**Foaie de Titlu**

**Cuprins**

**Introducere**

În contextul digital actual, aplicațiile web reprezintă elemente esențiale în interacțiunea dintre utilizatori și sisteme informatice, indiferent de domeniu – de la comerț și educație, la sănătate și administrație publică. Odată cu creșterea complexității acestor aplicații, este necesară utilizarea unor arhitecturi software fiabile, scalabile și ușor de întreținut. Modelele arhitecturale (architectural patterns) răspund acestei cerințe, oferind structuri standard pentru organizarea componentelor aplicației și pentru definirea modului în care acestea interacționează. Spre deosebire de șabloanele de design orientate pe obiecte, care vizează soluții locale la nivel de cod, modelele arhitecturale tratează aspecte globale ale sistemului, influențând performanța, mentenabilitatea, securitatea și scalabilitatea.Top of Form

Această lucrare își propune să analizeze în profunzime principalele modele arhitecturale clasice utilizate în dezvoltarea aplicațiilor web, cu accent pe Model-View-Controller (MVC), Model-View-Presenter (MVP) și Model-View-ViewModel (MVVM). Aceste arhitecturi au avut un rol esențial în structurarea aplicațiilor, oferind un cadru clar pentru separarea logicii de prezentare, a datelor și a interfeței cu utilizatorul. Lucrarea evidențiază contribuția acestor modele la separarea responsabilităților, creșterea testabilității și facilitarea mentenanței codului. Arhitecturile moderne sunt menționate doar pentru a contextualiza evoluția paradigmelor software, fără a constitui obiectul analizei detaliate. Metodologia include o analiză teoretică a acestor modele, comparații bazate pe criterii calitative și un studiu de caz care ilustrează aplicarea arhitecturii MVC într-un proiect concret.

Lucrarea este structurată în trei capitole principale. Primul capitol introduce conceptele de bază ale șabloanelor arhitecturale și oferă o clasificare a acestora în trei categorii: structurale, de comunicare și distribuite. Al doilea capitol prezintă o analiză comparativă a modelelor MVC, MVP și MVVM, axată pe modul de interacțiune dintre componente și efectele asupra testabilității și mentenanței aplicațiilor. În capitolul al treilea este prezentat un studiu de caz care ilustrează implementarea arhitecturii MVC într-o aplicație web pentru gestionarea programărilor medicale, incluzând cerințele funcționale și nefuncționale, tehnologiile utilizate și o evaluare a performanței sistemului.

**Bottom of Form**

**Capitolul 1. Șabloane de proiectare la dezvoltarea aplicațiilor web**

**1.1 Introducere în Șabloanele de Arhitectură Software**

Șabloanele de arhitectură software (cunoscut și sub numele de *Architectural Patterns*) reprezintă un set de soluții general acceptate și reutilizabile folosite pentru pentru organizarea structurii unui sistem software. Spre deosebire de șabloanele de proiectare orientate pe obiecte, care abordează soluții locale în cadrul codului sursă, șabloanele de arhitectură operează la un nivel mai înalt, descriind modul în care componentele principale ale unei aplicații interacționează, sunt separate și comunică între ele [1].

Un șablon arhitectural oferă o viziune standardizată asupra modului în care poate fi concepută arhitectura unui sistem complex, luând în considerare atât cerințele funcționale, cât și cele de calitate, precum scalabilitatea, performanța, mentenabilitatea sau securitatea. Exemple notabile includ Model-View-Controller (MVC), Microservices, Layered Architecture sau Event-Driven Architecture. Acestea sunt cadre conceptuale ce pot fi adaptate în funcție de contextul aplicației [1][3].

În contextul dezvoltării aplicațiilor web sau enterprise, unde complexitatea este ridicată și cerințele se pot modifica frecvent, șabloanele de arhitectură permit o abordare structurată. De exemplu, utilizarea arhitecturii pe straturi (Layered Architecture) permite separarea clară între prezentare, logică de business și acces la date, făcând posibilă testarea și modificarea fiecărui strat independent [1].

Adoptarea unui șablon arhitectural presupune o înțelegere clară a *atributelor de calitate* pe care sistemul trebuie să le îndeplinească – precum disponibilitate, timp de răspuns, etc. Cartea lui Cervantes și Kazman propune o metodologie clară – *Attribute-Driven Design (ADD)* – pentru a transforma cerințele în decizii arhitecturale, selectând șabloane potrivite care pot susține aceste atribute [1].

Prin urmare, utilizarea șabloanelor de arhitectură nu doar că simplifică proiectarea inițială, dar și facilitează extinderea și întreținerea sistemului pe termen lung. Ele asigură o separare clară a responsabilităților, o scalabilitate crescută și o bază solidă pentru colaborarea în echipe mari de dezvoltare [1][3].

**1.2 Clasificarea Șabloanelor Arhitecturale**

**1.2.1 Structurale**

Aceste șabloane definesc modul de organizare ierarhică sau stratificată a componentelor unei aplicații. Accentul este pus pe separarea responsabilităților și pe claritatea structurii generale.

*Layered Architecture* - Aceasta este una dintre cele mai comune arhitecturi software [1]. Modelul împarte sistemul în mai multe straturi bine definite, cum ar fi interfața utilizator (UI), logica de business și accesul la date. Această separare facilitează testarea, mentenanța și înțelegerea aplicației. Arhitectura pe straturi este deosebit de potrivită pentru aplicații enterprise sau aplicații desktop tradiționale.

*Client-Server* - Tratat ca o variantă structurală esențială pentru aplicațiile distribuite, modelul client-server presupune existența unui server care oferă servicii și a unuia sau mai multor clienți care consumă aceste servicii. Este un model scalabil, dar poate duce la blocaje dacă serverul devine supraîncărcat. În ciuda acestei limitări, rămâne un șablon de bază în arhitectura sistemelor distribuite[1].

*Microservices* - În contextul nevoii de separare și autonomie a componentelor, arhitectura pe microservicii presupune ca fiecare serviciu să fie independent și să comunice cu celelalte prin API-uri bine definite. Această abordare este extrem de scalabilă și se potrivește foarte bine mediilor cloud. Totuși, pentru a fi eficientă, necesită o infrastructură solidă și un grad ridicat de automatizare, implicând bune practici de tip DevOps. Este o soluție modernă, adoptată frecvent în dezvoltarea aplicațiilor de mari dimensiuni.

Această categorie este extrem de relevantă pentru proiectele care necesită separarea clară a responsabilităților și extindere în timp.

**1.2.2 De comunicare și interacțiune**

Aceste șabloane gestionează fluxul de informații și interacțiunea dintre componente, în special în sisteme reactive sau distribuite.

*Event-Driven Architecture (EDA)* -Prezentată ca o arhitectură potrivită pentru sisteme care trebuie să reacționeze rapid la evenimente, EDA este utilizată frecvent în domenii precum finanțele sau Internet of Things (IoT). În acest model, o componentă, numită *producer*, generează un eveniment care este apoi interceptat și procesat de una sau mai multe componente consumatoare (*consumers*). Această abordare oferă o separare între componente și permite o mai mare flexibilitate și scalabilitate în tratarea fluxurilor de date dinamice[1].

*Message Bus* - Considerat o extensie a arhitecturii bazate pe evenimente, busul de mesaje este de obicei menționat în contextul pattern-urilor de integrare (*integration patterns*). Acesta permite mai multor servicii să comunice asincron printr-un canal comun de mesaje, eliminând dependențele directe între emițători și receptori. Este deosebit de util în arhitecturi complexe, cum ar fi cele întâlnite în integrarea sistemelor enterprise, unde coordonarea componentelor separate este esențială pentru funcționarea fluentă a întregului sistem [1].

Aceste șabloane sunt utile acolo unde componentelor li se cere să reacționeze dinamic, fără legături directe.

**1.2.3 Distribuite**

Acestea definesc modele de organizare în care componentele sunt răspândite pe mai multe noduri (mașini, servere), deseori în rețea. Aceste șabloane sunt relevante în sisteme distribuite cu toleranță la erori și nevoi ridicate de scalare.

*Service-Oriented Architecture (SOA)* - SOA este menționat ca un precursor al arhitecturii pe microservicii. În acest model, o aplicație este construită dintr-un set de servicii distincte care comunică între ele prin protocoale standardizate, cum ar fi SOAP. Accentul principal este pus pe interoperabilitatea între aplicații și platforme distincte după structură, ceea ce face ca SOA să fie potrivită pentru medii enterprise în care este necesară integrarea unor sisteme diverse. Deși mai rigidă decât microserviciile, SOA a fost esențială în evoluția arhitecturilor distribuite [1].

*Peer-to-Peer (P2P)* - O alternativă descentralizată la modelul traditional client-server, arhitectura Peer-to-Peer permite fiecărui nod din rețea să acționeze simultan atât ca server, cât și ca client. Această structură elimină punctele centrale de control, oferind scalabilitate și reziliență crescută. P2P este ideală pentru aplicații de partajare de fișiere, rețele blockchain sau alte sisteme în care distribuția responsabilităților și a resurselor este esențială [1].

* + 1. **De control și acces la date**

Acestea modelează **modul în care datele sunt procesate și gestionate** într-o aplicație, incluzând transformări, filtrări și partajare de informații. Șabloanele sunt potrivite pentru aplicații ce necesită manipulare complexă de date sau centralizarea informației.

**Pipe and Filter** - Arhitectura *Pipe and Filter* este un model conceput pentru procesarea secvențială a datelor printr-un lanț de componente. În acest model, fiecare componentă – denumită *filtru* – procesează datele primite și transmite rezultatul către următoarea componentă printr-un canal numit *conductă* (*pipe*). Această abordare modulară este extrem de utilă în aplicații precum compilatoarele, procesarea de conținut media sau fluxurile de transformare a datelor (*data transformation pipelines*), oferind flexibilitate și reutilizare a componentelor [1].

**Repository Architecture** - Descris ca un șablon arhitectural în care toate componentele accesează și partajează o bază de date comună, arhitectura de tip *Repository* este ideală pentru aplicații care impun un nivel ridicat de consistență și integritate a datelor. În astfel de sisteme, repository-ul central acționează ca un depozit comun de informații, facilitând coordonarea și sincronizarea între componente. Este frecvent utilizată în domenii critice precum sistemele medicale sau bancare, unde coerența datelor este esențială [1].

**1.3 Importanța în inginerie**

Alegerea unui șablon arhitectural nu este un act arbitrar, ci un proces profund ghidat de cerințele sistemului – în special de cele non-funcționale (cunoscute și sub denumirea de atribute de calitate). Printre cele mai relevante se numără: *performanța, scalabilitatea, disponibilitatea, posibilitatea de modificare și securitatea* – fiecare având un impact direct asupra structurii generale a aplicației și asupra deciziilor de proiectare [1].

Conform metodologiei propuse de Cervantes și Kazman în *Designing Software Architectures*, selecția șabloanelor arhitecturale este strâns legată de analiza cerințelor. Autorii propun o abordare sistematică, denumită *Attribute-Driven Design (ADD)*, care pornește de la cerințele de calitate și conduce spre decizii arhitecturale concrete [1]. De exemplu, într-un sistem care solicită timp de răspuns scăzut și disponibilitate ridicată, pot fi preferate arhitecturi bazate pe evenimente (Event-Driven) sau microservicii. În schimb, un sistem ce vizează modularitatea și ușurința mentenanței poate beneficia de o arhitectură stratificată (Layered) [1][3].

Aceste atribute de calitate nu doar influențează alegerea arhitecturii, ci și determină tacticile și compromisurile implicate. Spre exemplu, pentru a crește securitatea, pot fi introduse componente suplimentare pentru controlul accesului și criptare – măsuri ce pot afecta performanța generală. În mod similar, adoptarea unei arhitecturi de tip *Repository* facilitează consistența datelor într-un sistem distribuit, dar introduce un potențial punct unic de eșec, care poate afecta disponibilitatea [1].

Prin urmare, șabloanele arhitecturale fac legătura dintre ceea ce trebuie să facă un sistem și modul în care acesta este construit. Ele au un rol important în definirea cerințelor, deoarece transformă intențiile legate de calitate în structuri concrete și ajută la stabilirea modului în care sistemul va răspunde, în timp real, la schimbările și cerințele din mediul în care funcționează.

**Capitolul 2. Analiza șabloanelor de proiectare**

Deși șabloanele precum *MVC (Model–View–Controller)*, *MVP (Model–View–Presenter)* și *MVVM (Model–View–ViewModel)* au fost descrise anterior ca șabloane de proiectare orientate pe interfața utilizatorului, acestea pot fi încadrate și într-o *subcategorie specifică a șabloanelor arhitecturale*, cunoscută sub denumirea de *UI architectural patterns* sau *Presentation Layer Patterns* [1][2].

Aceste șabloane sprijină principiile de separare a responsabilităților și organizare clară a logicii de prezentare, permițând o mai bună testabilitate și posibilitatea de modificare a aplicațiilor [1]. Deși nu definesc arhitectura întregului sistem, ele sunt adesea încorporate în arhitecturi mai ample precum *Layered Architecture* sau *Microservices*, contribuind esențial la definirea și organizarea stratului de prezentare [1][3].

Astfel, MVC, MVP și MVVM se poziționează la intersecția dintre design și arhitectură, fiind considerate *șabloane structurale UI-centric*, cu aplicabilitate practică ridicată în aplicațiile moderne, în special cele web sau mobile [2].

**2.1 MVC(Model-View-Controller)**

Trygve Reenskaug a fost informatician norvegian care a adus mari contribuții în popularizarea POO (programării orientate pe obiecte) și a influențat evoluția modelelor de dezvoltare a aplicațiilor moderne**.** Model-View-Controller a fost introdus ca concept de Trygve între anii 1970 și 1980 pentru a rezolva problema controlului a mai multor utilizator asupra unui set enorm și complex de date, modelul fiind implementat deja pentru limbaju de programare Smalltalk-80 [6][7].

**2.1.1 Arhitectura MVC**

MVC implementează separarea aplicația dezvoltată in trei componente distincte: componenta de intrare, componenta de ieșire și cea de procesare a informației[6]. Modelul (Model) face legătura dintre aplicație și date stocate în memorie sau într-o bază de date. Vizualizarea (View) afișează datele în fața utilizatorului sau permite interacțiunea cu aplicația prin interfața de utilizator. Controlerul (Controller) deja descrie logica de transmitere a datelor de la model la vizualizare și executarea interacțiunilor cu aplicația.

Fiecare componenta poate interacționa cu celelalte, de exemplu o vizualizare afișează datele unui model, controlerul procesează o interacțiune cu utilizatorul de la o vizualizare sau modelul actualizează datele din bază de date transmise de controler, reprezentat în figura 2.1. În cel mai des caz vizualizarea și controlerul sunt luați ca o pereche din simplu motiv că au o interacțiune mai strânsă în modelul MVC. Fiecare vizualizare are la baza un controler, iar fiecare controler este are setul său de vizualizări pe care le gestionează [5][8].

Acces la date

Notificări

Răspunsul primit la executare unei cerei

Chemarea unei funcționalități

Actualizarea datelor

Extragerea obiectului model

figura 2.1 Reprezentarea interacțiunilor între componentele MVC

**2.1.2 Modelele**

Modelul este un obiect ce conține toate datele folosite de aplicație de obicei invizibil pentru utilizator în afara vizualizării prin care acesta interacționează. Pentru a interacționa cu celelalte componente ale aplicației modele furnizează interfață de acces la datele stocate. Modelele de asemenea pot varia de la o simpla variabilă de tip intreg (int) pană la o clasă, cu toate beneficiile POO de la încapsularea datelor pentru protecția lor și posibilitatea de reutilizare a codului până la abstractizare, ce permite modelelor să interacționeze cu alte componente fără să știe de prezența acestora. În alte cazuri ca modele sunt și serviciu complex cu mai multe nivele de arhitectura folosite pentru rezolvarea diverselor probleme sau implementarea diferitor funcționalități [6]. Cel mai frecvent pentru realiza legătura dintre date stocate și modele, care furnizează datele aplicației, sunt folosite ORM sau Object Relational Mapper, de exemplu EntityFramework pentru C#.

**2.1.3 Vizualizările**

Vizualizarea reprezintă prezentarea grafică a aplicației pentru utilizator, cu care acesta interacționează. Acestea permite acces la date prin interfață și interacționarea cu aceste date și însuși aplicația, dacă logica programului furnizează aceste opțiuni, prin diferite acțiuni ca click pe mouse sau tastatură. Vizualizările sunt de obicei realizate cu ajutorul limbajelor de markdown sau a variațiilor acestora. Ca exemplu banal este html. Chiar dacă vizualizarea reprezintă datele sau acțiunile efectuate pe aceasta, ea nu știe de logica efectuarii acestor acțiuni. View-urile sunt de asemenea imbricate de obicei având mai multe subview-uri in acestea[6].

**2.1.4 Controlere**

Controalele sunt fac de obicei legatura între vizualizări și modele și definește logica de transmitere a datelor dela una la alta într-un model MVC. Controlerul este responsabil de acceptarea input-ului utilizatorului de la vizualizare și executarea logicii programului conform datelor primite. Acest comportament implementează pattern-ul “Strategy” ce permite flexibilitatea și poate fi utilizat pentru a oferi un comportament editabil și unul needitabil în același formular.[6]

**2.2 MVVM(Model-View-ViewModel)**

În 2004, Marti Fowler (inginer softaware englez) a publicat un articol în care el a prezentat un pattern numit Presentation Model (PM). PM ca și MVP(Model-View-Presenter) separa vizualizarea de comportamentul și starea ei prin abstractizarea acesteia numită Presentation Model sau model de prezentare. Ca consecință vizualizarea devine o interpretare a modelului de prezentare [12].

Deja în 2005 John Gossman, unul dintre arhitecții framework-urilor WPF (Windows Presentation Foundation) și Silverlight ai companiei Microsoft, a prezentat pattern-ul MVVM sau Model-View-ViewModel în blog-ul său [12]. Ca și PM, MVVM avea la bază abstractizarea vizualizării în care este conținută starea și comportamentul. Cu toate acestea MVVM a fost introdus ca o standardizare a modelului pentru a simplifica crearea interfețelor de utilizator prezentă în WPF, în timp ce PM se axa mai mult pe crearea acestor interfețe interfețelor.

**2.2.1 Arhitectura MVVM**

Arhitectura MVVM este destinată mai mult pentru crearea aplicațiilor cu interfețelor de utilizator moderne în care de vizualizare este mai responsabil un designer decât un dezvoltator clasic.[5]

Model-View-ViewModel este un pattern care include trei componente esențiale: Model, View și ViewModel, așa cum sugerează denumirea modelului. Ca și în cazul la MVC, vizualizarea și modelul au acela rol de a afișa datele sau interacționa cu utilizatorul și de a face legătura și validarea acestor date respectiv. În cazul la ViewModel sau model de vizualizare, acesta are ca menire să descrie logica, comportamentul și starea vizualizării, în afară de transmiterea datelor și prelucrarea lor [10]. O simplă reprezentare a interacțiunii între componente este figura 2.2.

Actualizarea datelor

Acces la date

Chemarea unei acțiuni

Transmite rezultatul

figura 2.2 Reprezentarea interacțiunii între componentele MVVM

**2.2.2 Modelele**

Ca în cazul MVC-ului modelul are același rol de a face legătura cu datele din mediul de stocare și de a le furniza pentru afișare sau salva schimbările acestora. Acestea sunt de obicei clase model care încapsulează datele. Exemple de modele includ repozitorii, business objects, obiecte de transfer de date (DTO), obiecte CLR simple (POCO) și obiecte entitate și proxy generate.

**2.2.2 Vizualizările**

Vizualizările sunt responsabile de definirea structurii, amplasării și aspectului elementelor care apar la ecran. În cele mai dese cazuri vizualizările sunt definite în XAML sau XML. În caz ideal vizualizările nu conțin în ele nici o logică de execuție ci doar referințe la funcții definite în model de vizualizare. O vizualizare poate avea propriul model de vizualizare sau poate moșteni modelul de vizualizare al părintelui său. O vizualizare primește date de la modelul său de vizualizare prin intermediul legăturilor sau al invocării metodelor din modelul de vizualizare[11]. În timpul execuției, vizualizarea se modifică atunci când controalele interfeței de utilizator primesc răspuns de la modelului de vizualizare prin execuția evenimentului legat de componenta vizualizării [11]. Din acțiuni care pot chema execuția unei metode pot fi de exemplu click, dublu click, etc.

**2.2.3 Modele de Vizualizări**

În cele mai dese cazuri modelele de vizualizare sunt descrise ca clase care fac legătura între model și vizualizare și sunt responsabile pentru definirea logicii de comportament anume a vizualizărilor. Cu alte cuvinte se poate de spus că ViewModel-ul interacționează cu modelul aplicației prin chemarea metodelor și extragerea datelor din clasa model și cu vizualizarea prin furnizarea acestor date și implementarea metodelor în formă de evenimente (events) ce descriu logica interacțiunii utilizatorului cu aplicația prin aceasta. Accesul modelului de vizualizare la starea vizualizării este efectuată prin “data-binding” pentru a se sincroniza, ca să primească acces la elementele vizualizării și proprietăților acestora [11].

**2.3 MVP(Model-View-Presenter)**

Model-View-Presenter (MVP) a fost introdus pentru prima dată în sistemul de operare Taligent, companie formată din colaborarea Apple, IBM, și Motorola în 1996 [13]. Conceptul pattern-ului se bazează pe MVC cu definire mai clară a interacțiunii între componente. MVVM menționat mai devreme are ca predecesor MVP [5].

**2.3.1 Arhitectura MVP**

MVP a fost conceput pe baza a două întrebări: gestionarea datelor și interfața utilizator. Adică, cum să fie gestionate datele și cum să fie implementată interacționarea utilizatorilor cu datele[5]. Acest pattern de design a fost creat pentru a ușura dezvoltarea interfețelor grafice ca consecință a separării funcțiilor între vizualizare și componenta care implementează logica vizualizării. În formă simplificată modelul MVP conține trei componente: model, vizualizare și preseter ca și în cazul celorlalte pattern-e, însă în unele cazuri MVP este divizat în 6 componente: selections, commands, interactor și cele trei menționate mai devreme. În caz general reprezentarea componentelor și interacțiunilor între acestea este în figura 2.3.

Actualizarea vizualizării

Actualizarea datelor

Acces la date

Chemarea unei acțiuni

Figura 2.3 Reprezentarea interacțiunii între componentele MVP

**2.3.2 Modele**

Din cauză că MVP este succesorul MVC și predecesorul MVVM, ca în celelalte cazuri modelele au aceeași structură și funcție în cadrul aplicației.

**2.3.2 Vizualizările**

Vizualizările iarăși au aceeași funcție ca și în MVC și MVVM cu excepția faptului că acestea nu au acces direct la model, ci comunică cu modelul prin prezentatoare evidențiind prin aceasta diferența față de predecesorul MVC. În cazul MVP vizualizările sunt definite ca interfețe care sunt implementate de preseter cu componentele acesteia fiind definite în limbajelor de markup ceea ce permite preseter-ului să manipuleze cu vizualizarea direct [14].

**2.3.3 Prezentatoare**

Prezentatorul este o dezvoltare a controlerului din Model-View-Controller și are ca menire implementarea logicii funcționării aplicației prin efectuarea legăturii între model și vizualizare. Spre deosebire de echivalentul său în MVC, prezentatorul decide interacțiunea cu vizualizarea și are acces complet la aceasta și elementele acesteia cât și la valorile acestora [4].

**2.4 Compararea șabloanelor de arhitectură**

Șabloanele arhitecturale MVC, MVP și MVVM, deși pornesc de la aceleași principii de separare a responsabilităților și organizare clară a aplicației, se diferențiază semnificativ în ceea ce privește rolul componentelor și fluxul de date între ele. Toate cele trei modele definesc o structură clară între logică, interfață și manipularea datelor, însă modul în care acestea interacționează diferă esențial.

În cazul MVC, controlerul acționează ca un intermediar între model și vizualizare, dar nu controlează complet afișarea datelor. Vizualizarea poate accesa direct modelul pentru a obține datele necesare, ceea ce poate duce la o dependență mai mare între componente. Controlerul răspunde la acțiunile utilizatorului și actualizează modelul, iar modificările din model se pot reflecta direct în vizualizare. Acest model este simplu de implementat și potrivit pentru aplicații unde logica de interfață este redusă, însă devine greu de menținut pe măsură ce aplicația crește în complexitate [6][7][8].

MVP aduce o îmbunătățire importantă față de MVC prin introducerea unui presenter care controlează complet logica interfeței. Vizualizarea comunică doar cu presenterul și nu interacționează direct cu modelul. Astfel, MVP oferă o separare mai clară a responsabilităților și permite testarea ușoară a logicii aplicației, fiind potrivit pentru aplicații unde logica de prezentare este mai complexă. Totuși, interdependența puternică dintre presenter și view poate duce la o creștere a codului boilerplate și a complexității generale [4][5].

MVVM, pe de altă parte, a fost conceput special pentru framework-urile care suportă data-binding bidirecțional, cum este WPF. Acest model introduce componenta ViewModel, care acționează ca o punte între model și vizualizare, menținând logica de prezentare și starea UI-ului. Vizualizarea este legată direct de ViewModel prin data-binding, ceea ce reduce semnificativ codul necesar pentru actualizarea interfeței. MVVM este ideal pentru aplicații moderne, în special cele desktop sau mobile, unde separarea clară între interfață și logică este crucială, dar poate deveni complex de implementat în lipsa suportului adecvat pentru binding [9][10][11].

În concluzie, alegerea între MVC, MVP și MVVM depinde de cerințele specifice ale proiectului, complexitatea interfeței utilizator, nivelul dorit de testabilitate și tehnologiile utilizate. MVC este simplu și potrivit pentru aplicații mici sau medii, MVP oferă un control mai bun asupra logicii de prezentare, iar MVVM este optim pentru aplicații cu interfețe complexe și binding suportat nativ.

**Capitolul 3. Un exemplu de implementare a MVC**

**3.1 Formularea problemei. Cerințe funcționale și nonfuncționale către aplicație**

În contextul actual, sănătatea unei persoane este un factor esențial pentru calitatea vieții și pentru bunăstarea întregii societăți. În Republica Moldova, procesul de digitalizare a sistemului medical este insuficient dezvoltat, ceea ce duce la dificultăți semnificative în accesarea serviciilor medicale. De cele mai multe ori, pacienții trebuie să se deplaseze fizic la instituțiile medicale pentru a se programa la un medic, iar acest lucru implică adesea timp pierdut și stat la cozi.

Acest context evidențiază necesitatea unei soluții informatice care să permită programarea online la consultații medicale. O astfel de aplicație ar contribui la reducerea timpului de așteptare, ar optimiza interacțiunea pacient–medic și ar oferi pacienților posibilitatea de a alege medicul în funcție de recenzii, specializare sau disponibilitate.

Un alt aspect important îl reprezintă accesibilitatea aplicației. Având în vedere diversitatea lingvistică a populației, este esențial ca aceasta să fie disponibilă în mai multe limbi, astfel încât utilizatorii să o poată folosi chiar dacă, de exemplu, nu cunosc limba română, dar vorbesc engleza – sau invers.

**Obiectivul principal al aplicației:**

Crearea unei aplicații web folosing arhitectura MVC care să permită programarea online la medic, prin completarea unui formular cu date personale (nume, prenume, date de contact), alegerea zilei și orei consultației, în funcție de intervalele disponibile în programul medicului.

**Cerințe funcționale:**

* Utilizatorii pot crea un cont și se pot autentifica în aplicație.
* Utilizatorii pot vizualiza lista medicilor disponibili și pot face o programare în baza orelor libere.
* Medicul poate accepta sau refuza o programare.
* Utilizatorul este notificat prin e-mail cu privire la starea programării (acceptată/refuzată).
* Aplicația este disponibilă în cel puțin două limbi (română/engleză).

**Cerințe nefuncționale:**

* Cel puțin 4 roluri: Guest, User, Doctor, Administrator.
* Interfața aplicației trebuie să fie intuitivă și ușor de utilizat.
* Sistemul trebuie să fie scalabil, pentru a susține un număr mare de utilizatori.
* Aplicația trebuie să fie securizată, în special în ceea ce privește stocarea și transmiterea datelor personale.
* Timpul de răspuns al aplicației trebuie să fie redus, pentru a asigura o experiență de utilizare optimă.

**3.2 Descrierea proiectului sistemului informatic**

Proiectul propus reprezintă o aplicație web dezvoltată în arhitectura **ASP.NET Core MVC** [15][16], care permite utilizatorilor să se programeze online la consultații medicale, să selecteze limba de afișare a interfeței și să primească notificări prin e-mail. Aplicația este gândită să fie ușor de utilizat, rapidă și accesibilă.

Arhitectura MVC (Model–View–Controller) separă responsabilitățile aplicației în trei componente majore:

* **Modelul** gestionează logica de acces la date și reprezintă entitățile persistente;
* **View-ul** este responsabil de interfața grafică și interacțiunea cu utilizatorul;
* **Controllerul** acționează ca intermediar între Model și View, preluând cererile utilizatorului și oferind răspunsul adecvat.

**3.2.1 Arhitectura generală a aplicației**

Mai jos este reprezentată diagrama care ilustrează interacțiunea dintre componente:

Servicii

Utilizator

Model (ex: Account, Appointment, etc.)

View (Razor Page, ex: Index.cshtml, etc.)

Controller (ex: HomeController, AccountController, etc)

Bază de date (SQL Server)

**3.2.2 Structura generală a proiectului**

Figura 3.1 Reprezentarea interacțiunii între componentele aplicației.

Aplicația are o structură clar definită:

* wwwroot: fișiere statice (CSS, JS, imagini)
* Models: entitățile EF Core și DTO-urile pentru schimbul de date
* Controllers: logica de control
* Views: interfața cu utilizatorul
* Services: servicii pentru trimiterea e-mailurilor și localizare
* Classes: funcționalități suplimentare (ex: conversii de imagine)

**3.2.3 Componentele aplicației**

**Model**

Modelul este format dintr-un set de clase care definesc structura bazei de date și logica de lucru cu datele. Clasele esențiale sunt:

* Account: reprezintă contul utilizatorului
* Doctor, DoctorSpecialitie: definesc entitățile medicale
* Appointment: reprezintă o programare la medic
* Culture, Resources: utilizate pentru localizarea interfeței
* LogEvent: pentru sistemul de logare a activităților

Persistența datelor este realizată cu ajutorul **Entity Framework Core** [17], utilizând abordarea **Code First**. Accesul la date se face prin interogări LINQ [18], în cadrul contextului DoctorAppDbContext, derivat din DbContext al EF Core.

**View**

Partea de interfață este construită folosind **Razor Pages**, integrate în cadrul ASP.NET Core MVC. Interfața conține:

* o pagină principală cu informații despre aplicație, o listă cu medici disponibili, un formular de programare;
* o pagină pentru logare și inregistrare;
* o pagină de profil al doctorului
* o pagină pentru lista de programări la doctor(în cazul în care utilizatorul are rol de doctor)
* o bară de navigare care conține și un selector de limbă (combobox) care permite comutarea între mai multe limbi prin sistemul de localizare implementat;
* O pagină pentru administrare

Toate vizualizările sunt organizate în folderul Views, separate pe controllere, iar layout-ul general se află în folderul Shared.

**Controller**

Controller-ele definesc logica de manipulare a cererilor utilizatorului. Exemple de controllere:

* AccountController: gestiunea conturilor
* ConfirmationController: confirmarea programărilor
* HomeController: vizualizarea paginii principale, profilelor de doctor, efectuarea înregistrării la doctor, etc
* LogsController, DoctorController, DoctorSpecialitiesController, StatisticsController, LocalizationController și UsersController: administrarea resurselor și informației

Metodele din controllere returnează IActionResult și interacționează cu modelele și serviciile (ex: trimiterea unui e-mail după o programare reușită).

**3.2.4 Tehnologii utilizate**

* **Backend**: ASP.NET Core MVC, EF Core (cu SQL Server), MailKit [19] (pentru trimiterea de e-mailuri)
* **Frontend:** Razor Views cu HTML/CSS/JS, Bootstrap (pentru stilizare și interfață responsive)
* **Persistență**: EF Core + SQL Server
* **Localizare**: sistem propriu de localizare prin clasele StrLocalizer și StrLocalizerFactory
* **Securitate**: sistem de autentificare și autorizare propriu
* **Logare**: sistem de jurnalizare a evenimentelor prin LogEvent

**3.3 Particularitățile dezvoltării sistemului informatic**

Dezvoltarea sistemului informatic a presupus integrarea mai multor componente tehnologice și logice, menite să răspundă atât cerințelor de funcționalitate, cât și celor de securitate, performanță și accesibilitate. În cele ce urmează sunt prezentate principalele particularități care au influențat procesul de dezvoltare și arhitectura aplicației.

Proiectul urmează o structură organizată conform arhitecturii MVC, cu separarea clară a responsabilităților în foldere precum Models, Views, Controllers, Services și Classes. Această organizare facilitează mentenanța aplicației, testarea componentelor și extinderea funcționalităților pe viitor.

**3.3.1 Utilizarea arhitecturii ASP.NET Core MVC**

Arhitectura ASP.NET Core MVC a fost aleasă datorită capacității sale de a separa în mod clar responsabilitățile între componenta de interfață (View), logica de control (Controller) și modelul de date (Model). Această separare permite o dezvoltare modulară, testabilă și ușor de întreținut.

Modelul gestionează entitățile aplicației și interacțiunea cu baza de date prin intermediul Entity Framework Core, folosind abordarea Code First. Controller-ele intermediază cererile utilizatorului și transmit rezultatele corespunzătoare către View-uri, care sunt responsabile de afișarea datelor către utilizator.

**3.3.2 Implementarea unui sistem propriu de autentificare și autorizare**

O componentă esențială a aplicației este sistemul de autentificare și autorizare, dezvoltat fără utilizarea template-urilor implicite din ASP.NET Identity. Acesta permite:

* Crearea conturilor de utilizator cu roluri distincte (User, Doctor, Administrator);
* Autentificarea pe baza datelor introduse în formularul de login, cu criptarea parolelor;
* Verificarea contului printr-un cod de confirmare trimis pe e-mail;
* Gestionarea sesiunii utilizatorului prin cookie-uri și validarea autentificării pe parcursul navigării;
* Protejarea accesului la resurse în funcție de rol.

**3.3.3 Interfața aplicației și utilizarea Bootstrap**

Interfața grafică a aplicației a fost dezvoltată utilizând **Razor Pages** pentru integrarea nativă în ASP.NET Core MVC, în combinație cu HTML, CSS și **framework-ul Bootstrap**. Folosirea Bootstrap a oferit următoarele avantaje:

* Un design responsive, compatibil cu diferite rezoluții și dispozitive;
* Componente vizuale reutilizabile (formulare, butoane, navigație, modale etc.);
* O interfață coerentă și modernă, care contribuie la experiența utilizatorului.

Navigarea este intuitivă, cu o bară de meniu fixă, care conține și un selector de limbă. Fiecare pagină este clar structurată pentru a asigura accesul rapid la funcționalitățile esențiale: programări, profil doctor, listă programări, administrare etc.

**3.3.4 Localizarea aplicației în mai multe limbi**

Pentru a asigura accesibilitatea aplicației unui public larg, s-a implementat un sistem propriu de localizare, fără a folosi fișiere .resx. Acesta se bazează pe o structură de date personalizată salvată în baza de date SQL Server, care conține tabelele Culture și Resource. Fiecare resursă textuală are un key și o value specifică limbii asociate (ro, en etc.).

Sistemul de localizare este implementat prin clasele StrLocalizer și StrLocalizerFactory, care respectă interfețele IStringLocalizer și IStringLocalizerFactory din cadrul ASP.NET Core. La pornirea aplicației, dacă baza de date nu conține încă resurse de localizare, se adaugă automat valori implicite pentru cele două limbi suportate. În paginile Razor, localizarea este integrată prin directiva @inject IViewLocalizer Localizer, care permite afișarea dinamică a textelor în funcție de cultura curentă a utilizatorului.

Această abordare oferă flexibilitate, permițând adăugarea și modificarea textelor direct din baza de date, fără a recompila aplicația.

**3.3.5 Trimiterea de notificări prin e-mail**

Pentru implementarea mecanismului de confirmare a contului și îmbunătățirea comunicării cu utilizatorii, aplicația include un **sistem de notificări prin e-mail**, prin care este realizată confirmarea (ca exemplu a contului) și care informează pacientul dacă programarea a fost realizată și dacă aceasta a fost acceptată, refuzată, etc. Acesta este implementat cu ajutorul bibliotecii **MailKit**, care oferă control complet asupra autentificării SMTP, formatului mesajului (HTML sau text simplu) și trimiterea asincronă.

Serviciul de trimitere a e-mailurilor este integrat în backend și apelat din controller-ele în care sunt apelate metodele acestora în funcție de acțiunile efectuate de pacient/medic.

**3.3.6 Logarea evenimentelor în sistem**

Pentru a asigura o transparență și o posibilitate de audit, aplicația include un sistem propriu de **logare a evenimentelor**, prin clasa LogEvent. Acesta înregistrează acțiuni importante precum:

* Autentificări reușite sau eșuate;
* Modificări de cont sau programări;
* Accesarea paginilor;

Etc.

Evenimentele sunt înregistrate în baza de date și pot fi vizualizate de administrator într-o pagină dedicată, contribuind la securitate și controlul asupra funcționării aplicației.

**3.3.7 Accesul la date și persistența**

Pentru gestionarea datelor persistente, aplicația folosește **Entity Framework Core**, configurat cu **SQL Server** ca sistem de baze de date. Accesul la date se face prin:

* Abordarea Code First, care permite modelarea bazei de date direct din clase C#;
* Utilizarea de interogări LINQ pentru manipularea entităților;
* Controlul tranzacțiilor și salvarea modificărilor prin DbContext.

Această metodă oferă flexibilitate în modificarea structurii bazei de date și permite un control mai bun asupra procesului de dezvoltare.

**3.4 Testarea Aplicației**

Aplicația web a fost testată pentru a verifica funcționalitatea principalelor componente, precum și pentru a evalua experiența utilizatorului în diferite scenarii. În cadrul aplicației, toate paginile dispun de un **selector de limbă (combobox)**, prin intermediul căruia utilizatorul poate schimba limba de afișare a interfeței (vezi Figura 3.2).

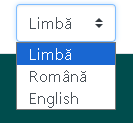


Figura 3.2. Selectorul de limbă pentru localizarea interfeței

**4.1 Testarea procesului de autentificare și înregistrare**

Aplicația oferă o pagină de **autentificare** și una de **înregistrare** (vezi Anexa 1 și Anexa 2). Utilizatorul are posibilitatea de a se înregistra fie ca **utilizator obișnuit**, fie ca **doctor**. În cazul înregistrării ca doctor, sistemul solicită completarea unor informații suplimentare, care sunt ulterior verificate pentru validitate.

Pentru înregistrarea unui cont de tip doctor, utilizatorul trebuie să selecteze opțiunea corespunzătoare din meniul tipului de utilizator. După alegerea acestei opțiuni, utilizatorul este redirecționat către o pagină specială, unde introduce datele necesare despre activitatea sa medicală (vezi Anexa 3).

**4.2 Testarea funcționalităților de bază**

După autentificare, utilizatorul este direcționat către pagina principală a aplicației (vezi Anexa 4). Aceasta oferă următoarele funcționalități:

* posibilitatea de a schimba limba interfeței;
* vizualizarea serviciilor medicale disponibile (vezi Anexa 5);
* accesarea listei de doctori;
* consultarea detaliilor unui medic prin accesarea profilului său (vezi Anexa 6 și Anexa 7);
* completarea unui formular de programare (vezi Anexa 8).

Formularul de programare permite alegerea medicului dorit, introducerea numelui, prenumelui, datei și orei dorite pentru consultație. După trimiterea formularului, programarea este salvată în baza de date.

**4.3 Testarea funcționalităților pentru doctori**

Utilizatorii înregistrați cu rol de **doctor** pot accesa pagina „Profilul meu”, care conține secțiunea de gestionare a programărilor (vezi Anexa 9 – Anexa 11). Aici, fiecare doctor poate vizualiza programările primite și are opțiunea de a le **accepta** sau **refuza**. Lista afișată este filtrată automat astfel încât să includă doar programările viitoare, afișate în ordine cronologică ascendentă.

**4.4 Testarea funcționalităților pentru utilizatorii obișnuiți**

Utilizatorii obișnuiți pot accesa secțiunea proprie cu programările efectuate, unde pot consulta statusul acestora (confirmată, refuzată sau în așteptare). Aceste programări pot fi vizualizate în paginile prezentate în Anexa 11 – Anexa 13.

**Concluzie**

În concluzie, această lucrare a evidențiat relevanța continuă a modelelor arhitecturale clasice – MVC, MVP și MVVM – în dezvoltarea aplicațiilor web. Aceste pattern-uri oferă un echilibru eficient între claritate structurală, flexibilitate și ușurință în mentenanță. Analiza comparativă a arătat că MVC, prin separarea logicii de prezentare de cea de business, este potrivit pentru aplicații de complexitate redusă și medie, permițând prototiparea rapidă și extinderea controlată a funcționalităților. MVP se dovedește eficient în aplicații cu interfețe complexe, unde controlul logicii este centralizat în componenta *presenter*, facilitând testarea și reducerea cuplajului. MVVM, deși conceput inițial pentru medii cu data-binding nativ, aduce beneficii și în dezvoltarea web prin reducerea codului redundant și o mai bună separare a responsabilităților.

Studiul de caz privind o aplicație pentru gestionarea programărilor medicale a demonstrat aplicabilitatea practică a arhitecturii MVC în asigurarea modularității, scalabilității și întreținerii eficiente. Implementarea funcționalităților precum autentificarea, localizarea și notificările a arătat că organizarea stratificată a codului permite integrarea facilă a componentelor și menținerea unui grad ridicat de testare automată. Evaluările de performanță au confirmat că o arhitectură clar definită contribuie la timpi de răspuns stabili și costuri reduse de mentenanță pe termen lung.

Pentru cercetări viitoare, se recomandă investigarea arhitecturilor hibride, care combină avantajele modelelor clasice cu principii moderne, cum ar fi integrarea micro-frontendurilor în cadrul MVC sau utilizarea MVVM în aplicații cu suport extins pentru data-binding. De asemenea, extinderea studiilor de caz la proiecte de mari dimensiuni ar permite o evaluare mai precisă a impactului asupra scalabilității și a costurilor operaționale. În ansamblu, lucrarea susține că o înțelegere solidă a arhitecturilor clasice este esențială pentru fundamentarea deciziilor tehnice și pentru construirea unor soluții web robuste și adaptabile.

**Bibiliografie:**

1. [Humberto Cervantes, and Rick Kazman. *Designing Software Architectures: A Practical Approach*. Addison-Wesley, 2016/2024.]

2. [Eric Freeman, and Elisabeth Robson. *Head First Design Patterns*. O'Reilly Media, 2004.]

3. [Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.]

4. [iamprovidence, MVC vs MVP vs MVVM with C# examples, https://medium.com, 2022](<https://medium.com/@iamprovidence/mvc-vs-mvp-vs-mvvm-with-c-examples-8013745e3c4c>)

5. [Tian Lou, A Comparison of Android Native App Architecture – MVC, MVP and MVVM, Master’s Thesis, Espoo, September 06, 2016](<https://pure.tue.nl/ws/portalfiles/portal/48628529/Lou_2016.pdf>)

6. [[Model-View-Controller architectural pattern and its evolution in graphical user interface frameworks](https://lutpub.lut.fi/handle/10024/92156)][https://lutpub.lut.fi/bitstream/handle/10024/92156/Model-View-Model%20architectural%20pattern%20and%20its%20evolution%20in%20graphical%20user%20interface%20frameworks.pdf?sequence=2&isAllowed=y](https://lutpub.lut.fi/bitstream/handle/10024/92156/Model-View-Controller%20architectural%20pattern%20and%20its%20evolution%20in%20graphical%20user%20interface%20frameworks.pdf?sequence=2&isAllowed=y)

7. [The original MVC reports Trygve Reenskaug Dept. of Informatics University of Oslo](<https://folk.universitetetioslo.no/trygver/2007/MVC_Originals.pdf>)

8. [Architectural Design of Modern Web Applications](<https://www.researchgate.net/profile/Lech-Madeyski/publication/221679095_Architectural_Design_of_Modern_Web_Applications/links/5c8fdf3292851c1df94a5233/Architectural-Design-of-Modern-Web-Applications.pdf>)

9. [MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern] (<https://conferences.big.tuwien.ac.at/biweek2024/pdfs/biweek2024_paper_87.pdf> )

10. [The MVVM Pattern] (<https://conferences.big.tuwien.ac.at/biweek2024/pdfs/biweek2024_paper_87.pdf> )

11. [Model-View-ViewModel (MVVM)](<https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm>)

12. [Patterns - WPF Apps With The Model-View-ViewModel Design Pattern] (<https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern> )

13. [MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java] (<https://www.wildcrest.com/Potel/Portfolio/mvp.pdf> )

14. [Model View Presenter](<https://anshul-vyas380.medium.com/model-view-presenter-b7ece803203c>)

15. [ASP.NET documentation] (<https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0>)

16. [Overview of ASP.NET Core MVC](

<https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-9.0#aspnet-core-mvc> )

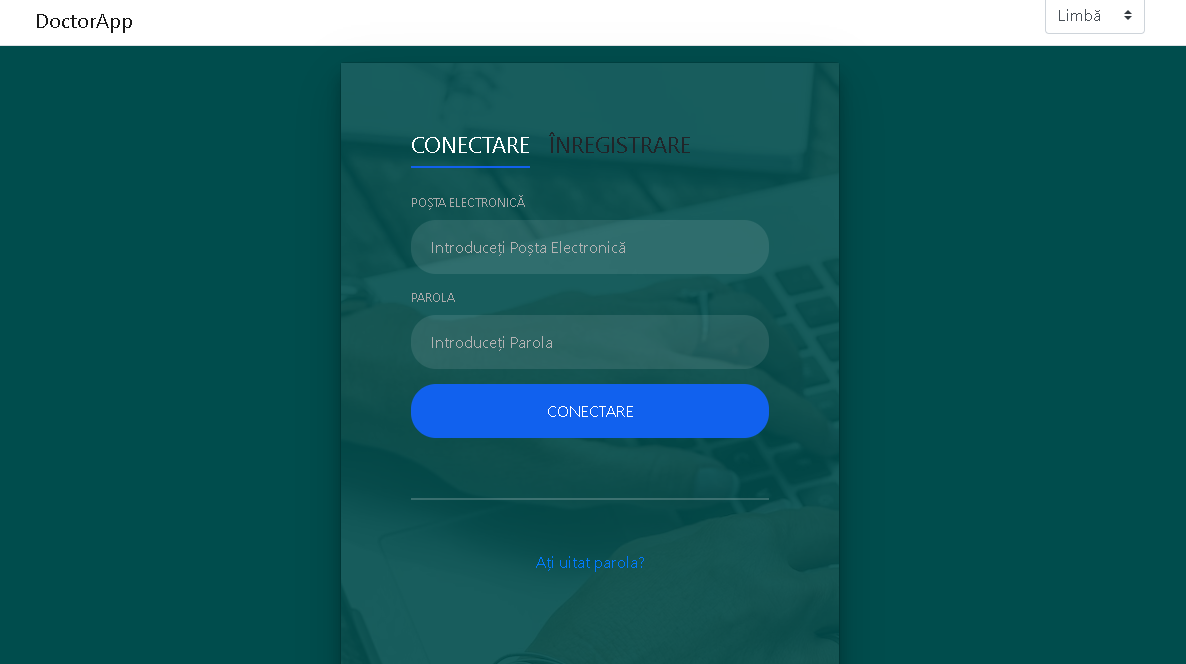
17. [Entity Framework documentation hub](<https://learn.microsoft.com/en-us/ef/> )

18. [Language Integrated Query (LINQ)](<https://learn.microsoft.com/en-us/dotnet/csharp/linq/> )

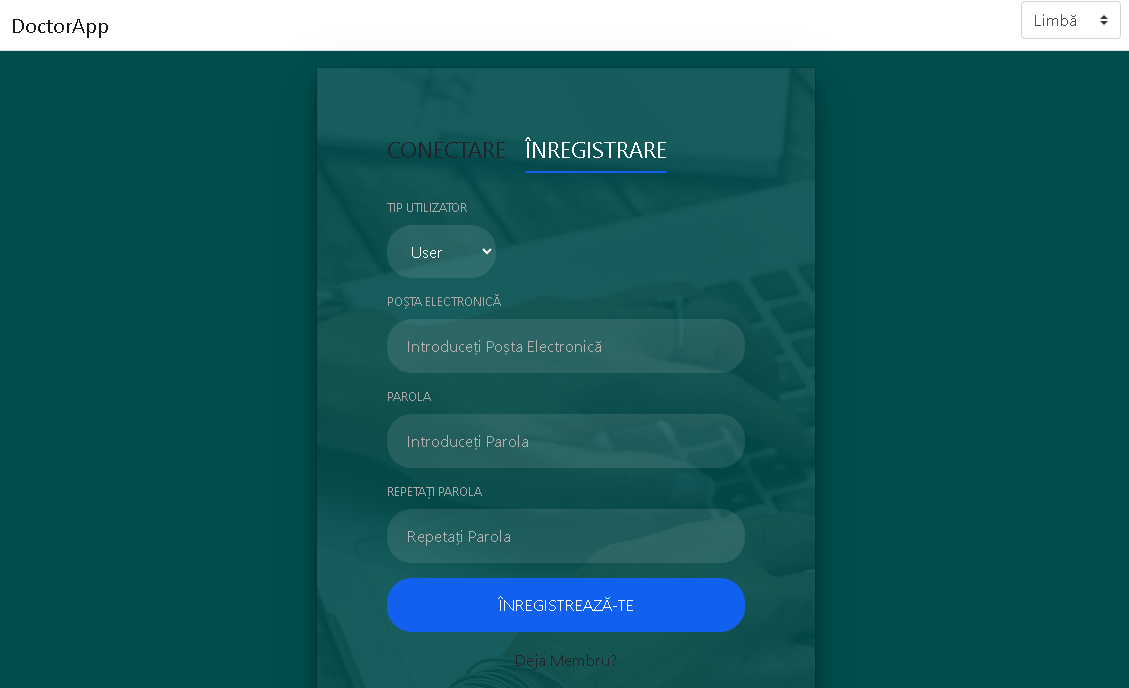
19. [MailKit Documentation]( <https://mimekit.net/docs/html/Introduction.htm> )

**Anexe**

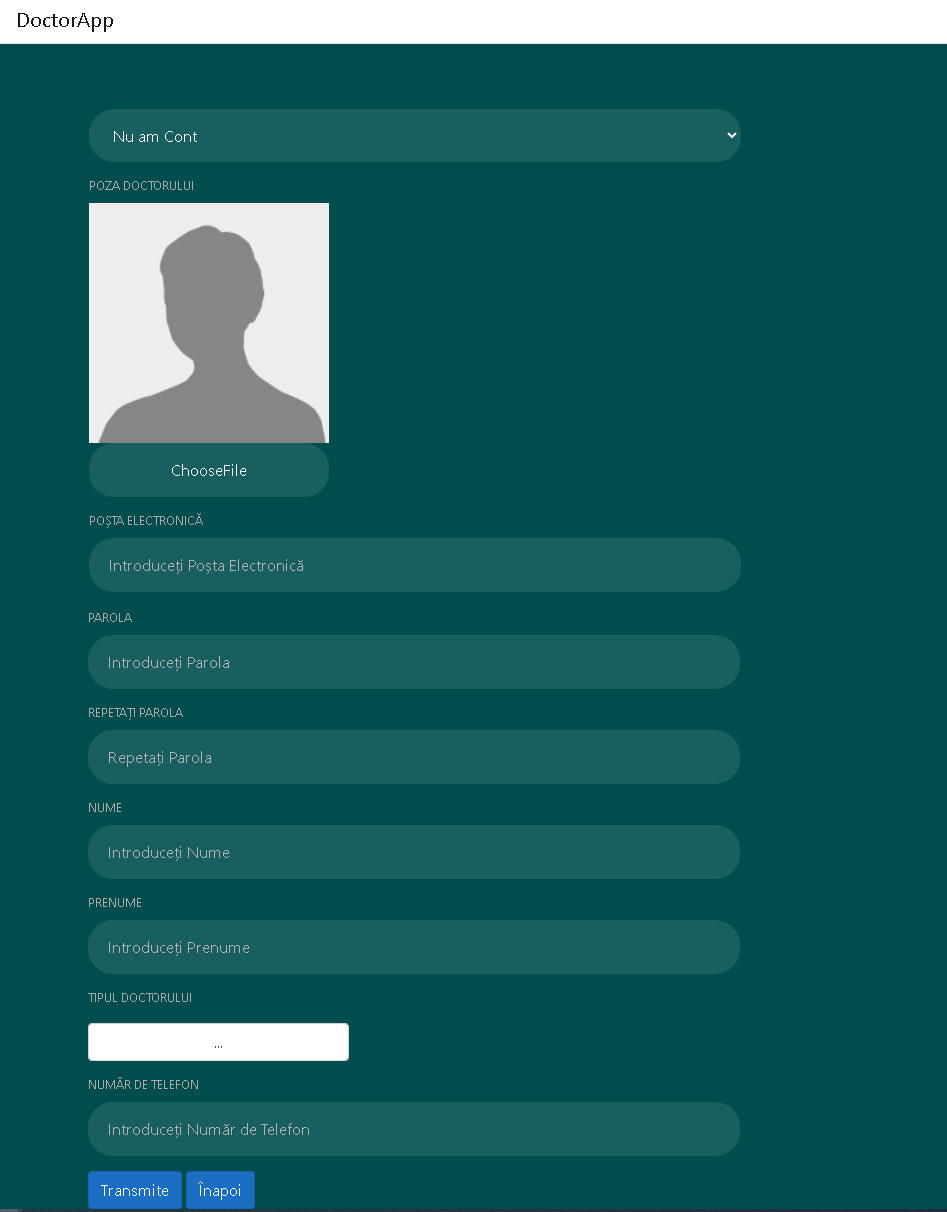
*Anexa 1: Pagina de login*



*Anexa 2: Pagina de înregistrare*



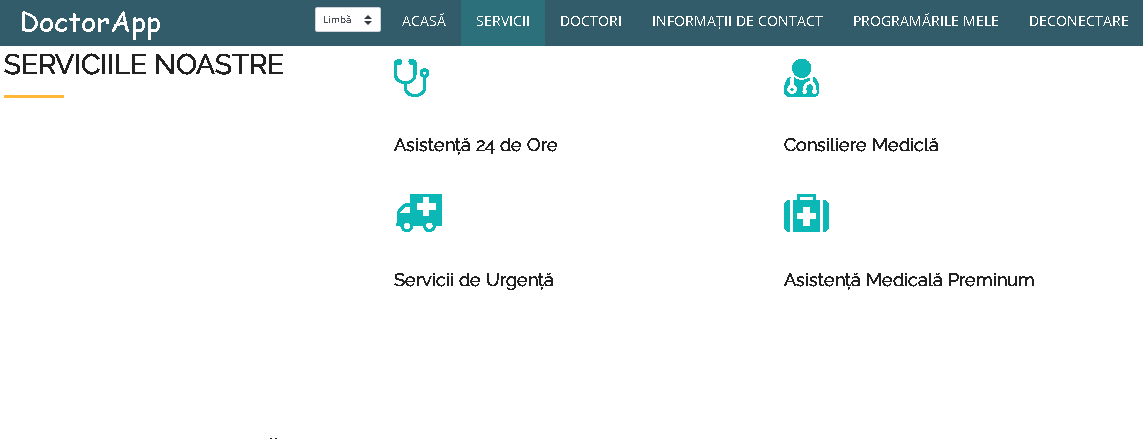
*Anexa 3: Pagina introducere a datelor pentru candidat la rolul de doctor*



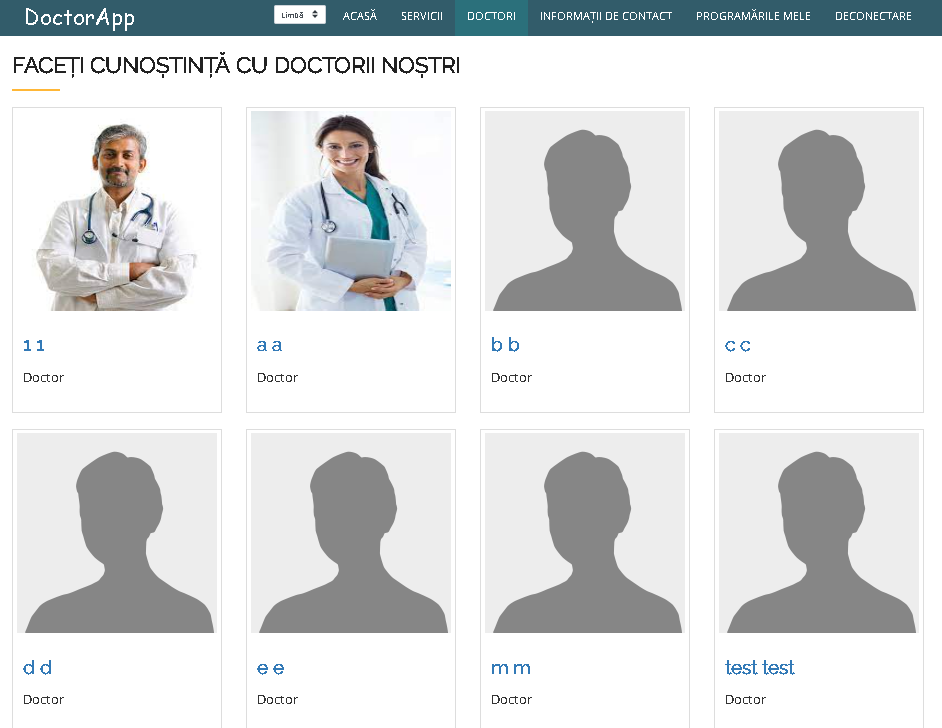
*Anexa 4: Pagina principală*



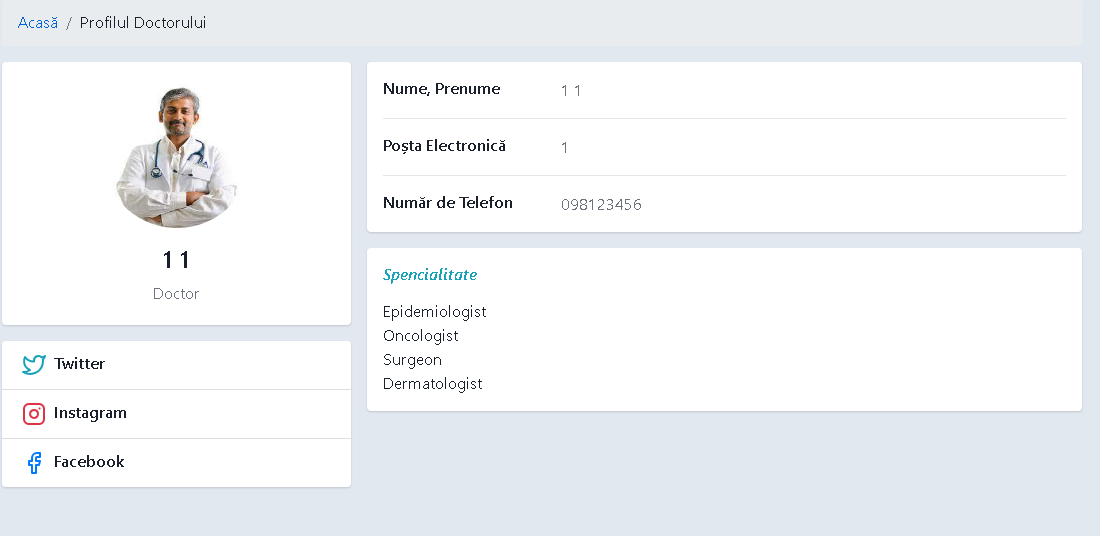
*Anexa 5: Pagina principală (secțiunea cu servicii medicale)*



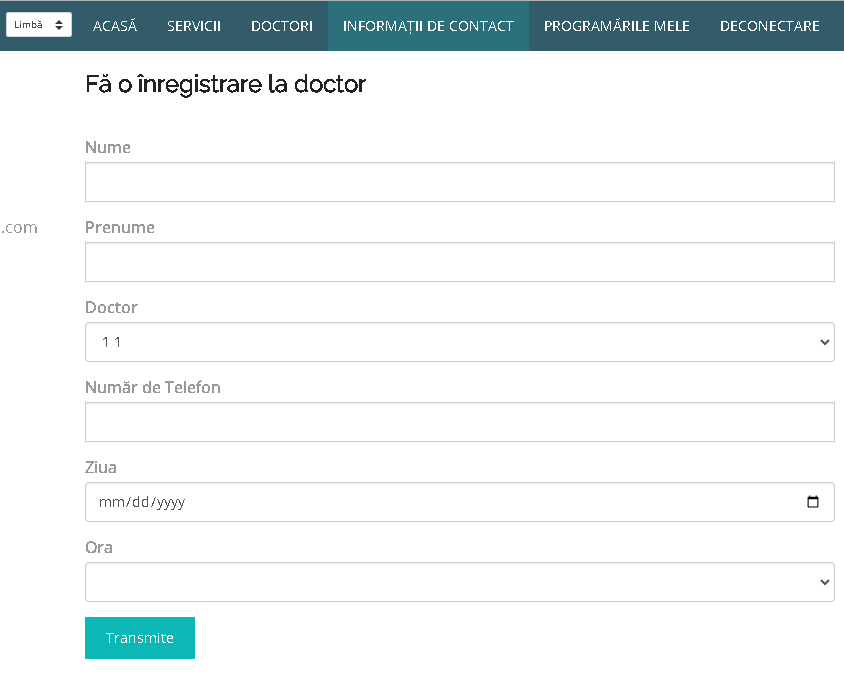
*Anexa 6. Pagina principală (secțiunea cu informație despre doctori)*



*Anexa 7. Pagina cu informație despre un anumit doctor*



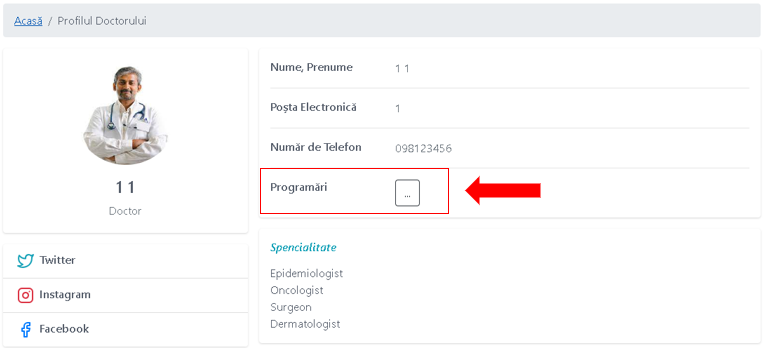
*Anexa 8. Pagina principală (secțiunea în care se poate de făcut înregistrarea la doctor)*



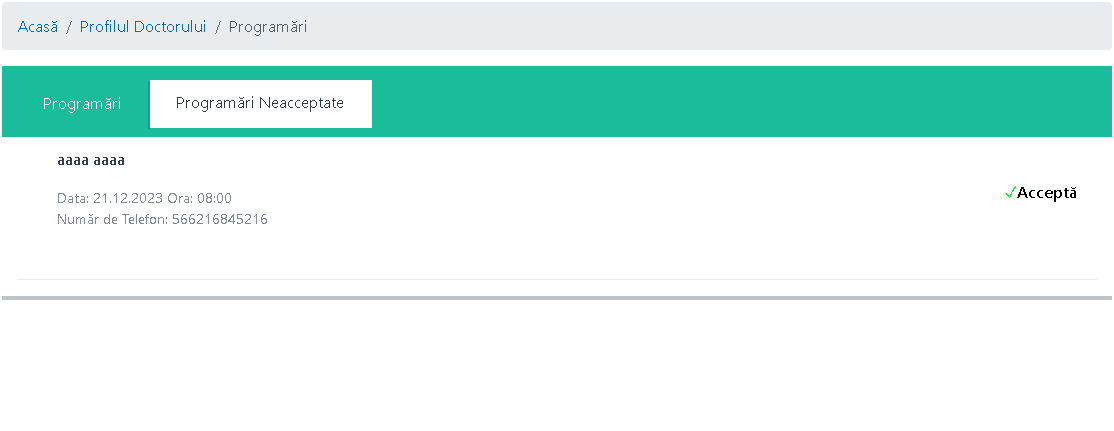
*Anexa 9. Pagina principală din perspectiva unui doctor*



*Anexa 10. Profilul din perspectiva doctorului*



*Anexa 11. Lista de înregistrări care pot fi acceptate*



*Anexa 12. Pagina principală (Programările Mele)*



*Anexa 13. Pagina listei de programări care au fost efctuate de utilizatorul*

