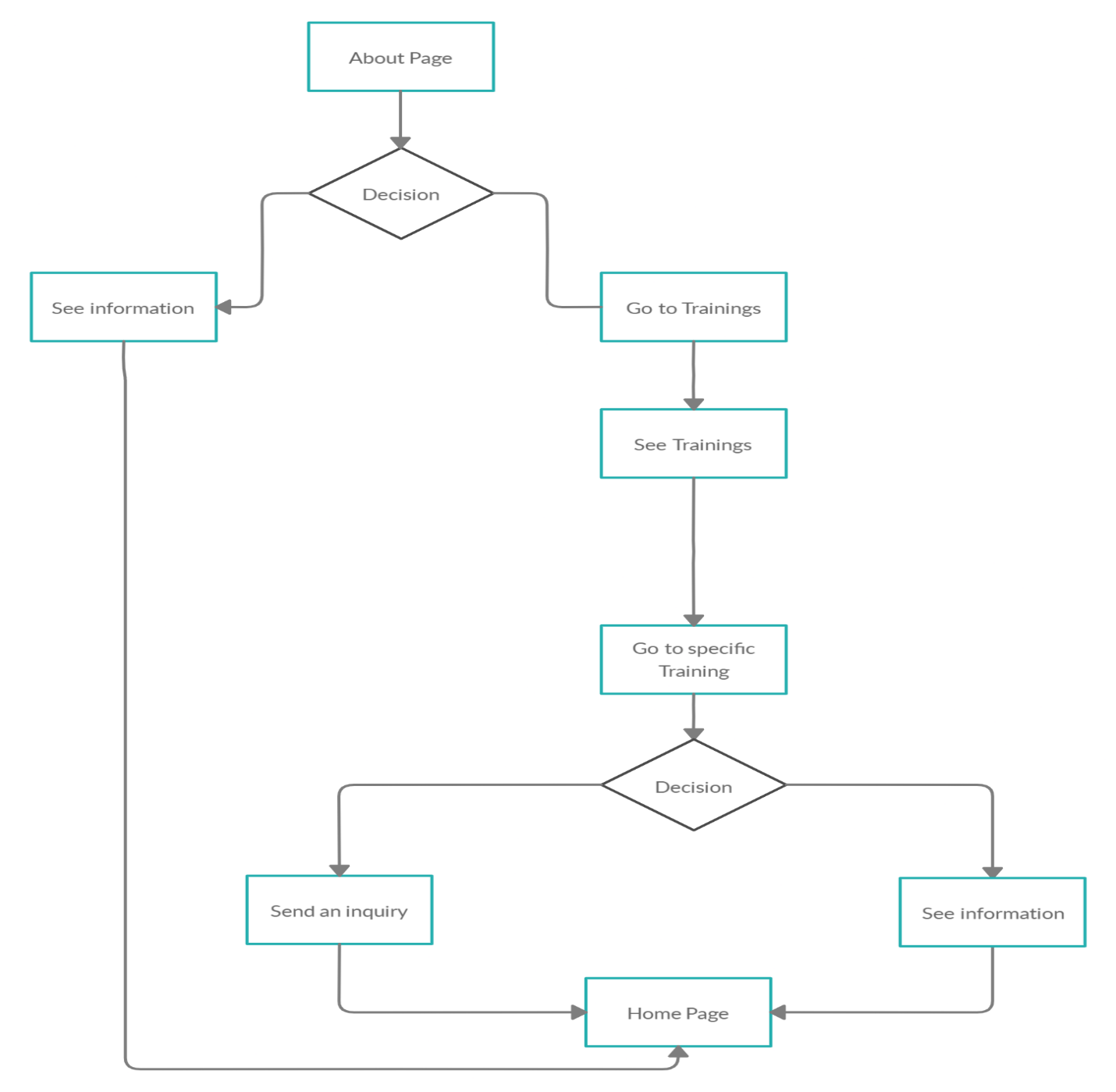


Figura 3.23: Diagrama UML a paginii About

Figura de mai sus dezvăluie opțiunile pe care le are un utilizator atunci când accesează pagina *About* și anume: poate să vadă informațiile necesare sau să meargă la antrenamente și să ofere feedback sau să ceară detalii suplimentare sau să vadă informațiile potrivite. Orice ar alege, la final se poate întoarce la pagina *Home*.



Figura 3.24: Interfața paginii About

După cum se observă în figura de mai sus utilizatorul are următoarele opțiuni: să vizualizeze informații despre stilul de luptă, instructorul sau instructorii lunii, echipa de instructori, să se întoarcă la pagina *Home* prin intermediul aceluiași buton, să meargă la antrenamente prin apăsarea butonului *View Our Trainings*.

Pentru pagina *About*:

# Metoda about pentru pagina About

def about(request):

    #Se iau toți instructorii ordonați descrescător după câmpul employ\_date

    instructors = Instructor.objects.order\_by('-employ\_date')

    #Se iau toți instructorii filtrați după câmpul  month\_instructor

    month\_instructors = Instructor.objects.all().filter(month\_instructor=True)

# Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'instructors': instructors,

        'month\_instructors': month\_instructors

    }

    # Se returnează pagina About

    return render(request,'pages/about.html', context)

Funcționalitatea metodei de mai sus este similară cu cea a metodei *index* pentru pagina *Home*.

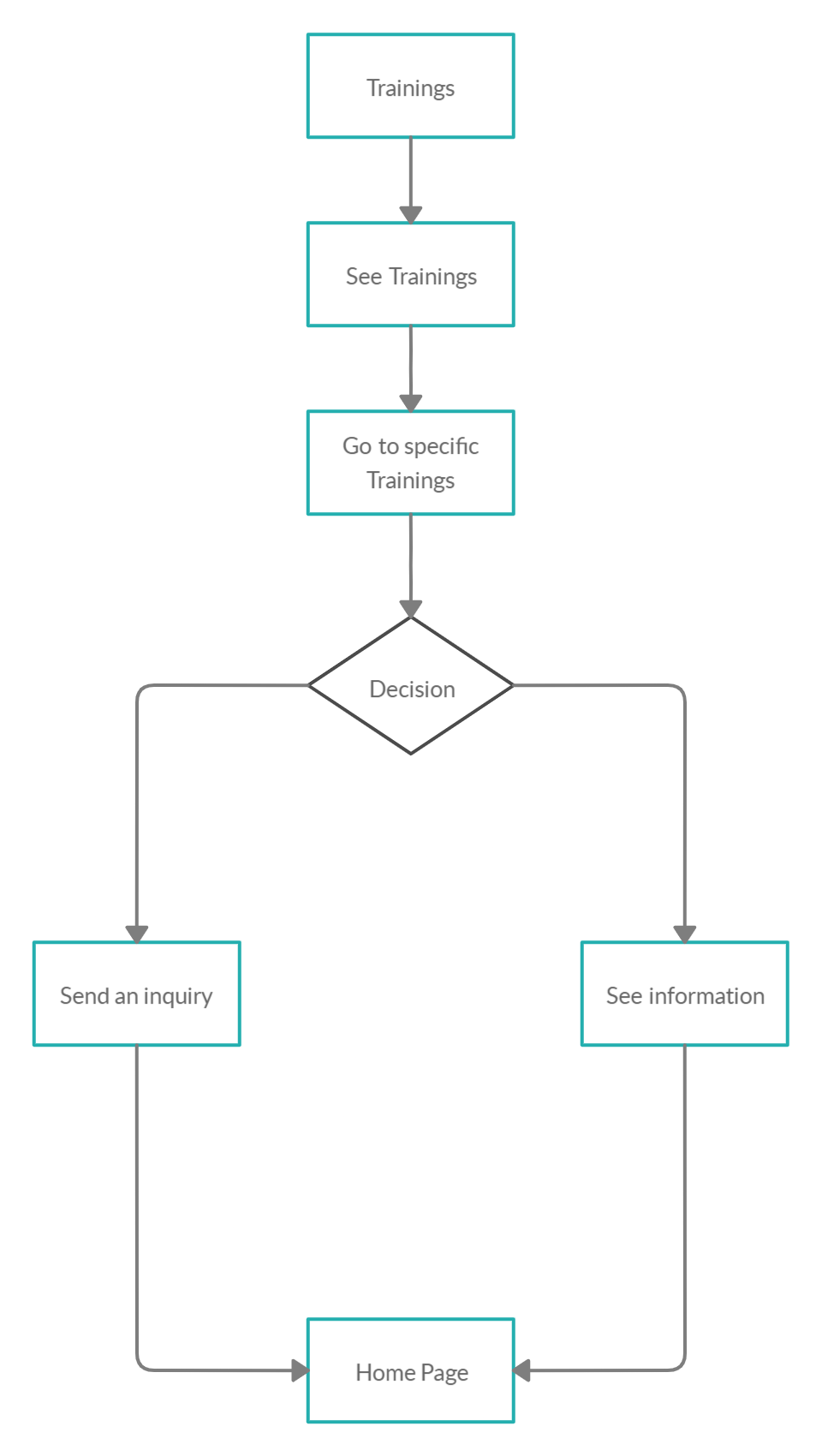


Figura 3.25: Diagrama UML a paginii Trainings

Diagrama arată ce poate face un utilizator dacă accesează pagina *Trainings*: alege un antrenament specific și poate să ceară mai multe informații sau să ofere feedback, precum și să vizualizeze informații despre acel antrenament.

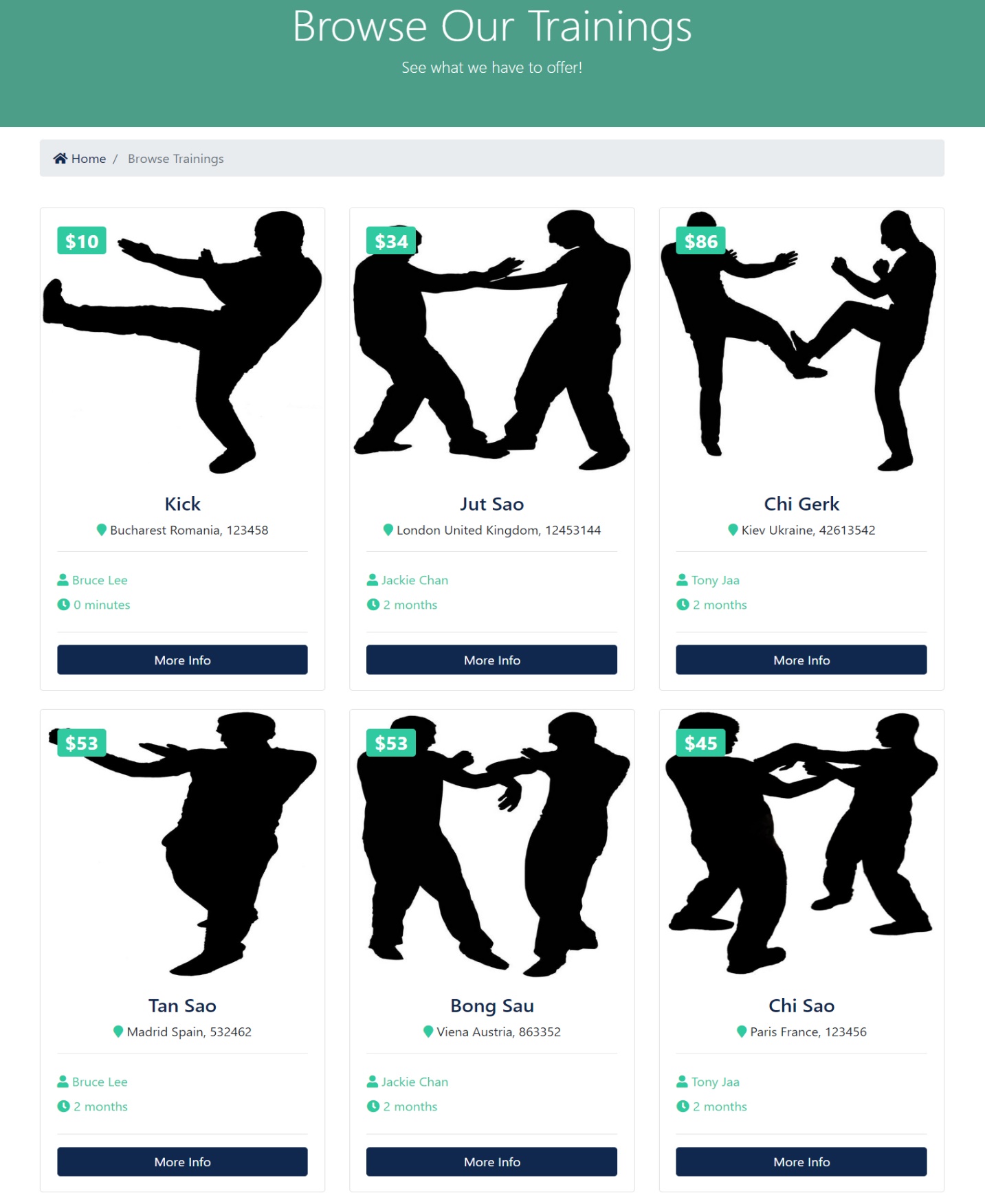


Figura 3.26: Interfața paginii Trainings

După cum se observă utilizatorul poate să vizualizeze orice antrenament disponibil și să-l acceseze prin butonul *More info* sau să meargă la pagina *Home* prin butonul cu același nume.

Pentru pagina *Trainings*:

# Metoda index pentru pagina Trainings

def index(request):

# Get all trainings from database ordered descending and filtered by is\_posted field

# Se iau toate antrenamentele din baza de date ordonate descrescător pe baza câmpului training\_date

    # și filtrate după câmpul is\_posted

    trainings = Training.objects.order\_by('-training\_date').filter(is\_posted=True)

    # Pentru paginare

    paginator = Paginator(trainings, 6)

    page = request.GET.get('page')

    paged\_trainings = paginator.get\_page(page)

# Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'trainings': paged\_trainings

    }

    # Se returnează pagina Trainings

    return render(request,'trainings/trainings.html', context)

“Toate metodele de paginare utilizează clasa *Paginator*. Face tot ce este mai greu în ceea ce privește divizarea efectivă a unui QuerySet în obiecte Page.”[45]

“Dacă i se dă clasei *Paginator* o listă de obiecte, plus numărul de articole care se doresc a fi afișate pe fiecare pagină, aceasta oferă metode pentru accesarea articolelor pentru fiecare pagină.”[45]

Folosind Paginator care ia ca argumente antrenamentele din baza de date și numărul de articole care se doresc a fi afișate pe fiecare pagină, se trece la colectarea numărului paginii din URL (‘*page’* care este parametrul din URL) pe baza *request.GET.get*(‘*page’*) dacă cererea este de tip GET, iar la final se folosește *paginator.get\_page* care ia ca parametru numărul paginii și construiește obiectul *Page* (*paged\_trainings*) care este transmis mai departe în dicționarul *context*.

Metoda *get\_page()* din *Paginator* are rolul de a se ocupa cu “furnizarea modeului documentat de gestionare a numereleor de pagini invalide.”[46]

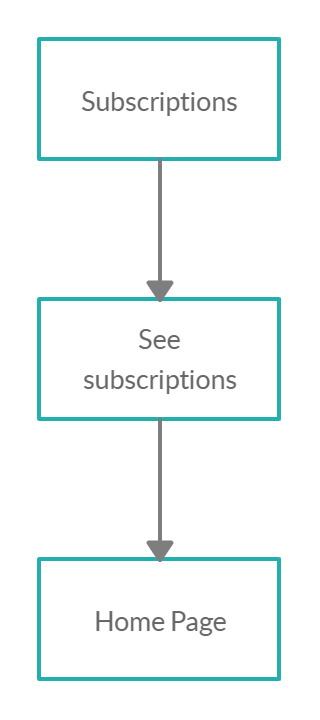


Figura 3.27: Diagrama UML a paginii Subscriptions

Schema de mai sus indică faptul că un utilizator ce accesează pagina *Subscriptions* poate să vadă abonamentele disponibile, iar la final se poate întoarce la pagina *Home*.

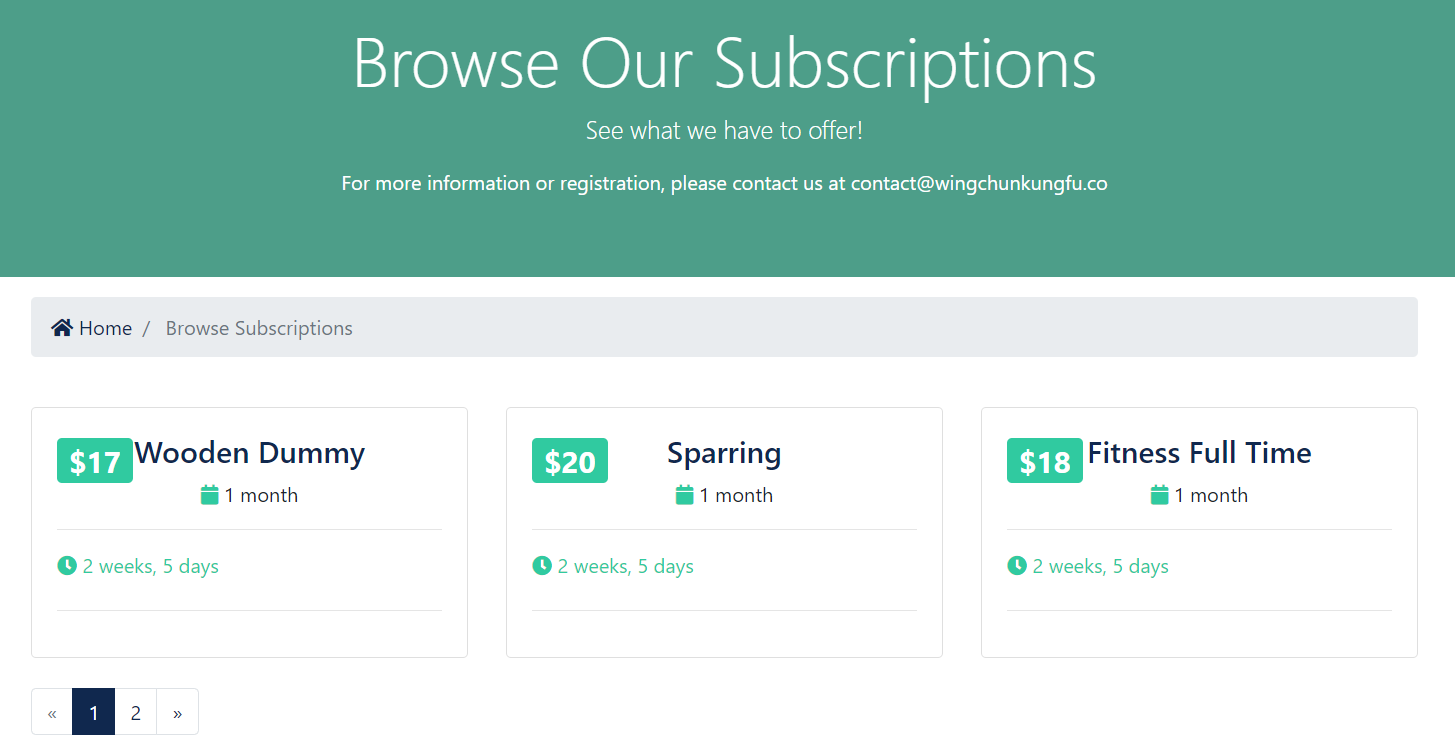


Figura 3.28: Interfața paginii Subscriptions

Pe baza interfeței se poate observa că un utilizator poate vizualiza diferitele abonamente disponibile, precum și că se poate întoarce la pagina *Home* prin apăsarea butonului cu același nume.

Pentru pagina *Subscriptions*:

# Metoda subscription pentru pagina Subscriptions

def subscription(request):

# Se iau toate abonamentele ordonate descrescător după begin\_date și filtrate după is\_posted

    subscriptions = Subscription.objects.order\_by('-begin\_date').filter(is\_posted=True)

    # Pentru paginare

    paginator = Paginator(subscriptions, 3)

    page = request.GET.get('page')

    paged\_subscriptions = paginator.get\_page(page)

# Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'subscriptions': paged\_subscriptions

    }

    # Se returnează pagina Subscriptions

    return render(request, 'catalog/subscription.html', context)

Funcționalitatea metodei de mai sus este similară cu cea a metodei *index* pentru pagina *Trainings*.

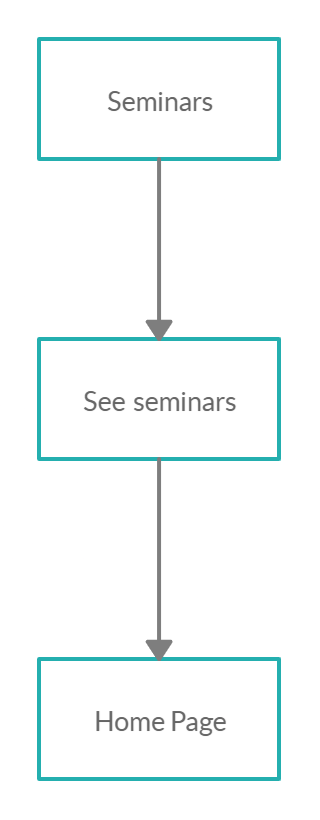


Figura 3.29: Diagrama UML a paginii Seminars

Graficul de mai sus arată faptul că un utilizator ce merge pe pagina *Seminars* poate să vizualizeze seminariile disponibile, iar la final se poate întoarce la pagina *Home*.

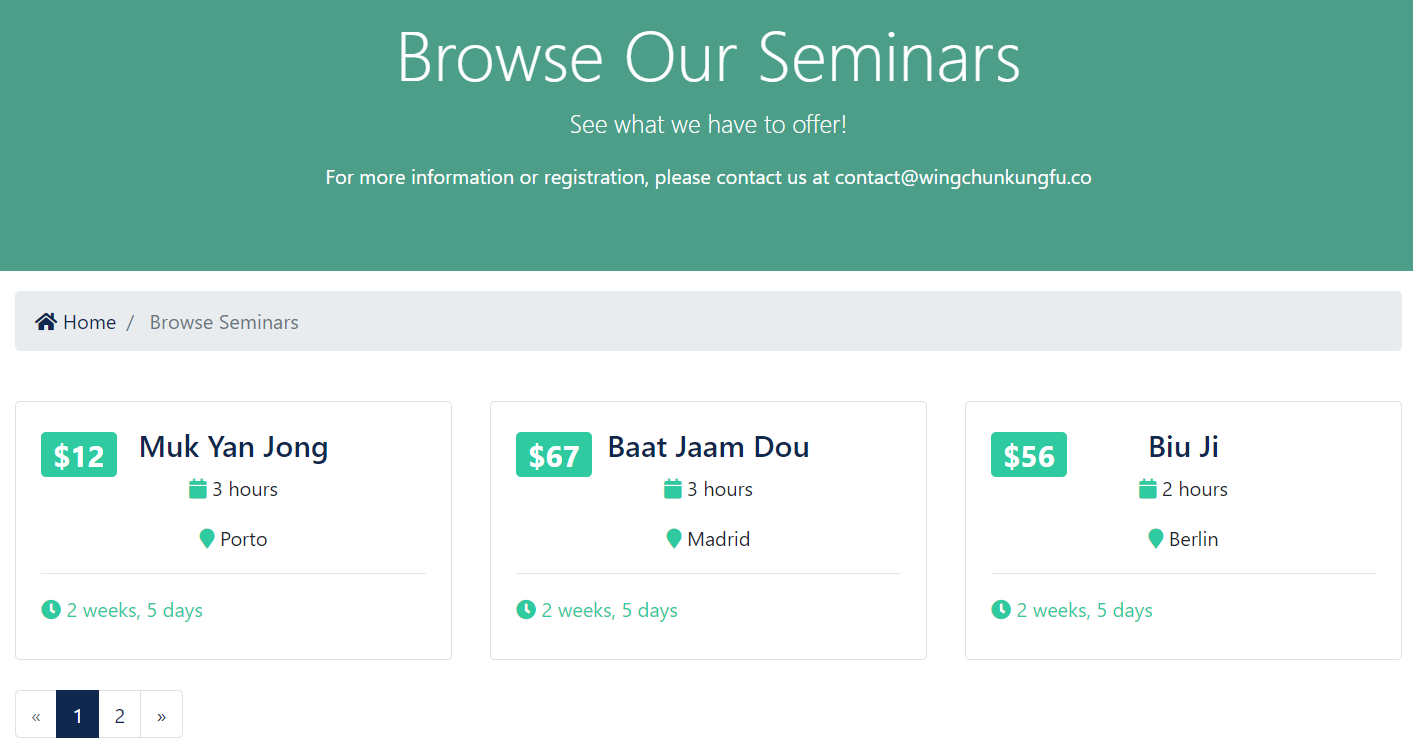


Figura 3.30: Interfața paginii Seminars

Pe baza interfeței se poate observa că un utilizator poate vizualiza diferitele abonamente disponibile, precum și că se poate întoarce la pagina *Home* prin apăsarea butonului cu același nume.

Pentru pagina *Seminars*:

# Metoda seminar pentru pagina Seminars

def seminar(request):

    # Se iau toate seminariile ordonate descrescător după begin\_date și filtrate  după is\_posted

    seminars = Seminar.objects.order\_by('-begin\_date').filter(is\_posted=True)

    # Pentru paginare

    paginator = Paginator(seminars, 3)

    page = request.GET.get('page')

    paged\_seminars = paginator.get\_page(page)

# Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'seminars': paged\_seminars

    }

    # Se returnează pagina Seminars

    return render(request, 'catalog/seminar.html', context)

Funcționalitatea metodei de mai sus este similară cu cea a metodei *index* pentru pagina *Trainings*.

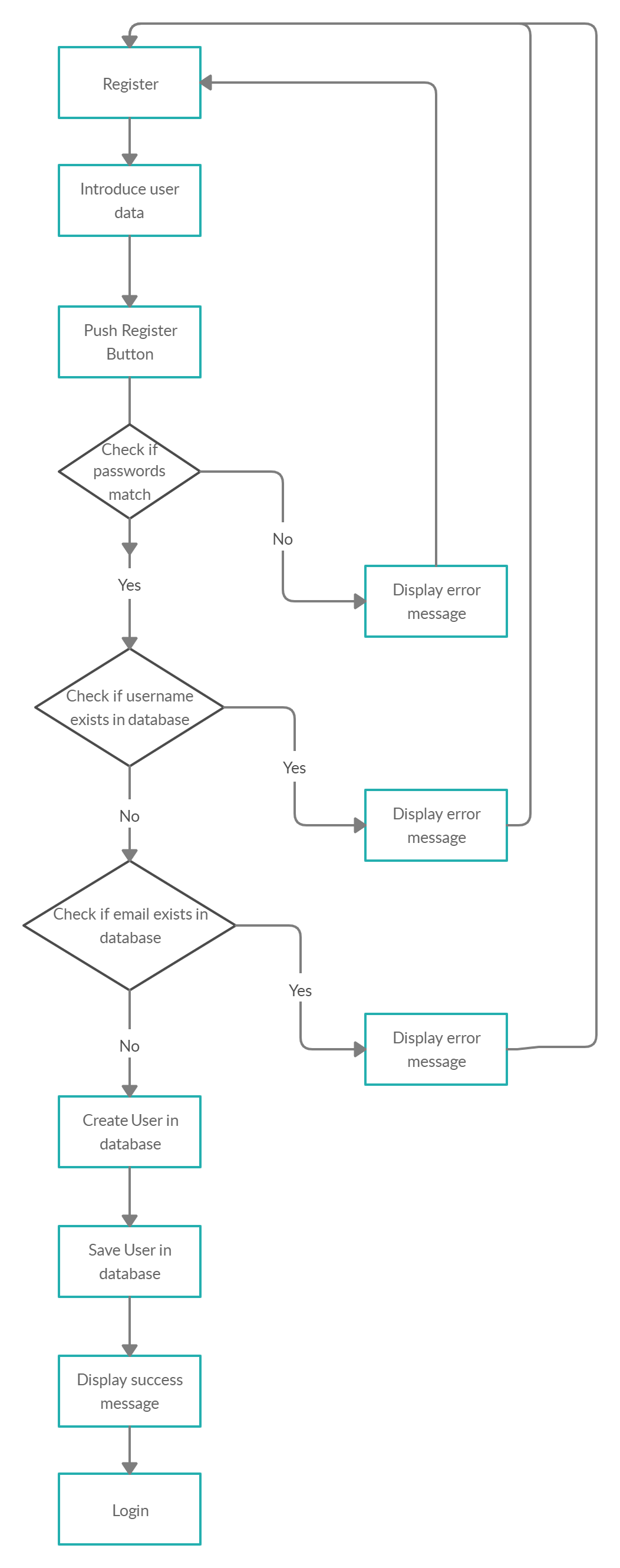


Figura 3.31: Diagrama UML a paginii Register

Diagrama prezintă ce trebuie să facă un utilizator ca să se înregistreze cu succes.

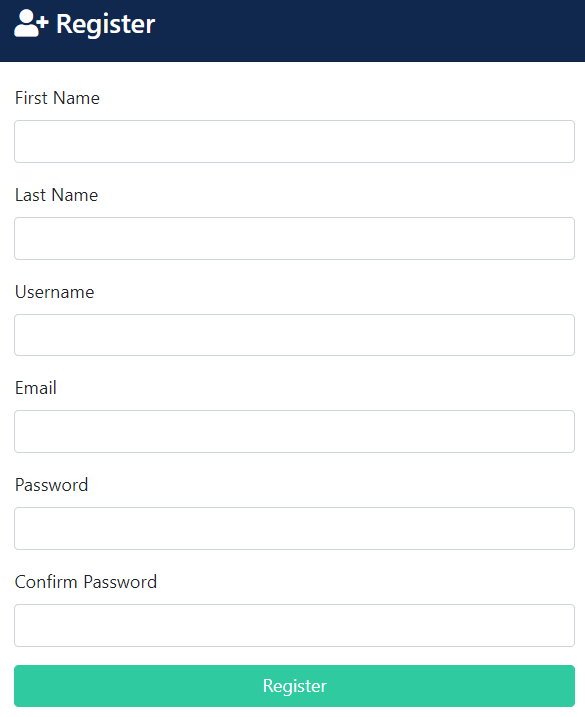


Figura 3.32: Interfața paginii Register



Figura 3.33: Mesaj înregistrare cu succes



Figura 3.34: Mesaj de eroare când numele de utilizator folosit există deja în baza de date



Figura 3.35: Mesaj de eroare când email-ul introdus deja există în baza de date



Figura 3.36: Mesaj de eroare când cele 2 parole nu se potrivesc

Pentru pagina *Register*:

# Metoda register pentru pagina Register

def register(request):

    # Dacă un utilizator face o cerere POST

    if request.method == 'POST':

        # Se iau valorile din formular

        first\_name = request.POST['first\_name']

        last\_name = request.POST['last\_name']

        username = request.POST['username']

        email = request.POST['email']

        password = request.POST['password']

        password2 = request.POST['password2']

        # Se verifică dacă parolele se potrivesc

        if password == password2:

            # Se verifică numele de utilizator dacă există în baza de date

            if User.objects.filter(username=username).exists():

                # Se afișează mesaj de eroare

                messages.error(request, "That username is taken")

                # Se merge la pagina Register

                return redirect('register')

            else:

                # Se verifică daca email-ul există în baza de date

                if User.objects.filter(email=email).exists():

                    # Se afișează mesaj de eroare

                    messages.error(request, "That email is being used")

                    # Se merge la pagina Register

                    return redirect('register')

                else:

                    # The username and the email inserted are not in the database

                    # Numele de utilizator și email-ul nu sunt în baza de date și astfel se poate crea utilizatorul

                    user = User.objects.create\_user(username=username, password=password, email=email, first\_name=first\_name, last\_name=last\_name)

                    # Utilizatorul rămâne autentificat după înregistrare

                    # auth.login(request, user)

                    # messages.success(request, 'You are now logged in')

                    # return redirect('index')

                    # Se salvează utilizatorul

                    user.save()

                    # Se afișează mesaj de succes

                    messages.success(request, 'You are now registered and can log in')

                    # Se merge la pagina Login

                    return redirect('login')

        else:

            # Se afișează mesaj de succes

            messages.error(request, 'Passwords do not match')

            # Se merge la pagina Register

            return redirect('register')

    else:

        # Se merge la pagina Register

        return render(request, 'accounts/register.html')

“Obiectele utilizator sunt nucleul sistemului de autentificare. De obicei, aceștia reprezintă persoanele care interacționează cu site-ul și sunt utilizate pentru a permite lucruri precum restricționarea accesului, înregistrarea profilurilor de utilizator, asocierea conținutului cu creatorii etc. Numai o clasă de utilizator există în cadrul de autentificare Django, adică „superuseri” sau utilizatorii tip „ personalul administrativ” care sunt doar obiecte de utilizator cu seturi de atribute speciale, nu clase diferite de obiecte utilizator.”[47]

“Atributele principale ale unui utilizator implicit sunt: ”[47]

* “nume de utilizator”[47]
* “parola”[47]
* “email”[47]
* “prenume”[47]
* “nume de familie”[47]

“Cea mai directă modalitate de a crea utilizatori este folosirea funcției de ajutor create\_user() inclusă.”[47]

“Metoda de creare utilizator are parametrii impliciți în felul următor create\_user (*nume de utilizator, email = None, parolă = None, \*\* câmpuri suplimentare*).”[47]

“Creează, salvează și returnează un utilizator.”[47]

“*Numele de utilizator* și *parola* sunt setate ca atare. Porțiunea de domeniu de *email* este convertită automat în minuscule, iar obiectul returnat de *utilizator* va fi setat ca *fiind\_activ* pe *True*.”[47]

“Dacă nu este furnizată nicio parolă, *set\_unusable\_password ()* va fi apelat.”[47]

“Argumentele de cuvinte cheie *câmpuri suplimentare* sunt transmise metodei *utilizatorului* *\_\_init\_\_* pentru a permite setarea câmpurilor arbitrare pe un model de utilizator personalizat.”[47]

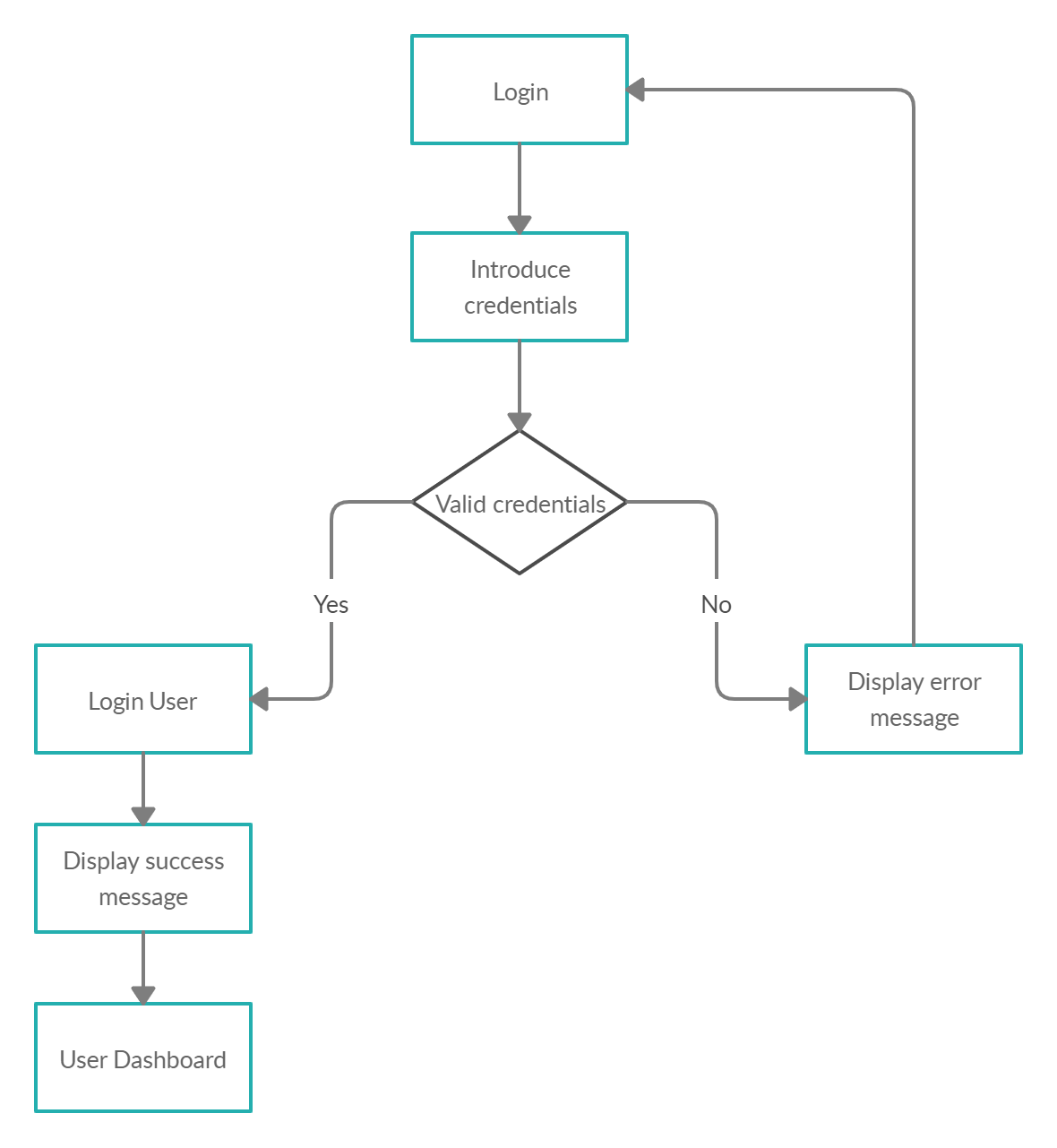


Figura 3.37: Diagrama UML a paginii Login

Diagrama prezintă ce trebuie să facă un utilizator ca să se autentifice cu succes.

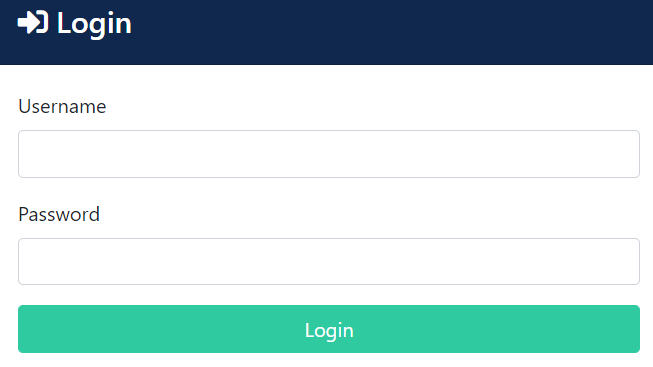


Figura 3.38: Interfața paginii Login



Figura 3.39: Mesaj confirmare logare



Figura 3.40: Mesaj eroare logare

Pentru pagina *Login*:

# Metoda login pentru pagina Login

def login(request):

    # Dacă un utilizator face o cerere POST

    if request.method == 'POST':

        # Se iau valorile din formular

        username = request.POST['username']

        password = request.POST['password']

        #Se realizează autentificarea

        user = auth.authenticate(username=username, password=password)

        # Dacă există un utilizator introdus în formular

        if user is not None:

            # Se face logarea pe baza cererii și utilizatorului

            auth.login(request, user)

            # Se afișează mesaj de succes

            messages.success(request, ' You are now logged in')

            # Se merge la pagina Dashboard

            return redirect('dashboard')

        else:

            # Se afișează mesaj de eroare

            messages.error(request, 'Invalid credentials')

            # Se merge la pagina Login

            return redirect('login')

    else:

        # Se merge la pagina Login

        return render(request, 'accounts/login.html')

“Se folosește *authenticate()* pentru a verifica un set de credențiale. Acesta ia credențiale ca argumente de cuvinte cheie, *nume de utilizator* și *parolă* pentru cazul implicit, le verifică pentru fiecare backend de autentificare și returnează un obiect *utilizator* dacă credențialele sunt valide pentru un backend. Dacă credențialele nu sunt valabile pentru niciun backend sau dacă un backend afișează *PermissionDenied*, acesta returnează *None*.”[47]

“Pentru a conecta un utilizator, dintr-un view, se utilizează *login()*. Este nevoie de un obiect *HttpRequest* și un obiect de *utilizator*. Metoda *login()* salvează ID-ul utilizatorului în sesiune, folosind cadrul de sesiune al lui Django.”[47]

“Este de reținut că orice set de date în timpul sesiunii anonime este păstrat în sesiune după ce un utilizator s-a logat.”[47]

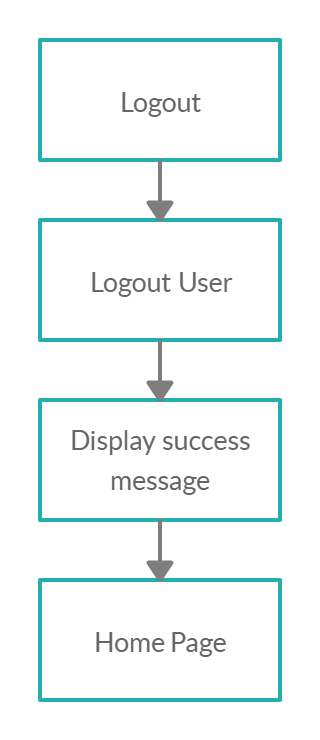


Figura 3.41: Diagrama UML a paginii Logout

Diagrama prezintă ce trebuie să facă un utilizator ca să iasă din cont cu succes.



Figura 3.42: Interfața paginii Logout

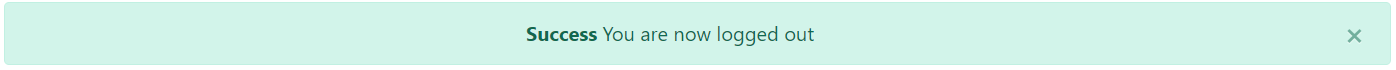


Figura 3.43: Mesaj confirmare delogare

Pentru delogare cont:

# Metoda logout

def logout(request):

    # Dacă un utilizator face o cerere POST

    if request.method == 'POST':

        # Delogare bazată pe request

        auth.logout(request)

        # Se afișează mesaj de succes

        messages.success(request, 'You are now logged out')

        # Se merge la pagina Home

        return redirect('index')

“Pentru a deconecta un utilizator care a fost conectat prin *django.contrib.auth.login()*, se folosește *django.contrib.auth.logout()* în view. Este nevoie de un obiect *HttpRequest* și nu are o valoare returnată.”[47]

“Este de reținut că *logout()* nu aruncă erori dacă utilizatorul nu a fost conectat.”[47]

“Când se apelează *logout()*, datele sesiunii pentru solicitarea curentă sunt complet curățate. Toate datele existente sunt eliminate. Acest lucru este pentru a împiedica o altă persoană să folosească același browser Web pentru a se conecta și a avea acces la datele sesiunii utilizatorului anterior. Dacă se dorește introducerea unor date în sesiunea care va fi disponibilă pentru utilizator imediat după deconectare, se face acest lucru după ce se apelează *django.contrib.auth.logout()*.”[47]

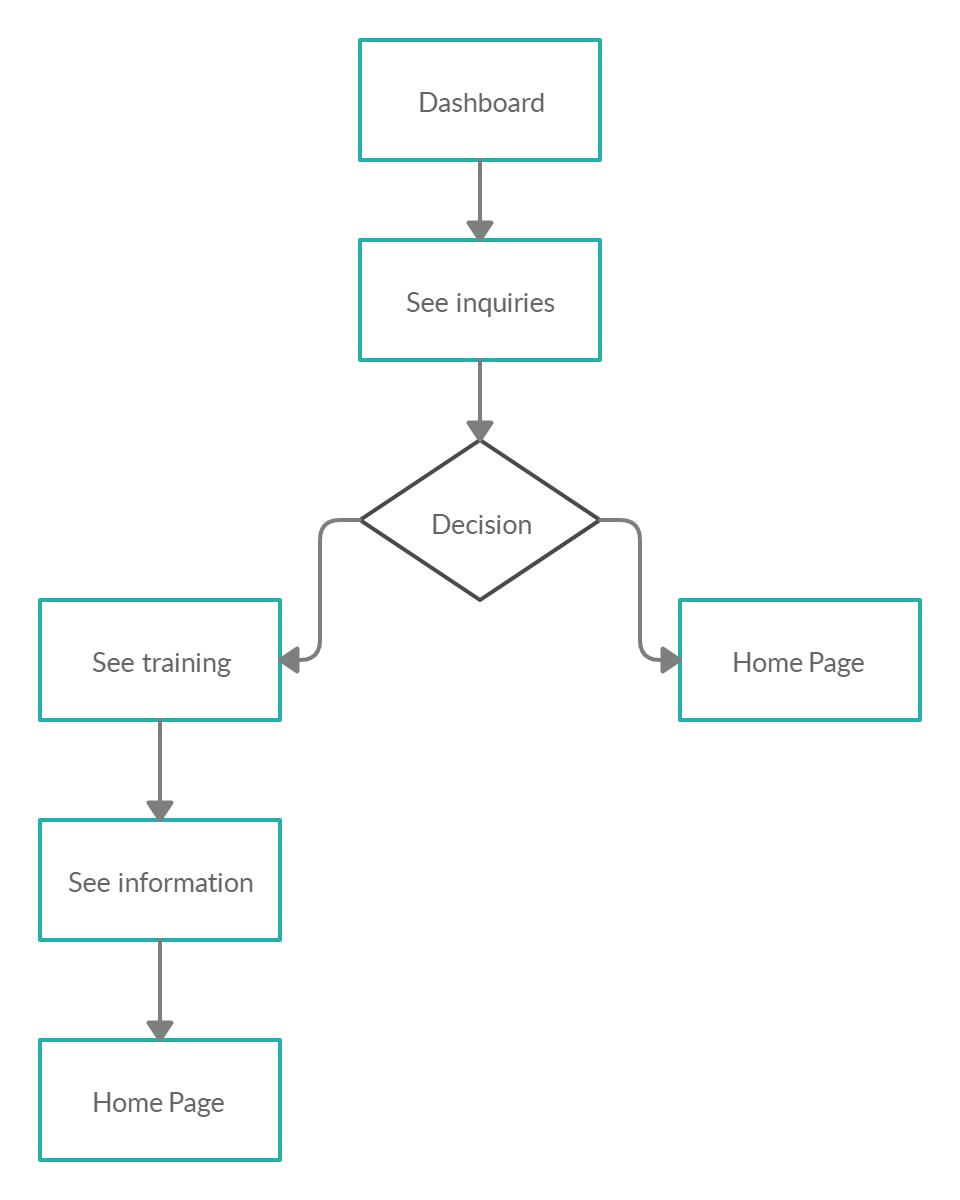


Figura 3.44:Diagrama UML a paginii Dashboard

Diagrama prezintă pașii pe care-i poate face un utilizator când se află în panoul *Dashboard*.

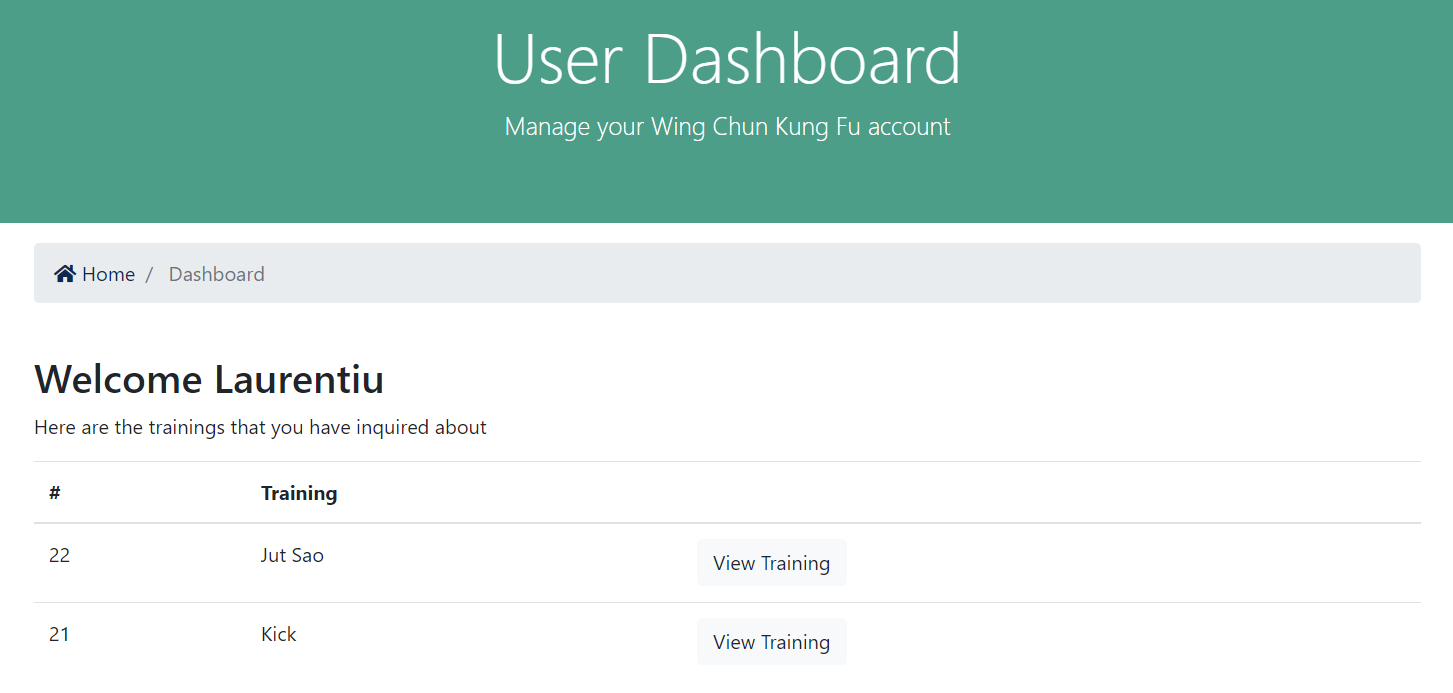


Figura 3.45: Interfața paginii Dashboard

Figura de mai sus arată ce opțiuni are la dispoziție un utilizator care folosește pagina *Dashboard*: să vizualizeze antrenamentele asupra cărora a efectuat cereri prin apăsarea butonului *View Training* sau să meargă la pagina *Home* prin apăsarea butonului cu același nume.

Pentru pagina Dashboard:

# Metoda dashboard pentru pagina Dashboard

def dashboard(request):

    # Se iau toate contactele din baza de date ordonate descrescător după câmpul contact\_date și filtrate după id-ul utilizatorului

    contacts\_of\_the\_user = Contact.objects.order\_by('-contact\_date').filter(user\_id=request.user.id)

    # Dicționarul context este folosit pentru a arăta conținutul bazei de date în frontend

    context = {

        'contacts': contacts\_of\_the\_user

    }

    # Se returnează pagina Dashboard

    return render(request, 'accounts/dashboard.html',context)

Funcționalitatea metodei de mai sus este similară cu cea a metodei *index* pentru pagina *Home*.

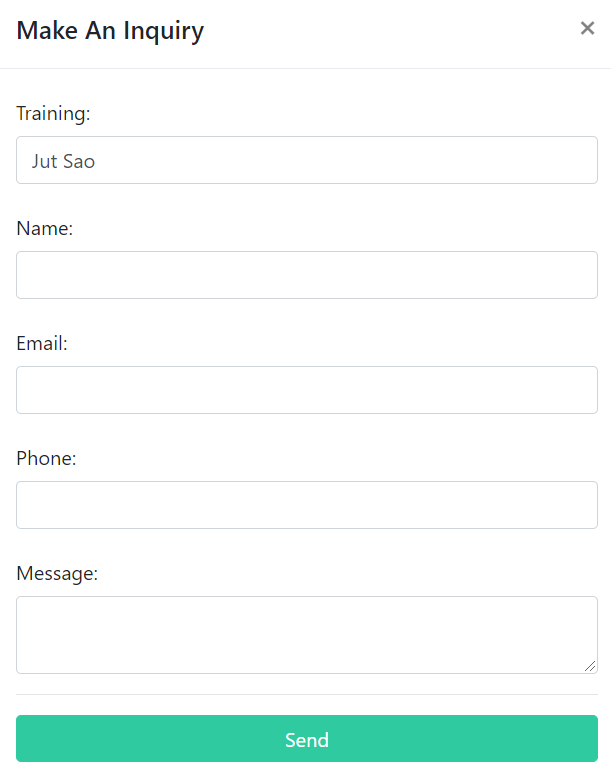


Figura 3.46: Formularul când utilizatorul nu este autentificat

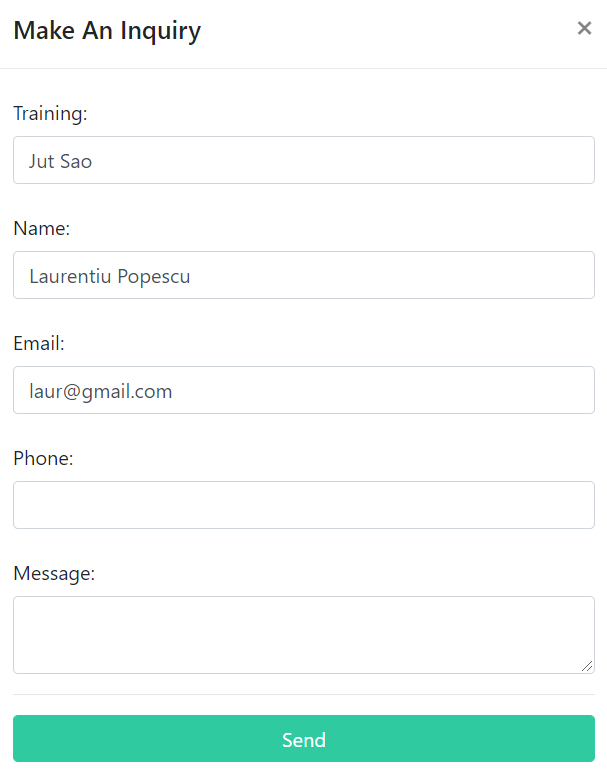


Figura 3.47: Formularul când utilizatorul este autentificat



Figura 3.48: Mesaj confirmare cerere de informații suplimentare/feedback



Figura 3.49: Mesaj de eroare când s-a mai făcut o cerere de informații suplimentare/feedback asupra aceluiași antrenament

Codul care face ca din moment ce utilizatorul este autentificat să fie introduse automat datele în câmpurile *Name* și *Email* este:

{% if user.is\_authenticated %} value="{{ user.first\_name }} {{ user.last\_name }}" {% endif %}

{% if user.is\_authenticated %} value="{{ user.email }}" {% endif %}

Atributul *is\_authenticated* este “atribut numai de citire, care este întotdeauna *True* (spre deosebire de *AnonymousUser.is\_authenticated* care este întotdeauna *False*). Acesta este un mod de a spune dacă utilizatorul a fost autentificat. Aceasta nu implică permisiuni și nu verifică dacă utilizatorul este activ sau are o sesiune validă.”[48]

În continuare se va prezenta codul pentru funcționarea formularului de cerere de informații suplimentare/feedback care conține și configurarea email-urilor la care se trimite acest formular.

Pentru formular:

# Metoda contact pentru formular

def contact(request):

    # Dacă un utilizator face o cerere POST

    if request.method == 'POST':

        # Se iau valorile din formular

        training\_id = request.POST['training\_id']

        training = request.POST['training']

        name = request.POST['name']

        email = request.POST['email']

        phone = request.POST['phone']

        message = request.POST['message']

        user\_id = request.POST['user\_id']

        instructor\_email = request.POST['instructor\_email']

        # Se verifică dacă utilizatorul a făcut o cerere de informații suplimentare/feedback

        if request.user.is\_authenticated:

            # Se ia id-ul utilizatorului

            user\_id == request.user.id

            # Se iau toate contactele pe baza antrenamentului curent și utilizatorului curent

            has\_already\_contacted = Contact.objects.all().filter(training\_id=training\_id, user\_id=user\_id)

            # Dacă există contacte

            if has\_already\_contacted:

                # Se afișează mesaj de eroare

                messages.error(request, 'You have already made an inquiry for this training')

                # Se merge la antrenamentul specific pe baza id-ului

                return redirect('/trainings/'+training\_id)

        # Se creează contactul în baza de date

        contact = Contact(training=training, training\_id=training\_id, name=name, email=email, phone=phone, message=message, user\_id=user\_id)

        # Se salvează contactul

        contact.save()

        # Se trimite email numai după ce contactul a fost salvat

        send\_mail(

            'Inquiry about Training ', # Subiect

            'There has been an inquiry for ' + training + '. Sign into the admin panel for more info', # Mesaj

            'from@example.com',# from@example.com

            [instructor\_email , 'to@example.com'], # to@example.com

            fail\_silently=False

        )

        # Se afișează mesaj de succes

        messages.success(request, 'Your request has been submitted, an instructor will get back to you soon')

        # Se returnează pagina de antrenament

        return redirect('/trainings/'+training\_id)

Trebuie făcute modificările necesare și în fișierul **settings.py** pentru email:

# Configurare email

EMAIL\_HOST = 'smtp.gmail.com'

EMAIL\_PORT = 587 #port implicit pentru gmail

EMAIL\_HOST\_USER = 'example@gmail.com' # adresa de email dorită

EMAIL\_HOST\_PASSWORD = 'example' # parola adresei de email

EMAIL\_USE\_TLS = True#se precizează ce protocol de securitate să se folosească(aici TLS) pentru a se conecta la server