

**Universitatea Tehnică a Moldovei**

**Tehnologia informației în comerț electronic**

**Information tehnology in e-commerce**

|  |  |
| --- | --- |
| **Student:** | **gr. TI-194,**  **Rotaru Dan** |
| **Coordonator:** | **Cebotari Daria**  **lector universitar** |

**Chişinău, 2023**

**MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Admis la susţinere**

**Şef departament:**

**FIODOROV Ion dr., conf.univ.**

**--------------------------------**

**„\_\_\_”**  **2023**

**Tehnologia informației în comerț electronic**

**Proiect de licenţă**

|  |  |  |
| --- | --- | --- |
| **Student:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Rotaru Dan, TI-194** |
| **Coordonator:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Cebotari Daria, lect. univ.** |
| **Consultant:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Cojocaru Svetlana, asist.univ.** |

**Chişinău, 2023**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Programul de studii Tehnologia informaţiei**

**Aprob**

**Șef departament:**

**Fiodorov Ion, dr., conf.univ.**

**„ *06” octombrie* *2022***

**CAIET DE SARCINI**

**pentru proiectul de licenţă al studentului**

*Rotaru Dan*

*(numele şi prenumele studentului)*

**1. Tema proiectului de licenţă**  *Tehnologia informației în comerț electronic*

**confirmată prin hotărârea Consiliului facultăţii nr.** *2* **din**  „ *06*” *octombrie*  *2022*

**2. Termenul limită de prezentare a proiectului** **de licență** *20.05.2023*

**3. Date iniţiale pentru elaborarea proiectului de licență** *Sarcina pentru elaborarea proiectului de diplomă.*

**4. Conţinutul memoriului explicativ**

*Introducere*

*1* *Analiza domeniului de studiu*

*2 Modelarea și proiectarea sistemului*

*3 Realizarea sistemului*

*4 Documentarea produsului realizat*

*5 Estimarea costurilor proiectului*

*Concluzii*

**5. Conţinutul părţii grafice a proiectului de licență**

*Imaginea generală a sistemului, Înterfața de bază a sistemului, …….*  **6. Lista consultanţilor:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Consultant** | **Capitol** | **Confirmarea realizării activităţii** | |
| **Semnătura consultantului (data)** | **Semnătura studentului (data)** |
| *Cojocaru Svetlana* | *Standarde tehnologice, Controlul calităţii, Estimarea costului proiectului* |  |  |

**7. Data înmânării caietului de sarcini** *02.09.2023*

**Coordonator** *Cebotari Daria* **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

*semnătura*

**Sarcina a fost luată pentru a fi executată de către studentul** Rotaru Dan

*02.09.2023*

*semnătura, data*

**PLAN CALENDARISTIC**

|  |  |  |  |
| --- | --- | --- | --- |
| **Nr. crt.** | **Denumirea etapelor de proiectare** | **Termenul de realizare a etapelor** | **Nota** |
| *1* | *Elaborarea sarcinii, primirea datelor pentru sarcină* | *02.09.22– 30.09.22* | *10%* |
| *2* | *Analiza domeniului de studiu* | *06.10.21– 30.11.21* | *15%* |
| *3* | *Proiectarea sistemului* | *01.12.21 – 25.12.21* | *15%* |
| *4* | *Realizarea sistemului* | *10.01.22 – 05.03.22* | *35%* |
| *5* | *Descrierea sistemului* | *06.03.22– 01.04.22* | *10%* |
| *6* | *Estimarea costurilor sistemului* | *02.04.22– 15.04.22* | *15%* |
| *7* | *Finisarea proiectului* | *16.04.22– 14.05.22* | *5%* |

**Student**  *Rotaru Dan ( )*

**Coordonator de proiect de licență**  *Cebotari Daria ( )*

**DECLARAŢIA STUDENTULUI**

Subsemnatul Rotaru Dan, declar pe proprie răspundere că lucrarea de faţă este rezultatul muncii mele, pe baza propriilor cercetări şi pe baza informaţiilor obţinute din surse care au fost citate şi indicate, conform normelor etice, în note şi în bibliografie. Declar că lucrarea nu a mai fost prezentată sub această formă la nici o instituţie de învăţământ superior în vederea obţinerii unui grad sau titlu ştiinţific ori didactic.

Semnătura autorului

**UNIVERSITATEA TEHNICĂ A MOLDOVEI**

**FACULTATEA CALCULATOARE, INFORMATICĂ ŞI MICROELECTRONICĂ**

**DEPARTAMENTUL INGINERIA SOFTWARE ȘI AUTOMATICĂ**

**PROGRAMUL DE STUDII TEHNOLOGIA INFORMAŢIEI**

**AVIZ**

la proiectul de licenţă

**Titlul:**

Studentul Rotaru Dan gr. TI-194

1. **Actualitatea temei:**

**2.** **Caracteristica proiectului de licenţă**:

**3.** **Analiza prototipului:**

**4. Estimarea rezultatelor obţinute:**

**5. Corectitudinea materialului expus:**

**6. Calitatea materialului grafic:**

**7. Valoarea practică a proiectului:**

**8. Observaţii şi recomandări:**

**9. Caracteristica studentului şi titlul conferit :** Studentul Rotaru Dan a dat dovadă de profesionalism în elaborarea lucrării, a respectat cerinţele impuse şi a manifestat exigenţă în elaborarea şi calitatea tezei de licenţă. Din cele relatate, urmează că lucrarea de licenţă poate fi admisă spre susţinere.

Din cele relatate, urmează că lucrarea de licenţă poate fi admisă spre susţinere, cu nota \_\_\_\_\_\_\_\_\_

***Lucrarea în forma electronică corespunde originalului prezentat către susținere publică.*Coordonatorul proiectului de licenţă** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**Cebotari Daria, lect.univ**

*semnătura, data*

# REZUMAT

Una dintre principalele etape ale relației dintre client și companie/dezvoltator este procesul de alegere a unui produs. Și acesta este un proces foarte complex, pe care nu îl putem influența cu forța și vectorul pe care ni l-am dori. În primul capitol am analizat domeniul de studiu, am analizat toate platformele actuale de pe piață, și am făcut o analiză documentată, ceea ce ne permite să facem concluzii, care sunt plusurile și care sunt minusurile.

În următorul capitol (1.1) am explicat care este importanța temei, de ce avem nevoie de această platformă, cu ce platforma mea este mai bună ca alte platforme și de ce, acum pe piață este necesar de așa o platformă.

Cum am menționat în capitolul 1.1, o continuitate a acestuia a fost elaborat în capitolul 1.2, sisteme similare cu proiectul realizat. În acest capitol am exemplificat cu ce soluțiile existente pe piață nu sunt bune, și ce avantaje au, ceea ce a-și putea implementa și în proiectul realizat. Scopul, obiectivele și cerințele sistemului (1.3) – în capitolul dat am descris pentru ce e necesar în genere așa o platformă, care sunt obiectivele propuse de platformă, cu ce ea ar ajuta un utilizator simplu. De asemenea am scris cerințele sistemului, care este un punct destul de important din opinia mea. Descrierea comportamentală a sistemului (descris în capitolul 2) se referă la o metodă de descriere a comportamentului unui sistem informatic sau a unei componente a acestuia. Această descriere se concentrează pe modul în care sistemul interacționează cu utilizatorii și cu alte sisteme sau componente, în loc să se concentreze pe detaliile tehnice ale modului în care este construit sau implementat. În următoarele subcapitole am prezentat un șir de diagrame a platformei. De asemenea am descris fiecare funcțional în parte.

În următorul capitol (3) și subcapitole am realizat însăși sistemul. Am descris framework-ul Vue.JS, am exemplificat și descris sintaxa vue.js, interpolarea textului. Am descris Composition API – un nou standart adăugat în vue 3, vuex – sistem de stocare a informațiilor. De asemenea am descris și alte biblioteci, atât pe frontend cât și pe backend. Axios – este o bibliotecă node.js care face legătura dintre frontend și backend. Express.JS – backend web framework. Am descris rutarea, metode de traseu, parametrii rutei, funcții middleware. De asemenea în subcapitolul 3.3 am descris baza de date selectată – MongoDB. Am exemplificat cum MongoDB lucrează (o succintă descriere a bazei de date). Mongoose – o bibliotecă de interacțiune cu baza de date MongoDB în Node.JS. Am descris de asemenea sistemul controlului al versiunilor – Git. În capitolul 3.5 am descris la nivel de cod pe module a proiectului. Am reprezentat folder structure, careva secvențe de cod principale. În ultimul subcapitol din acest capitol (3.6) am descris testarea sistemului. Jest – bibliotecă javascript pentru testarea automată a sistemului. De asemenea a fost folosit Postman – testarea API-ului creat.

În final, am documentat produsul realizat (capitolul 4) în care am făcut un rezumat al proiectului realizat. Am descris funcționalul principal și cum de rulat proiectul, și în capitolul 5 – am făcut estimarea costurilor proiectului, în care am scris (estimat) costurile pe care le-aș putea avea.

# ABSTRACT

One of the main stages of the relationship between a client and a company/developer is the process of choosing a product. And this is a very complex process that we cannot influence by force or the vector we would like. In the first chapter, we analyzed the field of study, examined all the current platforms on the market, and conducted a documented analysis, which allows us to draw conclusions about the pros and cons.

In the next chapter (1.1), we explained the importance of the topic, why we need this platform, how my platform is better than other platforms, and why such a platform is necessary in the market now. As mentioned in chapter 1.1, a continuity of this was developed in chapter 1.2, similar systems to the project completed. In this chapter, we exemplified why the existing solutions on the market are not good and what advantages they have, which could be implemented in the completed project. The purpose, objectives, and requirements of the system (1.3) - in the given chapter, we described why such a platform is generally necessary, what are the objectives proposed by the platform, and how it would help a simple user. We also wrote down the system requirements, which is a fairly important point in my opinion. The behavioral description of the system (described in chapter 2) refers to a method of describing the behavior of an information system or one of its components. This description focuses on how the system interacts with users and other systems or components, rather than focusing on the technical details of how it is built or implemented. In the following subchapters, we presented a series of platform diagrams. We also described each functionality separately.

In the next chapter (3) and subchapters, we actually created the system. We described the Vue.JS framework, exemplified and described the Vue.js syntax, text interpolation. We described the Composition API - a new standard added in Vue 3, Vuex - information storage system. We also described other libraries, both on the frontend and backend. Axios - is a node.js library that connects the frontend and backend. Express.JS - backend web framework. We described routing, route methods, route parameters, middleware functions. Also, in subchapter 3.3, we described the selected database - MongoDB. We exemplified how MongoDB works (a succinct description of the database). Mongoose - a library for interacting with the MongoDB database in Node.JS. We also described the version control system - Git. In chapter 3.5, we described the project modules at the code level. We represented the folder structure, some main code sequences. In the last subchapter of this chapter (3.6), we described system testing. Jest - javascript library for automated testing of the system. Postman was also used - testing the created API.

Finally, we documented the completed product (chapter 4) in which we summarized the completed project. We described the main functionality and how to run the project, and in chapter 5 - we estimated the project costs, in which we wrote down (estimated) the costs that I could have.

**CUPRINS**

[REZUMAT 16](#_Toc133965000)

[ABSTRACT 17](#_Toc133965001)

[INTRODUCERE 11](#_Toc133965002)

[1 ANALIZA DOMENIULUI DE STUDIU 12](#_Toc133965003)

[1.1 Importanța temei 14](#_Toc133965004)

[1.2 Sisteme similare cu proiectul realizat 15](#_Toc133965005)

[1.3 Scopul, obiectivele și cerințele sistemului 18](#_Toc133965006)

[2 MODELAREA ȘI PROIECTAREA SISTEMUL INFORMATIC 29](#_Toc133965007)

[2.1 Descrierea comportamentală a sistemului 31](#_Toc133965008)

[2.1.1 Imaginea generală asupra sistemului 32](#_Toc133965009)

[2.1.2 Modelarea vizuală a fluxurilor 35](#_Toc133965010)

[2.1.3 Stările de tranzacție a sistemului 38](#_Toc133965011)

[2.1.4 Descrierea scenariilor de utilizare a aplicaţiei 41](#_Toc133965012)

[2.1.5 Fluxurile de mesaje şi legăturile dintre componentele sistemului 44](#_Toc133965013)

[2.2 Descrierea structurală a sistemului 46](#_Toc133965014)

[2.2.1 Descrierea structurii statice a sistemului 47](#_Toc133965015)

[2.2.2 Relatiile de dependență între componentele sistemului 50](#_Toc133965016)

[2.2.3 Modelarea echipamentelor mediului de implementare 53](#_Toc133965017)

[3 REALIZAREA SISTEMULUI 54](#_Toc133965018)

[3.1 Vue.JS - Front end web framework 55](#_Toc133965019)

[3.1.1 Sintaxa șablonului 56](#_Toc133965020)

[3.1.2 Interpolarea textului 56](#_Toc133965021)

[3.1.3 ​Composition API 56](#_Toc133965022)

[3.1.5 Vuex – model de management a stărilor sistemului 59](#_Toc133965023)

[3.1.6 Axios – interacțiunea frontend – backend prin trimiterea cererilor HTTP 60](#_Toc133965024)

[3.2 Express.JS – Back end web framework 61](#_Toc133965025)

[3.2.1 Rutarea 62](#_Toc133965026)

[3.2.2 Metode de traseu 62](#_Toc133965027)

[3.2.3 Parametrii rutei 63](#_Toc133965028)

[3.2.4 Funcțiile middleware 64](#_Toc133965029)

[3.3 MongoDB – sistem de gestiune a bazelor de date 65](#_Toc133965030)

[3.3.1 Mongoose 66](#_Toc133965031)

[3.4 Git – sistem control al versiuniilor 67](#_Toc133965032)

[3.5 Descrierea la nivel de cod pe module 69](#_Toc133965033)

[3.6 Testarea sistemului 74](#_Toc133965034)

[3.6.1 Jest 74](#_Toc133965035)

[3.6.2 Postman 76](#_Toc133965036)

[4 DOCUMENTAREA PRODUSULUI REALIZAT 77](#_Toc133965037)

[5 ESTIMAREA COSTURILOR PROIECTULUI 78](#_Toc133965038)

[CONCLUZII 79](#_Toc133965039)

[ANEXA A 81](#_Toc133965040)

# INTRODUCERE

În era digitală actuală, produsele software reprezintă o necesitate esențială în multe domenii, inclusiv în afaceri, educație, divertisment și în viața de zi cu zi a oamenilor. În același timp, cumpărarea și vânzarea produselor software reprezintă un proces complex și important pentru atât dezvoltatorii, cât și utilizatorii acestora. Vânzarea și cumpărarea produselor software implică transferul drepturilor de utilizare și distribuție a unui anumit program de la dezvoltator către utilizator. Acest proces se realizează prin intermediul unei licențe de utilizare a software-ului, care reprezintă un acord legal între cele două părți. Dezvoltatorii de software au nevoie de vânzarea produselor lor pentru a putea finanța continuarea dezvoltării și îmbunătățirii acestora, precum și pentru a acoperi costurile de marketing și distribuție. În același timp, utilizatorii de software caută soluții pentru nevoile lor specifice și sunt dispuși să plătească pentru produse de calitate, care să le ofere un beneficiu real. Produsele software sunt utile în multe domenii, iar interesul pentru achiziționarea lor poate veni din partea unor persoane fizice sau juridice. Astfel, utilizatorii individuali pot fi interesați de programe pentru îmbunătățirea productivității, editare foto și video, jocuri și multe altele. În același timp, întreprinderile caută soluții software pentru gestionarea proceselor de afaceri, contabilitate, managementul relațiilor cu clienții și multe alte activități.

În elaborarea acestei platforme, au fost analizate toate platformele similare de pe piață. Care, în opinia mea nu sunt create pentru o utilizare vastă a utilizatorilor. Majoritatea platformelor existente nu facilitează procesul de gestionare a produselor software. Da, ele sunt predispuse să facă acel “schimb” dintre cumpărător și vânzător, însă majoritatea sunt greu de accesat, nu au o funcționalitate vastă, greu de perceput utilizatorilor simpli, ba chiar și dezvoltatorilor.

Platforma creată facititează procesul de vânzare și cumpărare a produselor software. Ea ajută dezvoltatorii să-și vândă produsul creat iar cumpărătorii să achiziționeze pentru uz personal sau pentru o dezvoltare în continuare a acesteia. Dezvoltatorii pot simplu și ușor să-și plaseze anunțurile sale pe site, iar cumpărătorii au o gamă de funcționalități largi pentru a simplifica procesul de interacțiune cu dezvoltatorii sau însăși platformă.

În cadrul acestei platforme au fost utilizate o serie de tehnologii și instrumente pentru a crea o experiență de utilizare fluidă și intuitivă. Printre acestea se numără tehnologii de programare web, baze de date, rețele de distribuție de conținut și multe altele. Pentru partea de front-end a platformei, s-au utilizat tehnologii precum HTML, CSS și JavaScript pentru crearea interfeței utilizatorului și a elementelor vizuale. În plus, s-a folosit biblioteca Vue.js, pentru a simplifica dezvoltarea interfețelor complexe și pentru a optimiza performanța. Vuex a fost de asemenea utilizat pentru o stocare eficientă a datelor. Pe partea de back-end, platforma a utilizat diverse tehnologii, Node.JS, Express.JS, Socket.IO ș.i altele, pentru dezvoltarea serverelor web și a aplicațiilor de bază de date. Pentru bazele de date, au fost utilizate bazele de date MongoDB în combinație cu biblioteca Mongoose, care este des utilizată în aplicațiile Node.JS.

# 1 ANALIZA DOMENIULUI DE STUDIU

Una dintre principalele etape ale relației dintre client și companie/dezvoltator este procesul de alegere a unui produs. Și acesta este un proces foarte complex, pe care nu îl putem influența cu forța și vectorul pe care ni l-am dori. Există o serie de motive pentru acest lucru.

Să fim sinceri - există puține companii sau dezvoltatori care produc un produs total unic. Toate produsele noastre poate nu sunt unice însă au anumite avantaje, dar din punctul de vedere al cumpărătorului în etapa de selecție, totul este același. În consecință, el caută un mijloc de a-și închide nevoile.

Alegerea în sine este o poveste psihologică subtilă, în care se pot juca factori brusci: nu prețul, ci ponturile, nu fiabilitatea, ci opinia unui prieten, nu funcționalitatea, ci frumusețea și netezimea interfeței. În consecință, chiar dacă sunteți mega-creier și marketing leneș au colectat toate informațiile despre client, nu vom ști niciodată pe deplin ce conduce un anumit cumpărător.

Trebuie să înțelegem clientul și identificăm corect nevoile acestuia. O nevoie este un motiv pentru a cumpăra un produs sau un serviciu. De regulă, ea este cea care mișcă o persoană atunci când decide să achiziționeze ceva. Desigur, putem argumenta, recent influența altor factori a crescut, dar din nou aceasta este o nevoie indirectă (nevoia de a părea mai rece, nevoia de a demonstra bogăția, stilul de viață etc.). Dacă nu înțelegem nevoile companie, nu vom putea vinde nimic. Sarcina pare descurajantă, dar în era internetului, nu este atât de dificilă: să adunăm informații, să intervievăm clientul.

Nu trebuie să ne împunem produsul și ne lăudăm cu funcționalitatea, ci trebuie să ajutăm clientul să-și rezolve problemele cu capacitățile software-ului sau serviciului dvs.

Software-ul joacă un rol foarte important în aproape toate piețele industriei. Valoarea și beneficiile software-ului nostru sunt enorme - oferă companiilor o mare oportunitate de a simplifica multe dintre operațiunile lor și oferă soluții rentabile pentru procedurile lor de afaceri.

Parteneriatul este un element important al succesului. Dacă există și alte persoane interesate de vânzări în afară de noi, acest lucru va duce în mod inevitabil la vânzări mai mari.

Interesul de a crea este destul de simplu: pentru a face un preț pentru dealeri mai mic decât de vânzare cu amănuntul. În cazul în care pentru a căuta parteneri nu este, de asemenea, o problemă. Trebuie să facem o secțiune separată de afiliere și să descrim termenii de cooperare și reduceri. Nu este nevoie să economisim pe parteneri. Este necesar să creăm condiții în care alte persoane sau servicii să vă ofere produsul clienților finali.

În general, vânzarea software-ului este o afacere foarte dificilă și supărătoare. Este posibil ca să întâmpinăm o problemă aparent simplă, dar săpăturile, se pare că totul nu este atât de simplu, iar această problemă nu va sta în planul de dezvoltare, ci va fi aproape de vânzare. Aș dori să ușurez viața dezvoltatorilor/programatorilor, autorilor de software.

În ultimele decenii, tehnologia informației a adus schimbări majore în ceea ce privește modul în care facem afaceri și cum cumparam și vindem produse. Comertul electronic (e-commerce) reprezintă o modalitate prin care companiile pot vinde produse și servicii prin intermediul internetului și tehnologiei informației. Acest lucru face ca procesul de cumpărare și vânzare să fie mult mai ușor și mai rapid pentru consumatorii finali, dar și pentru companii.

Tehnologia informației a revoluționat modul în care companiile își fac afacerile și cum interacționează cu clienții lor. Cu ajutorul tehnologiei, companiile pot avea o prezență online și pot oferi produse și servicii prin intermediul internetului. Acest lucru face ca procesul de cumpărare și vânzare să fie mai ușor și mai rapid pentru consumatorii finali, dar și pentru companii.

Companiile care utilizează tehnologia informației în comerțul electronic beneficiază de multe avantaje. Acestea includ:

* Accesibilitate: Companiile pot oferi produse și servicii prin intermediul internetului, ceea ce face ca acestea să fie disponibile în orice moment și oriunde în lume.
* Costuri reduse: Companiile pot economisi bani prin vânzarea produselor și serviciilor prin intermediul internetului, deoarece nu mai trebuie să plătească pentru chirie, utilități și alte costuri asociate cu o locație fizică.
* Îmbunătățirea experienței clienților: Companiile pot utiliza tehnologia pentru a îmbunătăți experiența clienților prin intermediul unui design intuitiv al site-ului și a unui proces de cumpărare ușor de utilizat.

Cu apariția internetului și a tehnologiei informației, companiile au început să își dezvolte prezența online prin intermediul site-urilor web și platformelor de comerț electronic. Această creștere continuă a comerțului electronic a fost influențată de mai mulți factori, cum ar fi:

* Accesibilitatea internetului: În prezent, internetul este disponibil aproape peste tot în lume. Aceasta face ca cumpărăturile online să fie mai ușoare și mai accesibile pentru consumatori.
* Dezvoltarea tehnologiei: Dezvoltarea tehnologiei a făcut ca procesul de cumpărare și vânzare să fie mai rapid și mai eficient.
* Încrederea consumatorilor: Odată cu creșterea experienței și încrederea consumatorilor în comerțul electronic, aceștia sunt mai dispuși să facă achiziții online.

Tehnologia informației joacă un rol crucial în comerțul electronic. Fără accesul la internet și tehnologia informației, companiile ar avea probleme în a-și dezvolta prezența online și de a oferi produse și servicii prin intermediul internetului. Cu ajutorul tehnologiei informației, companiile pot:

* Dezvolta site-uri web și platforme de comerț electronic pentru a vinde produse și servicii online
* Utiliza tehnologia pentru a îmbunătăți experiența clienților și procesul de cumpărare

Utiliza tehnologia pentru a colecta și analiza datele clienților pentru a îmbunătăți serviciile și produsele oferite.

# 1.1 Importanța temei

Platforma creată va putea ușura și facilita semnificativ procesul de gestionare a proselor IT personale. Prin intermediul acesteia programatorii (sau vânzătorii) își vor putea vinde propriile sale proiecte simplu și rapid. Integrarea github va fi un avantaj major, ceea ce nu-l au alți concurenți. Vânzătorii vor putea “într-un click” să-și integreze proiectele de pe github pe platformă. De asemenea el va avea posibilitatea să stilizeze pagina produsului oricum își dorește acesta.

Programiștii, dezvoltatorii vor putea obține o nouă sursă de venit, ei își vor putea vinde propriile produse care poate le-a făcut pentru sine, pentru dezvoltare personală, sau poate, pentru oricare alt motiv, însă acum ei vor putea obține un oarecare profit. Cumpărarea unui cod sursă vă poate economisi mult timp și bani!

Investitorii sau cumpărătorii – persoane interesate în acele produse, programe, aplicații, care poate pentru sine, sau poate pentru uz larg, sau pentru o promovare eficientă în idee, nu contează pentru care scopuri este necesară. La moment nu sunt așa platforme “ușoare” și eficiente de gestionare a produselor IT.

Tehnologia informației în comerțul electronic reprezintă un subiect de mare importanță în lumea afacerilor de astăzi. Comerțul electronic, sau e-commerce-ul, este definit ca fiind procesul de cumpărare și vânzare de bunuri și servicii prin intermediul internetului sau a altor sisteme electronice. Această formă de comerț a crescut rapid în ultimele decenii și este în prezent una dintre cele mai importante surse de venit pentru multe companii.

Tehnologia informației are un rol cheie în dezvoltarea și gestionarea afacerilor de comerț electronic. Ea permite companiilor să își construiască și să își administreze site-uri web de e-commerce, să ofere o experiență de cumpărare online personalizată și eficientă, să își urmărească și să își gestioneze inventarul, să își analizeze datele de vânzare și să ia decizii mai bune de afaceri.

Prin urmare, înțelegerea și utilizarea adecvată a tehnologiei informației în comerțul electronic este esențială pentru succesul afacerilor într-un mediu digital în continuă schimbare. În plus, dezvoltarea și adoptarea noilor tehnologii în acest domeniu, cum ar fi inteligența artificială, blockchain sau Internet of Things, pot oferi oportunități noi și inovatoare pentru companii de a-și îmbunătăți afacerile și de a-și crește veniturile.

În concluzie, vânzarea și cumpărarea de produse IT, inclusiv software și aplicații, sunt esențiale pentru a furniza soluții digitale pentru companii și utilizatori individuali. Există o varietate de opțiuni de achiziționare și de distribuție a acestor produse, iar dezvoltatorii și utilizatorii trebuie să exploreze diferitele canale disponibile pentru a găsi cea mai bună opțiune pentru nevoile lor specifice.

# 1.2 Sisteme similare cu proiectul realizat

În general, pe piață deja sunt multe platforme respective, însă din anumite motive ele nu au obținut o pularitate largă. Fiecare își are avantajele și dezavantajele sale. Unele din cele mai populare platforme de genul dat ar fi următoarele:

**1. Sellcodes.com** – este o platformă de vânzare-cumpărare a codului personal care are scopul de a facilita procesul de gestiune a proiectelor personale. Sellcodes.com este o piață în care vânzătorii au propriile "magazine", ceea ce înseamnă că își definesc politicile de rambursare în mod individual. Prin urmare, înainte de a cumpăra careva coduri, trebuie atent de citit descrierea de pe pagina produsului.

Dacă vânzătorul nu face nicio declarație cu privire la rambursări, să presupunem că nu se va acorda nicio rambursare. Codurile software pot fi copiate cu ușurință după cumpărare și, prin urmare, nu pot fi "returnate". Prin urmare, rambursările sunt de obicei o excepție de la regulă, cu excepția cazului în care vânzătorul prevede altfel. Un avantaj mare pe care îl are această platformă este că ea are deja încorporată funcția de payments (vânzătorul poate direct pe platformă să-și vândă codul), iar platforma poate fi în calitate de garant, care gestionează plățile efectuate prin intermediul unui furnizor de plăți Mangopay. Un alt avantaj pe care îl are este că are o politică severă de protecția a datelor personale. Ok, deci am discutat despre dezavantajele acesteia. Un mare dezavantaj care a jucat un rol major (din opinia mea), nu este clar și înțeles de ce această platformă este mai bună decât altele, este destul de greu să navighezi și cel mai important să găsești anume ceea de ce ai nevoie.

În figura 1 putem observa pagina principală a platformei, care este, de fapt însăși produsele IT.

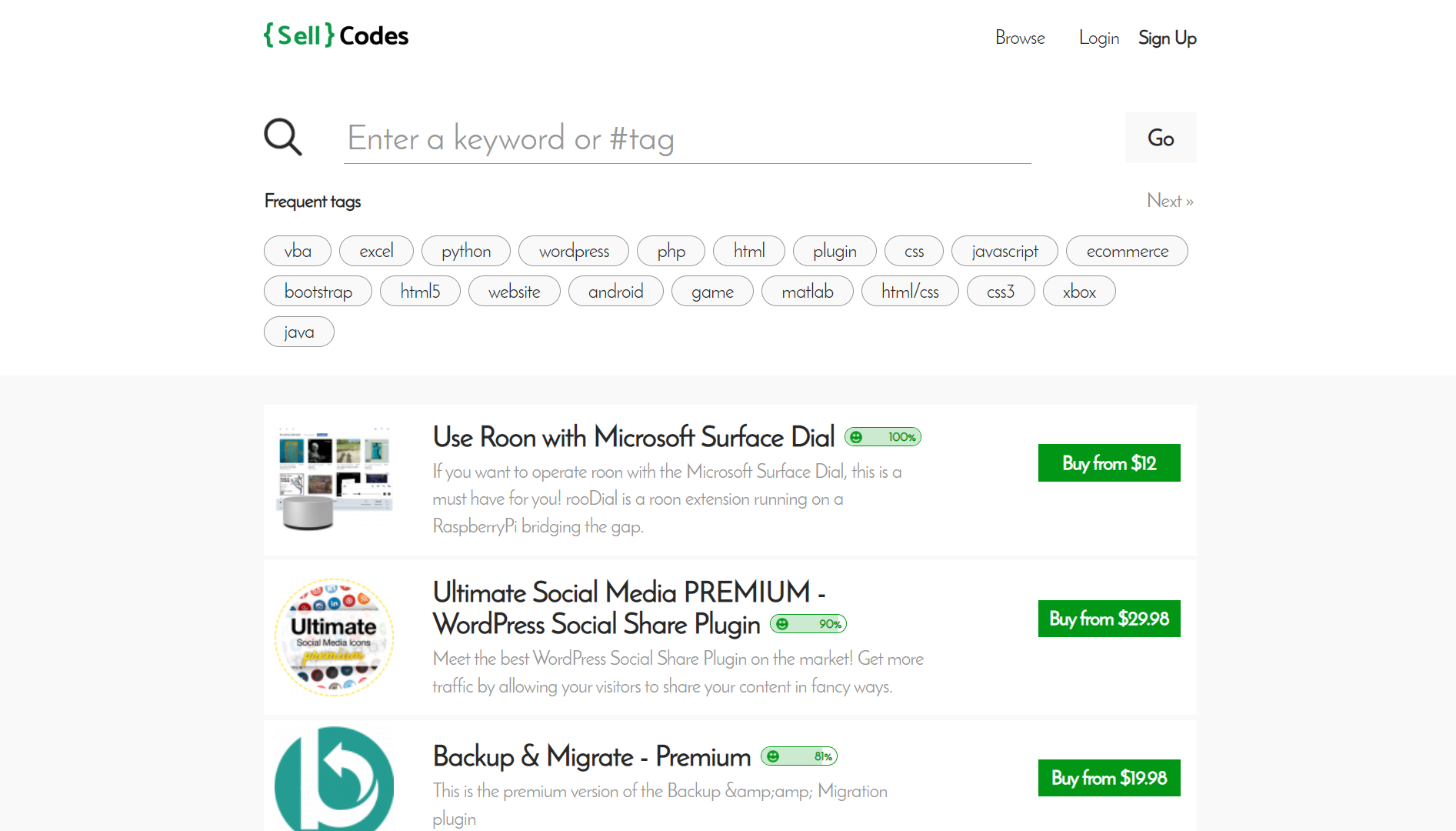


Figura 1 – Pagina principală a platformei Sell Codes

**2. Piecex** – este o altă platformă de vânzare-cumpărarea a produselor IT personale, cu practic același scop ca platforma anterioară (de gestiune a proiectelor pesonale). Avantajul PieceX major este asistența. Asistența vânzătorului este un serviciu furnizat de vânzătorii PieceX pentru produsele selectate la discreția vânzătorului. Acest suport înseamnă că vânzătorul vă va fi disponibil cu privire la orice problemă care utilizează elementul. În afară de suportul de bază inclus, fiecare produs are termeni proprii în ceea ce privește asistența. Acești termeni sunt definiți de vânzător și sunt suplimentari suportului de bază.

Produsele, inclusiv sprijinul vânzătorului, îl vor menționa în mod clar pe pagina lor de produse. Dacă un produs nu precizează că suportul este inclus, înseamnă că vânzătorul nu oferă asistență vânzătorului. Produsele cu cod sursă de la PieceX sunt vândute așa cum este. Dacă produsul include asistență pentru vânzător, se aplică următoarele:

* Produsele cu suport disponibil vor primi sprijin din partea vânzătorului pentru o perioadă de până la 6 luni de la cumpărare.
* Răspunsuri directe din partea vânzătorului cu privire la întrebările tehnice.
* Actualizări ale codului sursă cu privire la erorile raportate.
* Instrucțiuni din partea vânzătorului cu privire la funcționalitățile codului sursă sau informații care nu sunt incluse în Ghidul utilizatorului.

Sprijinul vânzătorului este oferit de vânzătorul produsului și supravegheat de PieceX. PieceX efectuează operațiuni de arbitraj și mediere cu privire la orice litigiu sau conflict cu privire la încălcarea sprijinului vânzătorului. Echipa de asistență PieceX va investiga circumstanțele conflictului atunci când cumpărătorul sau vânzătorul le ridică de către contactarea PieceX și furnizarea dovezilor justificative pentru afirmația lor. Vom lua o decizie pe baza tuturor informațiilor furnizate de ambele părți. Decizia luată de echipa PieceX este definitivă și obligatorie pentru ambele părți ale litigiului. Un alt avantaj pe care îl are această platformă este UI-ul. Spre deosebire de platforma anterioară aici este cu mult mai clar și simplă să găsești ceea de ce ai nevoie. Totul practic este clar și înțeles și cel mai important este un UX (user experience) bun.

În figura 2 putem observa pagina principală a platformei PieceX, de asemenea putem observa chiar pe pagina principală în prim plan avantajele acestei platforme.

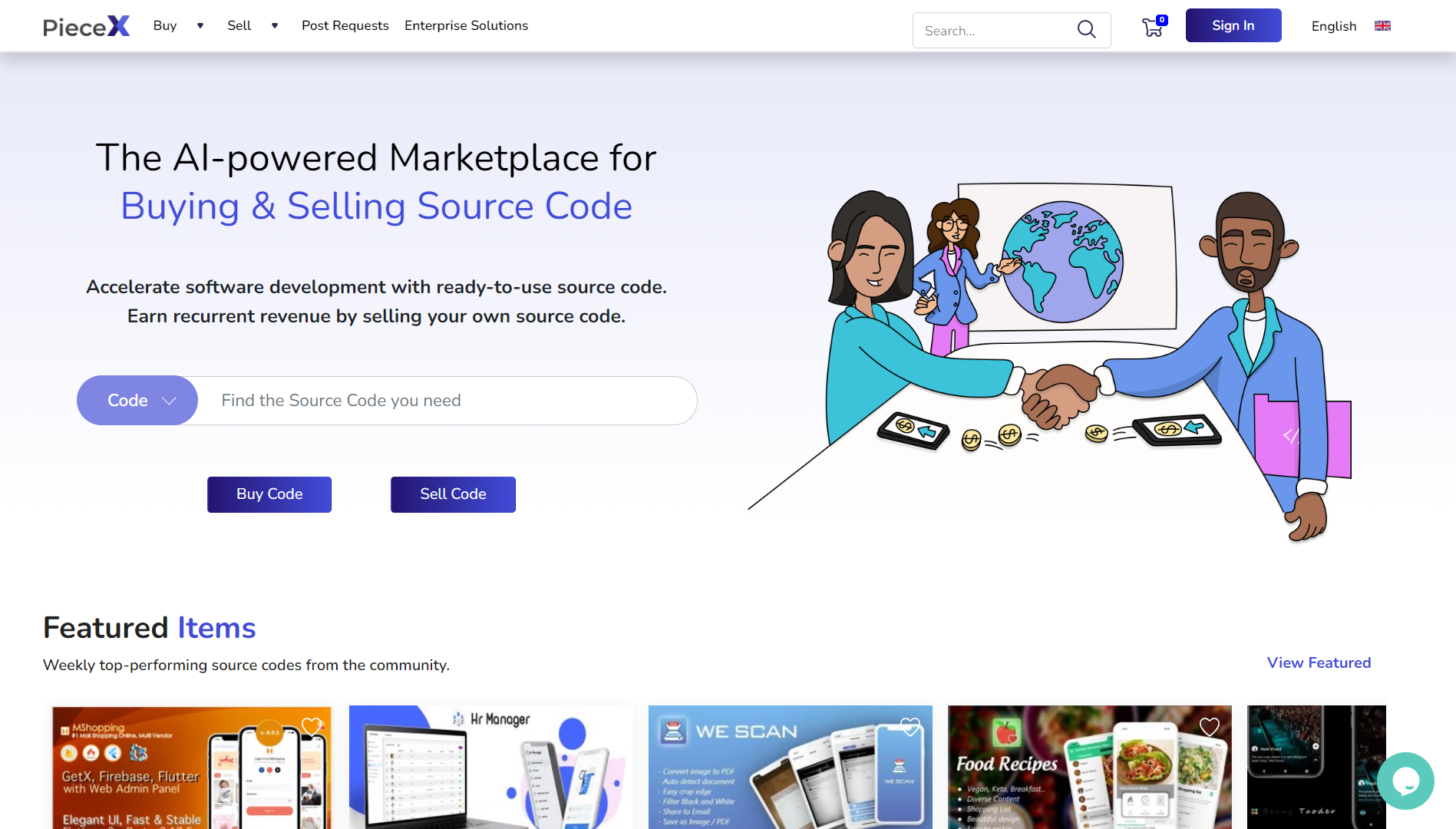
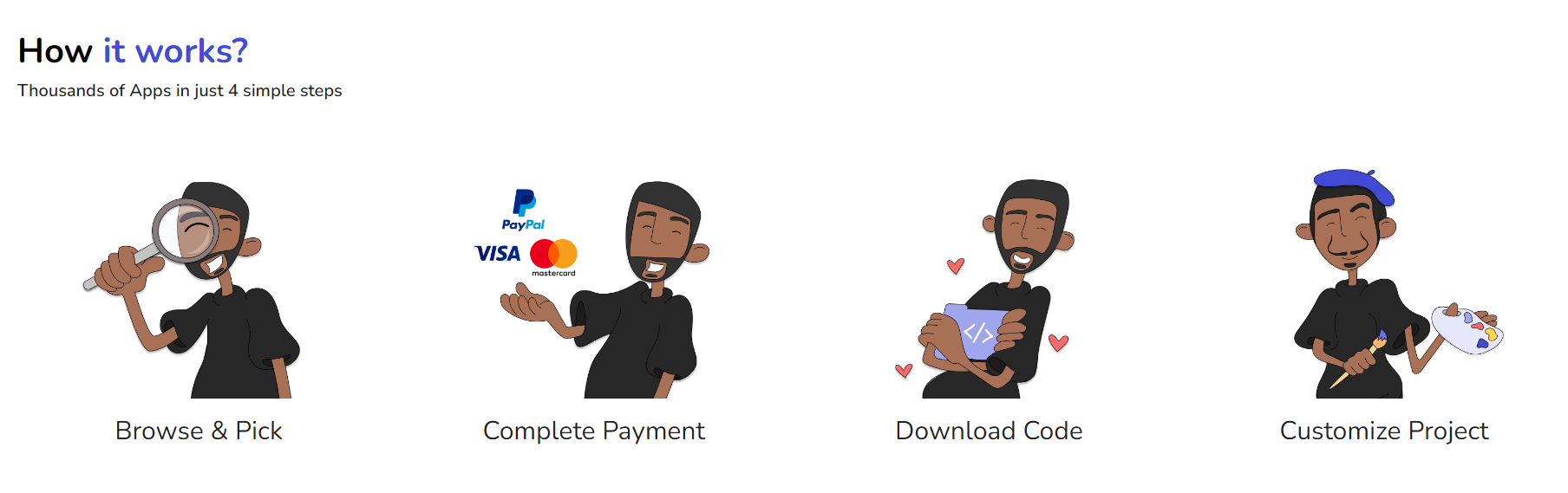


Figura 2 – Pagina principală a platformei PieceX

Avantajele sunt puse în prim-plan, ceea ce dă utilizatorilor o înțelegere clară cum lucrează platforma.



**3. SellMyApp** – este o altă platformă de gestionare a software-lui, care însă are o direcție puțin diferită, ea este destinată pentru vânzarea aplicațiilor și jocurilor în mare parte. Scopul ei primordial este facilitarea procesului de vânzare-cumpărare a jocurile, aplicațiilor mobile. SellMyApp există de la începutul anului 2012 și de atunci am ajuns să realizăm că utilizatorii au nevoie de un set de instrumente care să le permită să lucreze rapid și să-și extindă portofoliile de aplicații. Echipa noastră moderează cu atenție fiecare cod care este trimis pe piață în conformitate cu standardele noastre ridicate. Dezvoltatorii sunt respinși sau li se cere să-și îmbunătățească codul și documentația pentru a se potrivi nivelului nostru de calitate. SellMyApp oferă o soluție completă și tot ce aveți nevoie Pentru a vă lansa propria aplicație sau joc fără nicio experiență de codificare!

Procesul de gestionare platformei respective este descris foarte simplu și ușor și este construit din 3 pași:

1. Alegeți aplicația sau jocul din selecția noastră de șabloane.
2. Utilizați serviciul nostru de personalizare și cu propriul manager de proiect dedicat va fi decisă o temă.
3. Stai pe spate și lasă-ne să facem toate lucrările pentru tine. Acum aveți propria aplicație nouă!

Dezavantajul cel mai mare a platformei respective este că ea este destinată doar pentru gestionarea aplicațiilor mobile, jocurilor, nu însă și alte secvențe de cod, aplicații desktop, ș.a.m.d.

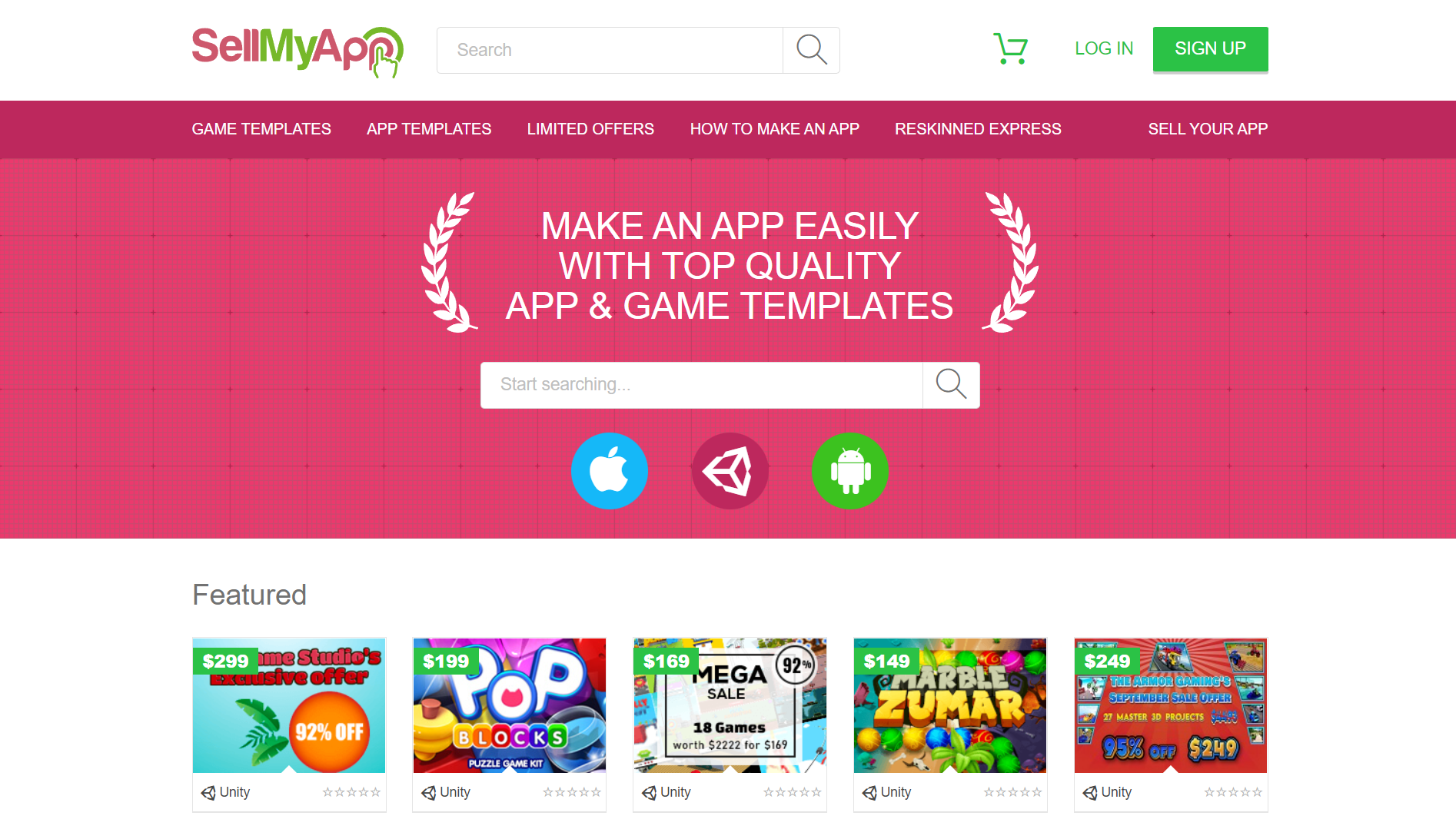


Figura 3 – Pagina principala a platformei SellMyApp

# 1.3 Scopul, obiectivele și cerințele sistemului

Proiectul care va fi realizat va fi o aplicație sau platformă WEB, care, va putea fi rulat cross-platform, singura condiție este ca el să fie conectat la internet, nu contează de plaformă – Windows, Android, IOS, Linux, Chrome OS, etc.

Platforma mea va fi destinat oricărui grup de oameni, fără careva limite de utilizare, toți vor putea avea access la ea indiferent de clasa sa.

Ea va avea un funcțional vast și un șir de obiective, așa ca:

1. proiectarea bazei de date
   1. va fi proiectată baza de date SQL sau MongoDB
2. elaborarea frontend-ului (user-friendly UI/UX);
   1. în mare parte va fi scris “de la zero”
   2. voi utiliza twig sau plain html
   3. voi utiliza SCSS pentru a facilita procesul de dezvoltare a stilizării paginilor
   4. voi utiliza JavaScript pentru a adăuga dinamicitatea site-ului, AJAX, WebSockets…
3. elaborarea backend-ului;
   1. va fi scris în limbajul de programare JavaScript (node.js)
   2. voi utiliza careva librării care vor spori și îmbunătăți procesul de dezvoltare a serverului
4. integrare github;
   1. utilizatorii vor putea să se înregistreze, logare cu github,
   2. utilizatorii (sau programatorii – vanzatorii) vor putea direct să-și adauge repozitoriile sale private pe site
5. elaborarea funcționalului de logare/înregistrare;
   1. abilitatea de logare sau înregistrare nu doar cu github dar și cu credențiale de tip login password
6. elaborarea funcționalului de afișare, creare, modificare, ștergere a anunțurilor;
   1. vânzătorii vor putea crea, modifica sau șterge anunțurile sale
   2. cumpărătorii vor putea vizualiza toate anunțurile adăugate
   3. vor fi adăugate o serie de criterii de validare, filtrare, căutare ș.a.m.d., care vor ușura să găsești ce ai nevoie
7. elaborarea funcționalului de comentarea, distribuirea anunțurilor;
   1. anunțurile vor putea fi comentate de utilizatori în public, poate careva întrebări generale, care vor fi date răspunsuri pentru toți utilizatorii
8. elaborarea funcționalului de interacțiunii dintre vânzător-cumpărător (chat);
   1. va fi creată posibilitatea de comunicare dintre vânzător și cumpărător
   2. va fi implementat chat-ul care va fi proiectat utilizând tehnologii moderne (websockets), care va avea realtime update și va fi securizat
9. crearea profilurilor vanzător/cumpărător;
   1. fiecare utilizator (indiferent de clasa sa – vânzător sau cumpărător) vor avea un profil personal public
   2. fiecare utilizator va putea vizualiza aceste profiluri, și poate să-și facă o concluzie referitor la oarecare utilizator

Sistemul va fi proiectat utilizând multe tehnologii. Limbajul pe care l-am selectat a fost JavaScript (Node.JS). Node.js este o platformă de partea serverului construită pe motorul JavaScript (V8 Engine) de la Google Chrome. Node.js a fost dezvoltat de Ryan Dahl în 2009, iar cea mai recentă versiune a sa este v0.10.36. Definiția nodului.js furnizată de documentația sa oficială este după cum urmează. Node.js este o platformă construită pe runtime JavaScript chrome pentru a construi cu ușurință aplicații de rețea rapide și scalabile. Node.js utilizează un model I/O bazat pe evenimente, fără blocare, care îl face ușor și eficient, perfect pentru aplicațiile în timp real care rulează pe dispozitive distribuite. Node.js este un mediu runtime open source, cross-platform pentru dezvoltarea aplicațiilor de partea serverului și a aplicațiilor de rețea. Node.js aplicațiile sunt scrise în JavaScript și pot fi rulate în node.js runtime pe OS X, Microsoft Windows și Linux.

**Caracteristicile principale ale limbajului Node.JS**

În urma sunt unele dintre caracteristicile importante care fac Node.js prima alegere a arhitecților software-ul.

**Asincrone și bazate pe evenimente** - Toate API-urile bibliotecii Node.js sunt asincrone, adică non-blocare. Aceasta înseamnă, în esență, un server bazat pe nod.js nu așteaptă niciodată ca un API să returneze date. Serverul se mută la următorul API după apelarea acestuia și un mecanism de notificare a evenimentelor nodului.js ajută serverul să obțină un răspuns de la apelul API anterior.

**Foarte rapid** - Fiind construit pe Google Chrome V8 JavaScript Engine, Node.js bibliotecă este foarte rapid în executarea codului.

**Single Threaded but Highly Scalable** − Node.js utilizează un singur model filetat cu looping eveniment. Mecanismul de eveniment ajută serverul să răspundă într-un mod care nu blochează și face serverul foarte scalabil, spre deosebire de serverele tradiționale care creează fire limitate pentru a gestiona solicitările. Node.js utilizează un singur program filetat și același program poate oferi servicii unui număr mult mai mare de solicitări decât serverele tradiționale, cum ar fi Apache HTTP Server.

**Fără buffering** − Node.js aplicațiile nu tamponează niciodată date. Aceste aplicații pur și simplu de ieșire a datelor în bucăți.

**Licență** − Node.js este eliberat sub licența MIT.

**Cine utilizează Node.JS**

În urma este link-ul de pe wiki-ul github care conține o listă exhaustivă de proiecte, aplicații și companii care utilizează Node.js. Această listă include eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo! și Yammer pentru a numi câteva.

**Când trebuie de utilizat Node.JS**

Urmează zonele în care Node.js se dovedește a fi un partener tehnologic perfect:

* I / O legate de aplicații
* Aplicații de streaming de date
* Aplicații intensive de date în timp real (DIRT)
* Aplicații bazate pe API-uri JSON
* Aplicații cu o singură pagină (Single Page Application – SPA)

Mai jos sunt anexate software-ul utilizat pentru proiectarea și dezvoltarea platformei

Tabelul 1 – Proiectarea și dezvoltarea platformei

|  |  |
| --- | --- |
| Software utilizat | Descriere |
| Sistemul de operare | Am ales sistemul de operare Windows pentru un funcțional larg al acestuia, deși se poate utilizat și Linux. |
| Limbajul JavaScript (Node.JS) | Pentru implementarea proiectului a fost utilizat limbajul de programare JavaScript (Node.JS), care este foarte rapid, și are un șir de avantaje care facilitează procesul de dezvoltare a serviciilor WEB |

**Protocoale și interfețe de comunicație**

Platforma respectivă poate rula pe toate tipurile de browsere WEB moderne, care suportă majoritatea tehnologiilor actuale. Ea va utiliza protoculul HTTP, care oferă o tehnică de comunicare prin care paginile web se pot transmite de la un computer aflat la distanță spre propriul computer.

Urmează transferul prin protocolul TCP pe portul standard 80 al serverului HTTP, ca răspuns la cererea HTTP-GET. Informații suplimentare ca de ex. indicații pentru browser, limba dorită ș.a. se pot adăuga în header-ul (antetul) pachetului HTTP. În urma cererii HTTP-GET urmează din partea serverului răspunsul cu datele cerute, ca de ex.: pagini în (X)HTML, cu fișiere atașate ca imagini, fișiere de stil (CSS), scripturi (Javascript), dar pot fi și pagini generate dinamic (SSI, JSP, PHP și ASP.NET). Dacă dintr-un anumit motiv informațiile nu pot fi transmise, atunci serverul trimite înapoi un mesaj de eroare. Modul exact de desfășurare a acestei acțiuni (cerere și răspuns) este stabilit în specificațiile HTTP.

HTTP va fi securizat prin intermediul unui certificat SSL – deci atunci deja avem HTTPS, care ne oferă o securitate sporită a datelor prin criptarea acestora.

De asemea voi utiliza WS (Web Sockets) care ne oferă o conexiune real-time – care va fi utilizat pentru comunicarea dintre vânzător și cumpărător.

WebSocket este un protocol de comunicare bidirecțională în timp real, care permite transferul de date între client și server fără a fi nevoie să se efectueze o cerere suplimentară de la client la server. Acesta oferă o alternativă mai eficientă și mai rapidă la tehnologiile tradiționale de comunicare, cum ar fi AJAX (Asynchronous JavaScript and XML).

WebSocket se bazează pe TCP (Transmission Control Protocol), oferind o conexiune persistentă între client și server. După inițierea conexiunii, serverul poate trimite date către client, iar clientul poate trimite date către server, fără a fi nevoie să se mai facă cereri suplimentare. Acest lucru face ca comunicarea între client și server să fie mult mai rapidă și mai eficientă decât în cazul tradițional al tehnologiilor web.

Protocolul WebSocket permite interacțiunea dintre un browser web (sau altă aplicație client) și un server web cu alternative aeriene mai mici decât semi-duplex, cum ar fi votarea HTTP, facilitând transferul de date în timp real de la și către server. Acest lucru este posibil prin furnizarea unui mod standardizat pentru server de a trimite conținut clientului fără a fi solicitat mai întâi de către client și permițând transmiterea mesajelor înainte și înapoi, menținând în același timp conexiunea deschisă. În acest fel, o conversație în curs de desfășurare bidirecțională poate avea loc între client și server. Comunicațiile se fac de obicei prin portul TCP numărul 443 (sau 80 în cazul conexiunilor nesecurizate), ceea ce este benefic pentru mediile care blochează conexiunile la Internet non-web utilizând un paravan de protecție.

Spre deosebire de HTTP, WebSocket oferă comunicare full-duplex. În plus, WebSocket permite fluxuri de mesaje în partea de sus a TCP. TCP singur se ocupă cu fluxuri de octeți cu nici un concept inerent de un mesaj. Înainte de WebSocket, portul 80 de comunicare full-duplex a fost atins folosind canale Comet; cu toate acestea, punerea în aplicare Comet este nontrivial, și din cauza tcp handshake și antet HTTP aeriene, este ineficient pentru mesaje mici. Protocolul WebSocket își propune să rezolve aceste probleme fără a compromite ipotezele de securitate ale web-ului.

Avantajele utilizării WebSocket sunt multiple, printre acestea se numără:

1. Latența redusă: Prin utilizarea unei conexiuni persistente între client și server, WebSocket reduce semnificativ timpul necesar pentru a transfera datele între aceștia. Acest lucru face ca trimiterea și primirea datelor să fie mult mai rapide decât în cazul tehnologiilor web tradiționale, cum ar fi AJAX.
2. Comunicare bidirecțională: Unul dintre cele mai mari avantaje ale WebSocket este posibilitatea de a permite comunicarea bidirecțională între client și server. Aceasta permite serverului să trimită informații către client fără a fi necesară o cerere suplimentară de la client la server. Aceasta este o opțiune utilă în cazul aplicațiilor web care necesită o interacțiune continuă între utilizator și server.
3. Reducerea traficului de rețea: Prin utilizarea WebSocket, traficul de rețea poate fi redus semnificativ. Acest lucru se datorează faptului că conexiunea este persistentă, ceea ce face ca cererile suplimentare să fie eliminate. Aceasta înseamnă că mai puține date sunt transferate între client și server, ceea ce duce la o reducere a traficului de rețea.
4. Eficiență crescută: Utilizarea WebSocket poate duce la o creștere semnificativă a eficienței în ceea ce privește procesarea și transferul datelor. Prin eliminarea cererilor suplimentare și prin reducerea traficului de rețea, se poate reduce și sarcina de lucru a serverului, ceea ce poate duce la o creștere a eficienței și a performanței.
5. Implementare ușoară: Implementarea WebSocket este relativ ușoară și nu necesită o experiență vastă în programare. De asemenea, există multe biblioteci și framework-uri disponibile care facilitează implementarea acestei tehnologii.

În concluzie WebSocket este o tehnologie puternică și eficientă, care permite transferul de date în timp real între client și server. Aceasta oferă o alternativă mai rapidă și mai eficientă la tehnologiile web tradiționale, ceea ce face ca comunicarea între client și server să fie mult mai rapidă și mai eficientă. De asemenea, oferă posibilitatea de a crea aplicații web interactiv și în timp real, cum ar fi jocuri online sau aplicații de chat.

Sistemul meu va stoca informația în baza de date SQL sau NoSQL – MongoDB.

MongoDB este o bază de date NoSQL open-source orientată pe documente. Acestă bază de date beneficiază de suport din partea companiei 10gen. MongoDB face parte din familia de sistemelor de baze de date NoSQL. Diferența principală constă în faptul că stocarea datelor nu se face folosind tabele precum într-o bază de date relațională, MongoDB stochează datele sub formă de documente JSON cu scheme dinamice.[2] https://ro.wikipedia.org/wiki/MongoDB

MongoDB este o bază de date open-source NoSQL scrisă în C++. Aceasta poate conține mai multe baze de date, colecții și indecși. În unele cazuri (baze de date și colecții ) aceste obiecte pot fi create implicit. Odată create, ele se găsesc în catalogul sistemului db.systems.collection, db.system.indexes. Colecțiile conțin documente (BSON). Aceste documente conțin la rândul lor mai multe câmpuri. În MongoDB nu există câmpuri predefinite spre deosebire de bazele de date relaționale, unde există coloanele care sunt definite în momentul în care tabelele sunt create. Nu există schemă pentru câmpurile dintr-un document, acestea precum și tipurile lor pot varia. Astfel nu există operația de „alter table” pentru adăugare de coloane. În practică este obișnuit ca o colecție să aibă o structură omogenă, deși nu este o cerință, colecțiile putând avea structuri diferite. Această flexibilitate presupune ușurință în migrarea și modificarea imaginii de ansamblu asupra datelor.

Caracteristicile MongoDB

* Stocarea datelor sub formă de documente ;
* Baza de date MongoDB stochează obiecte (documente). Aceste documente reduc nevoia de join;
* Prezintă support pentru indexare
* Indexarea pe fiecare din atribute se face în modul tradițional (RDBMS) asupra cheilor de regăsire ale documentelor;
* Disponibilitate - Disponibilitatea datelor este asigurată printr-un proces automat de failover;
* Auto-Sharding
* Shardingul sau partiționarea datelor pe orizontală se face automat. Citirile și scrierile sunt distribuite pe partiții. Lipsa joinurilor face ca interogările distribuite să fie rapide;
* Limbajul de interogare
* Limbajul de interogare este îmbunatatit și păstreaza principii SQL si C++;
* Modificări rapide
* Map Reduce

Tabelul 2 – Paralela MySQL versul MongoDB

|  |  |
| --- | --- |
| Termeni MySQL | Concepte MongoDB |
| Bază de date | Bază de date. |
| Tabelă | Colecție |
| Index | Index |
| Rând | Document BSON |
| Coloană | Câmp BSON |
| Join | Încapsulare și legatură |
| Cheie primară | Câmpul cheie unică \_id |
| Group by | Agregare |

Parțial datele vor fi criptate (spre exemplu parolele, sau alte informații de genul dat) cu prioritate sporită, unde avem nevoie o politică de securitate mai severă. Ea trebuie să fie securizată din oricare punct de vedere. În deosebi trebuie să fie securizat procesul de încărcare sau integrare a codului personal prin upload sau utilizând github. Codul sursă nu trebuie în nici un caz să fie obținut de alte persoane terțe, deci asta este cel mai important punct și trebuie să fie cel mai important. Da, desigur nu trebuie să uităm de procesul de logare și înregistrare, care este la fel de important. Parolele trebuie să fie criptate, ca ele să nu fie stocate în format plain text, doar criptate, un bun sistem pentru asta ar fi md5, care poate fi securizat destul de bine dacă pe lângă doar criptarea parolei mai adăugăm o concatinare cu o cheie unică la aceasta. Așa ca chiar dacă cei care vor obține această parolă criptată, să nu o poată decripta nicidecum.

Platforma dată are o perspectivă vastă, va avea careva avantaje asupra sistemelor deja existente. Un moment care va concura cu alte sisteme este integrarea github. Pe majoritatea platformelor utilizatorii trebuie de sine stătător să-și încarce codul pe site, de fiecare dată trebuie să-și bată capul despre versionare, informații care nu este destinată pentru publicul larg, (gen datele bazei de date, sau careva alte fișiere) care pot fi ignorate utilizând git prin intermediul fișierului .gitignore, unde specificăm care fișiere nu trebuie să fie stocate în git. Platforma va utiliza o bază de date relațională (SQL) sau nerelațională (NoSQL).

GitHub este un site și un serviciu pe care îl auzim pe geeks rave despre tot timpul, totuși o mulțime de oameni nu înțeleg cu adevărat ce face. Doriți să știți ce este vorba despre tot ce are gutubul GitHub? Citiți mai departe pentru a afla. [1]

"Git" în GitHub

Pentru a înțelege GitHub, mai întâi trebuie să aveți o înțelegere a lui Git. Git este un sistem de control al versiunii cu sursă deschisă, lansat de Linus Trovalds - aceeași persoană care a creat Linux. Git este similar cu alte sisteme de control al versiunilor - Subversion, CVS și Mercurial pentru a numi câteva.

Deci, Git este un sistem de control al versiunii, dar ce înseamnă asta? Când dezvoltatorii creează ceva (de exemplu, o aplicație), aceștia fac modificări constante ale codului, eliberând versiuni noi până la și după prima lansare oficială (non-beta).

Versiunile de control ale versiunilor păstrează aceste revizii drept, stocând modificările într-un depozit central. Acest lucru permite dezvoltatorilor să colaboreze cu ușurință, deoarece pot descărca o nouă versiune a software-ului, pot efectua modificări și pot încărca cea mai nouă revizie. Fiecare dezvoltator poate vedea aceste noi modificări, le poate descărca și poate contribui.

În mod similar, persoanele care nu au nimic de-a face cu dezvoltarea unui proiect pot descărca fișierele și le pot utiliza. Majoritatea utilizatorilor de Linux ar trebui să fie familiarizați cu acest proces, deoarece folosirea lui Git, Subversion sau a oricărei alte metode similare este destul de comună pentru descărcarea fișierelor necesare - în special în pregătirea pentru compilarea unui program din codul sursă (o practică destul de obișnuită pentru geek-urile Linux).

Git este sistemul de control al versiunilor preferat de majoritatea dezvoltatorilor, având multiple avantaje față de celelalte sisteme disponibile. Acesta stochează modificările fișierelor într-un mod mai eficient și asigură integritatea fișierelor mai bine. Dacă sunteți interesat să cunoașteți detaliile, pagina Git Basics conține o explicație detaliată a modului în care funcționează Git.

"Hub" în GitHub

Am stabilit că Git este un sistem de control al versiunilor, similar, dar mai bun decât numeroasele alternative disponibile. Deci, ce face GitHub atât de special? Git este un instrument de linie de comandă, dar centrul în jurul căruia toate lucrurile care implică Git se învârtesc este hub-GitHub.com - unde dezvoltatorii își păstrează proiectele și se conectează cu oameni asemănători.

Git este cel mai utilizat sistem de control al versiunilor software. Git este un proiect open source matur si intretinut activ. A fost dezvoltat initial in 2005 de Linus Torvalds, celebrul creator al kernel-ului sistemului de operare Linux. Un numar uimitor de proiecte software se bazeaza pe Git pentru controlul versiunilor, inclusiv proiecte comerciale si open source. De asemenea, functioneaza bine pe o gama larga de sisteme de operare (Linux, POSIX, Windows, OS X) si IDE (Integrated Development Environments). Vom detalia in continuare ce este Git si in ce mod il puteti utiliza.

Avand o arhitectura distribuita, Git este un exemplu de DVCS (Distributed Version Control System). In loc sa detina un singur loc pentru istoricul versiunilor complete ale software-ului (asa cum era in cazul sistemelor de control al versiunilor CVS sau Subversion), in Git, copia de lucru a fiecarui dezvoltator al codului este, de asemenea, un repository care poate contine istoricul complet al tuturor modificarilor. Astfel, puteti inregistra schimbarile facute unui fisier sau set de fisiere cu care putem restaura o versiune mai veche a fisierului, puteti compara diverse modificari efectuate in timp, sau puteti vedea cine a modificat un anumit proiect cu scopul de a detecta sursa unei probleme.

Pe langa arhitectura distribuita, Git a fost proiectat avand in vedere performanta, securitatea si flexibilitatea.

**1. Performanța**

* 1. Modificarile, ramificarea, combinarea si compararea versiunilor anterioare sunt optimizate pentru performanta
  2. Cunoasterea aprofundata a atributelor comune dicteaza modul in care sunt modificate fisierele si care sunt tiparele de acces.
  3. Git nu se lasa pacalit de numele fisierelor atunci cand determina care ar trebui sa fie stocarea si istoricul versiunilor din arborele fisierelor. In schimb, Git se concentreaza asupra continutului fisierului in sine, tinand cont de faptul ca fisierele sunt deseori redenumite, impartite si rearanjate.
  4. Formatul fisierelor din repository-ul Git foloseste o combinatie de codificare delta (stocand diferentele de continut), si comprimare, stocand in mod explicit continutul directorului si metadatele obiectelor legate de versiuni.
  5. Fiind un sistem distribuit, adauga plusuri importante perfomantei

**2. Securitatea**

* 1. Git a fost proiectat avand ca prioritate integritatea codului sursa gestionat
  2. Continutul fisierelor, relatiile dintre fisiere si directoare, versiuni, taguri si commit-uri, toate aceste obiecte din repository-ul Git sunt securizate cu un algoritm de hash securizat criptografic numit SHA1. Aceasta protejeaza codul si istoricul modificarilor atat impotriva schimbarilor accidentale, cat si a celor rau intentionate si asigura ca istoricul poate fi urmarit integral.
  3. Cu Git, puteti fi sigur ca aveti un istoric de continut autentic al codului sursa.
  4. Alte sisteme de control al versiunii nu detin protectie impotriva modificarilor secrete la o data ulterioara. Aceasta poate fi o vulnerabilitate serioasa a securitatii informatiilor pentru orice organizatie care se bazeaza pe dezvoltarea de software.

**3. Flexibilitate**

* 1. Git este flexibil in mai multe privinte: in sprijinul diferitelor tipuri de fluxuri de lucru neliniare de dezvoltare, in eficienta sa atat in proiecte mici, cat si in proiecte mari si in compatibilitatea sa cu multe sisteme si protocoale existente.
  2. Este proiectat sa suporte branch-urile noi si tag-urile, iar operatiunile care le afecteaza pe acestea (imbinare, revenirea) sunt stocate de asemenea in istoricul schimbarilor. Nu toate sistemele de control al versiunilor prezinta acest nivel avansat de tracking.

**Operatiuni de baza Git**

1. Initialize (crearea unui repository) – *git init*
2. Add (adaugarea unuia sau a mai multor fisiere) – *git add*
3. Commit (aduceti modificari head-ului) – *git commit*
4. Pull (aduceti si imbinati modificarile de pe serverul de la distanta in directorul dvs. de lucru) – *git pull*
5. Push (trimiteti modificari branch-ului principal al repository-ului) – *git push*

**Operatiuni avansate Git**

1. Branching (pointere specifice unui commit)
2. Merging (combinarea schimbarilor efectuate pe mai multe branch-uri)
3. Rebasing (integrarea modificarilor de la un branch la altul)

# 2 MODELAREA ȘI PROIECTAREA SISTEMUL INFORMATIC

Modelarea informațiilor de sistem (SIM) este procesul de modelare a sistemelor complexe conectate. Modelele de informații ale sistemului sunt reprezentări digitale ale sistemelor conectate, cum ar fi instrumentele electrice și sistemele de control, puterea și sistemele de comunicații. Obiectele modelate într-un SIM au o relație 1:1 cu obiectele din sistemul fizic. Componentele, conexiunile și funcțiile sunt definite și legate așa cum ar fi în lumea reală.

Modelarea sistemului este procesul de dezvoltare a modelelor abstracte ale unui sistem, fiecare model prezentând o vedere sau o pserspectivă diferită a sistemului respectiv. Este vorba despre reprezentarea unui sistem folosind un fel de notație grafică, care acum se bazează aproape întotdeauna pe notații în Unified Modeling Language (UML).

Modelarea este o tehnică de inginerie dovedită și bine acceptată. În arhitectura clădirilor, dezvoltăm modele arhitecturale de case și clădiri înalte pentru a ajuta la vizualizarea produselor finale. În UML, un model poate fi structural, punând accent pe organizarea sistemului sau poate fi comportamental, punând accent pe dinamica sistemului. Un model este o simplificare a realității, oferind schițele unui sistem. UML, în special:

* permite specificarrea unei structuri sau comportamentul unui sistem.
* ajuta la vizualizarea unui sistem.
* oferă șablon care ghidează în construirea unui sistem.
* ajută la înțelegerea sistemului complex parte cu parte.
* documentarea deciziile luate.

Construim model astfel încât să putem înțelege mai bine sistemul pe care îl dezvoltăm. Un model poate cuprinde o privire de ansamblu asupra sistemului luat în considerare, precum și o planificare detaliată pentru proiectarea, implementarea și testarea sistemului.

Pe măsură ce valoarea strategică a software-ului crește pentru multe companii, industria caută tehnici pentru a automatiza producția de software și pentru a îmbunătăți calitatea și a reduce costurile și timpul de lansare pe piață. Aceste tehnici includ tehnologia componentelor, programarea vizuală, modelele și cadrele. Companiile caută, de asemenea, tehnici de gestionare a complexității sistemelor, pe măsură ce acestea cresc în domeniul de aplicare și scară. În special, ei recunosc nevoia de a rezolva problemele arhitecturale recurente, cum ar fi distribuția fizică, concurența, replicarea, securitatea, echilibrarea sarcinii și toleranța la erori. În plus, dezvoltarea pentru World Wide Web, deși simplifică unele lucruri, a exacerbat aceste probleme arhitecturale. Limbajul de modelare unificat (UML) a fost conceput pentru a răspunde acestor nevoi.

Obiectivele principale în proiectarea UML sunt rezumate de Page-Jones în Fundamental Object-Oriented Design în UML, sunt: oferirea utilizatorilor un limbaj de modelare vizual expresiv, gata de utilizat, astfel încât să poată dezvolta și schimba modele semnificative; furnizarea de mecanisme de extensibilitate și specializare pentru a extinde conceptele de bază; independent de anumite limbaje de programare și procese de dezvoltare; gestionarea de bază formală pentru înțelegerea limbajului de modelare; sprijinul conceptului de dezvoltare de nivel superior, cum ar fi colaborări, cadre, modele și componente.

Modelele construite pe parcursul dezvoltarii unui sistem sunt reprezentari abstracte ale sistemului. Fiecare model reflecta o anumita vedere asupra sistemului si corespunde unui nivel de detaliu.

In etapa de analiza se construiesc modele care exprima cerintele impuse sistemului. Modelele de analiza corespund unei vederi externe asupra sistemului. Se folosesc de catre client, viitorii utilizatori ai sistemului, experti ai domeniului aplicatiei, analisti, echipa de verificare si validare a sistemului.

In etapa de proiectare se construiesc modele care redau arhitectura sistemului, alocarea cerintelor pe subsisteme, distributia proceselor in sistem, sincronizarea lor, starile si tranzitiile intre stari. Alte modele descriu realizarea fizica a sistemului, echipamentele din componenta sa si repartitia componentelor program.

Fiecare vedere asupra unui sistem poate avea aspecte structurale si aspecte comportamentale. In functie de natura sistemului, unele modele pot fi mai importante decat altele. De exemplu, pentru unele sisteme sunt mai importante aspectele structurale si functionale, pentru altele aspectele temporale. Construirea modelelor este ghidata de metode. O metoda defineste o abordare reproductibila care permite obtinerea de rezultate fiabile n mod repetat. Toate domeniile cunoasterii utilizeaza metode mai mult sau mai putin sofisticate si mai mult sau mai putin formalizate. De exemplu, bucatarii utilizeaza retete de bucatarie, arhitectii construiesc planuri, muzicienii urmeaza reguli de compozitie.

Modelele sunt alcatuite din elemente de modelare care constituie concepte fundamentale pentru reprezentarea sistemelor sau a fenomenelor. Elementele de modelare sunt adesea notatii grafice. Ele constituie limbajul de modelare. In plus fata de limbajul de modelare, o metoda defineste reguli de aplicare care descriu coordonarea diferitelor puncte de vedere, nlantuirea actiunilor, ordonarea sarcinilor si repartizarea responsabilitatilor.

Principalele scopuri ale modelarii sistemelor informatice sunt:

* vizualizarea, ca mijloc de usurare a comunicarii si intelegerii;
* specificarea, prin construirea de modele precise, neambigue si complete;
* documentarea cerintelor, a solutiilor de proiectare si a modului de realizare. [2]

# 2.1 Descrierea comportamentală a sistemului

Descrierea comportamentală a sistemului se referă la o metodă de descriere a comportamentului unui sistem informatic sau a unei componente a acestuia. Această descriere se concentrează pe modul în care sistemul interacționează cu utilizatorii și cu alte sisteme sau componente, în loc să se concentreze pe detaliile tehnice ale modului în care este construit sau implementat.

Descrierea comportamentală a sistemului poate include informații despre interfețele utilizatorului, fluxurile de date și informațiile care circulă prin sistem, precum și despre modul în care sistemul răspunde la diverse evenimente și intrări. Această descriere poate fi utilizată pentru a ajuta la dezvoltarea, testarea și documentarea sistemului și poate fi deosebit de utilă în cazul sistemelor complexe sau distribuite.

În general, descrierea comportamentală a sistemului se concentrează pe modul în care sistemul se comportă într-un anumit context sau situație, precum și pe modul în care interacționează cu alte sisteme sau componente. Aceasta poate fi utilizată pentru a identifica problemele și defecțiunile din sistem, precum și pentru a îmbunătăți înțelegerea globală a funcționării sistemului. Imaginea generală a unui sistem poate fi descrisă ca fiind un ansamblu de elemente sau componente interconectate, care lucrează împreună pentru a realiza un anumit scop sau obiectiv. Sistemele pot fi de diverse dimensiuni, complexități și domenii de aplicare, de la sisteme simple, cum ar fi un ceas de mână, la sisteme complexe, cum ar fi o rețea de comunicații. Modelarea vizuală a fluxurilor se referă la utilizarea diagramelor pentru a reprezenta și a comunica fluxurile de informații, de activități și de procese într-un sistem. Aceste diagrame pot fi utilizate pentru a vizualiza și a înțelege mai ușor fluxurile de lucru în cadrul unui sistem sau proces.

Un exemplu de diagramă de modelare a fluxurilor este diagrama de flux a proceselor (Process Flow Diagram - PFD), care este utilizată pentru a reprezenta etapele și fluxul de informații într-un proces. Această diagramă poate include componente precum intrări, ieșiri, fluxul de materiale sau energie, dispozitive de control și alarme. O altă diagramă de modelare a fluxurilor este diagrama de activități, care poate fi utilizată pentru a reprezenta activitățile și acțiunile necesare pentru a realiza un obiectiv într-un sistem. Această diagramă poate fi utilizată pentru a descrie și a vizualiza procesele de afaceri sau fluxurile de lucru.

# 2.1.1 Imaginea generală asupra sistemului

O imagine generală a unui sistem include de obicei descrierea componentelor sistemului și modul în care acestea interacționează între ele. Aceasta poate include, de exemplu, componente hardware și software, fluxuri de date, procese și protocoale de comunicație.

Pentru a dezvolta o imagine generală completă a unui sistem, este important să înțelegem scopul și obiectivele sistemului, precum și cerințele și restricțiile impuse asupra acestuia. De asemenea, este important să evaluăm performanța sistemului, capacitatea de extensibilitate și scalabilitate, precum și riscurile și vulnerabilitățile sistemului.

Diagrama de cazuri de utilizare (Use Case Diagram) este o diagramă UML (Unified Modeling Language) utilizată pentru a descrie interacțiunile între un sistem și actorii săi (utilizatori, alte sisteme, componente hardware etc.). Aceasta este una dintre cele mai utilizate și cunoscute diagrame UML și este utilizată pentru a prezenta modul în care utilizatorii sau alte sisteme interacționează cu sistemul.

Într-o diagramă de cazuri de utilizare, se reprezintă cazurile de utilizare (use cases), actorii și relațiile dintre acestea. Actorii sunt reprezentați ca fiind entități externe care interacționează cu sistemul. Cazurile de utilizare reprezintă acțiunile specifice pe care actorii le pot întreprinde în cadrul sistemului și sunt descrise în detaliu în documentația de cazuri de utilizare.

Diagrama de cazuri de utilizare este utilă pentru a ilustra modul în care utilizatorii și alte sisteme interacționează cu sistemul, astfel încât dezvoltatorii și designerii să poată înțelege și proiecta sisteme care satisfac nevoile utilizatorilor. Aceasta poate fi folosită și pentru a comunica cu utilizatorii și pentru a obține feedback-ul acestora cu privire la modul în care sistemul ar trebui să fie proiectat și dezvoltat.

În figura 4 ce urmează mai jos putem observa diagrama cazurilor de utilizare în interacțiunea unui utilizator cu sistemul nostru. Utilizatorul are proprietăți atât de vânzător cât și cumpărător. Utilizatorul are așa funcțional ca: Autentificare, Înregistrare, Resetarea parolei, Accessarea anunțurilor. Între sistem și utilizator se observă o relație de asociere dintre aceștia.

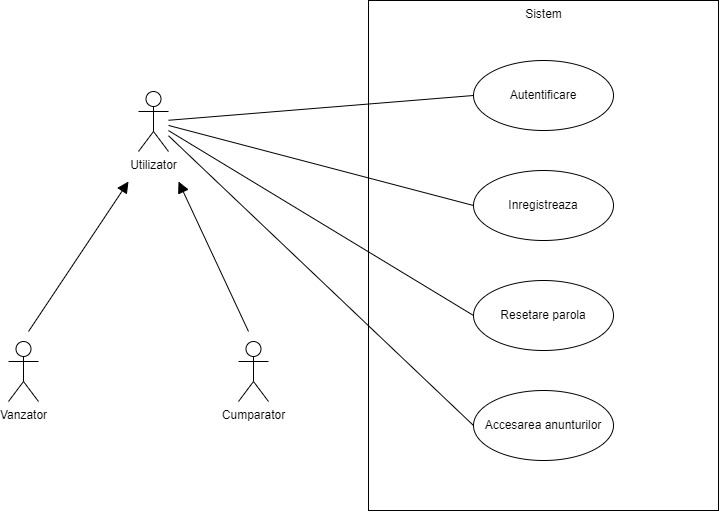


Figura 4 - Cazurile de utilizare în interacțiunea cu sistemul

În următoarea figură, figura 5 este reprezentat procesul de înregistrare a unui utilizator. Observăm procesul prin care user-ul poate să se înregistreze pentru a interacționa cu sistemul pe care l-am realizat. Administratorul este reprezentat ca un actor , împreună cu dezvoltatorul și serviciul de autentificare care face parte din aceleași componente. Prin intermediul relației *include* sunt reprezentate parțile componente a procesului de înregistrare în sistem și anume crearea unei parole, autentificarea prin browser-ul google, introducerea unei adrese de email valide. Crearea parolei, introducerea adresei și tastarea butonului finish, generează cazul de verificarea datelor valide .

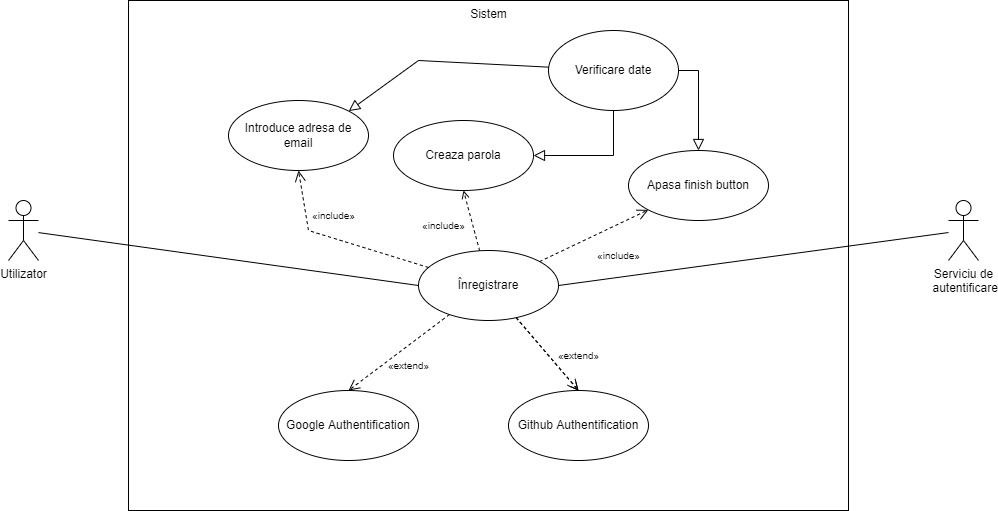


Figura 5 - Cazurile de utilizare pentru procesul de înregistrare a unui utilizator

# 2.1.2 Modelarea vizuală a fluxurilor

Modelarea vizuală a fluxurilor este importantă în dezvoltarea de software, deoarece ajută la îmbunătățirea comunicării și a înțelegerii între dezvoltatorii de software, manageri și utilizatori finali. De asemenea, poate fi folosită pentru a identifica și a elimina erorile și pentru a îmbunătăți eficiența și performanța sistemului.

Diagrama de activități (Activity Diagram) este o diagramă UML (Unified Modeling Language) utilizată pentru a reprezenta fluxurile de lucru și procesele dintr-un sistem sau proces de afaceri. Această diagramă poate fi utilizată pentru a modela activitățile și acțiunile necesare pentru a realiza un obiectiv în cadrul unui sistem sau proces și poate fi utilizată în diferite domenii, cum ar fi dezvoltarea de software, procese de afaceri, inginerie și altele.

Într-o diagramă de activități, activitățile sunt reprezentate ca blocuri de formă dreptunghiulară, iar deciziile sunt reprezentate ca romburi. Linii cu săgeți sunt utilizate pentru a conecta blocurile și pentru a indica ordinea activităților. Această diagramă poate fi folosită pentru a modela fluxurile de lucru și pentru a identifica și elimina eventualele probleme sau blocaje în procesul de afaceri sau dezvoltare de software.

Diagrama de activități poate fi utilizată pentru a comunica cu toți cei implicați într-un proiect, inclusiv cu dezvoltatorii, designerii, managerii și clienții. Aceasta poate fi utilă pentru a comunica planurile și obiectivele, pentru a identifica problemele și pentru a prezenta soluțiile posibile. De asemenea, poate fi folosită pentru a îmbunătăți procesele de afaceri și dezvoltarea de software, astfel încât acestea să fie mai eficiente și mai performante.

În următoarea figură (fig.6) este reprezentată diagrama de activitate care reprezintă nivelul de acces a aplicației. Prin această diagramă este modelat nivelul de specificare integrat în aplicație , care va delimita în două categorii accesul utilizatorilor la diverse categori ale sistemului. Va conține o categorie pentru utilizatorii care sunt înregistrați (care au access direct către funcționalitatea aplicației) și o altă categorie pentru utilizatorii neînregistrați (oaspeți) care vor putea vedea o landing page (o pagină unde vor putea vedea descrierea aplicației, descrierea funcționalului, design-ului ci nu și access direct la aplicație).

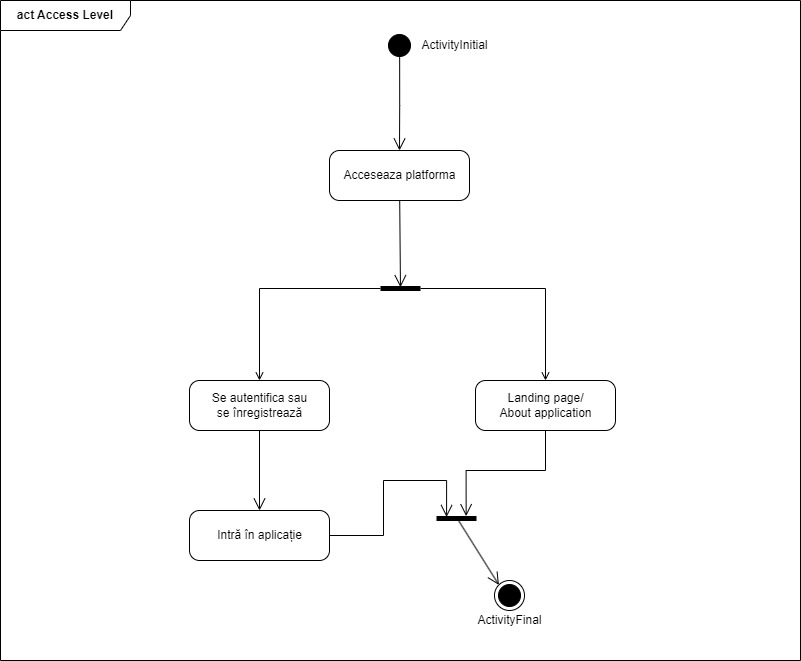


Figura 6 – Diagrama de activitate – Acess level

În următoarea diagramă (fig. 7) este reprezentată diagrama de activitate AddPost în care putem observa cum are loc adăugarea unui nou anunț în aplicație. Utilizatorului i se cere să introducă următoarele câmpuri – titlul, descrierea, suma/prețul produsului, o demo versiune sau video URL, categoria produsului software selectat.

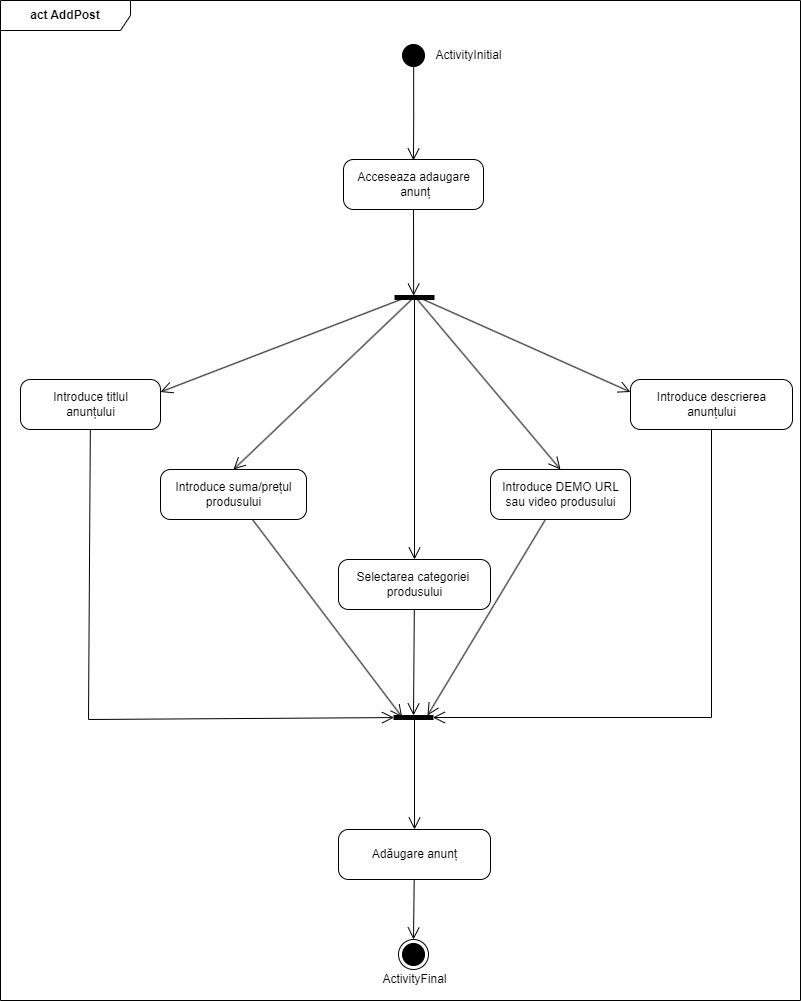


Figura 7 – Diagrama de activități – AddPost

# 2.1.3 Stările de tranzacție a sistemului

Diagrama de stări (Statechart Diagram) este o diagramă UML (Unified Modeling Language) utilizată pentru a modela comportamentul unui sistem sau a unui obiect în ceea ce privește schimbarea stării sale în funcție de evenimentele externe și a acțiunilor interne. Această diagramă poate fi utilizată pentru a modela comportamentul unui sistem sau a unui obiect și poate fi utilizată în diferite domenii, cum ar fi dezvoltarea de software, ingineria, sistemele automate și multe altele.

Într-o diagramă de stări, stările sunt reprezentate ca fiind ovale și sunt conectate prin tranziții care indică schimbarea de la o stare la alta. Tranzițiile sunt declanșate de evenimente și pot avea condiții de gard pentru a decide dacă tranziția trebuie să aibă loc sau nu. De asemenea, acțiunile pot fi atașate tranzițiilor pentru a indica acțiunile care trebuie să fie efectuate atunci când se face tranziția.

Diagrama de stări poate fi utilizată pentru a modela comportamentul unui sistem sau a unui obiect și pentru a identifica și elimina eventualele probleme sau blocaje în procesul de dezvoltare de software sau a sistemelor automate. Aceasta poate fi utilă pentru a comunica cu toți cei implicați într-un proiect, inclusiv cu dezvoltatorii, designerii, managerii și clienții. Aceasta poate fi, de asemenea, folosită pentru a verifica și valida modelele de comportament ale sistemului sau ale obiectelor, pentru a asigura că acestea funcționează conform specificațiilor și așteptărilor.

În următoarea diagramă (fig. 8) putem vedea o diagramă de stare „Autentificare”. În diagrama respectivă am reprezentat cum are loc procesul de autentificare. Utilizatorul introduce se autentifică (introducând login și password), cu acțiunea check, verificarea datelor introduse.

Se verifică datele introduse, dacă sunt valide ne duce la final, dacă nu, ne afișează un mesaj că datele sunt greșite.

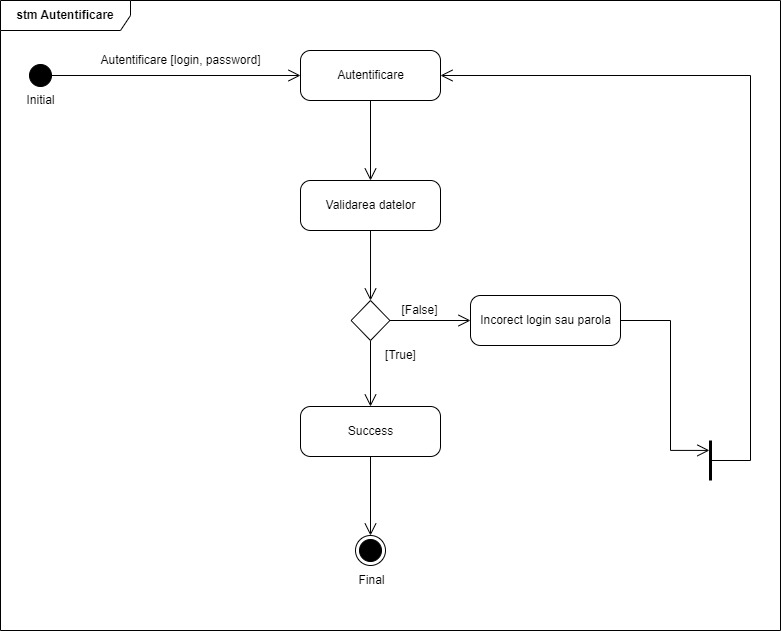


Figura 8 – Diagrama de stare – Autentificare

În diagrama ce urmează (fig. 9) putem observa o diagramă de stare – Înregistrare. În diagrama respectivă am reprezentat procesul de înregistrare în sistem. Acțiunea tranziției este Register având parametrii login, email, password, cu evenimentul check. În primul rând verificăm dacă este disponibil login-ul introdus. Dacă nu afișăm un mesaj respunzător și ne întoarcem la register, dacă da mergem la următoarea etapă, verificăm dacă email-ul este disponibil, aceeași procedură, apoi introducem parola și deja creăm însuși account-ul.

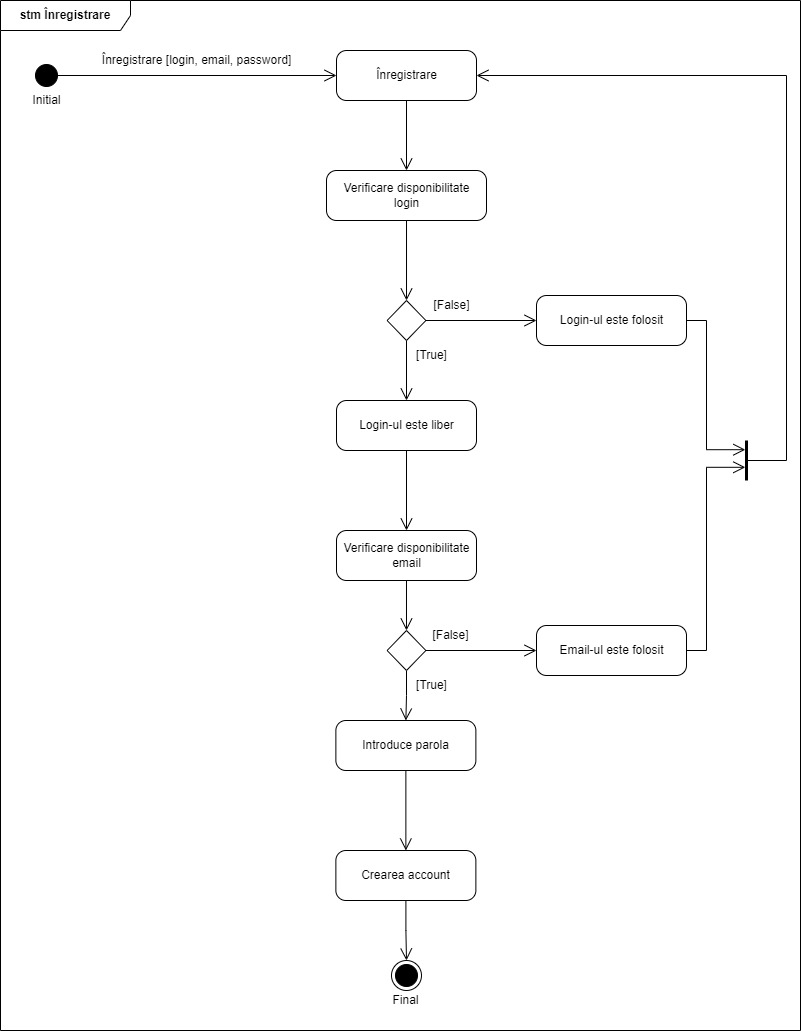


Figura 8 – Diagrama de stare – Înregistrare

# 2.1.4 Descrierea scenariilor de utilizare a aplicaţiei

Diagrama de secvențe (Sequence Diagram) este o diagramă UML (Unified Modeling Language) utilizată pentru a modela interacțiunile între obiecte sau componente dintr-un sistem. Această diagramă poate fi utilizată pentru a modela scenarii de utilizare, interacțiuni între obiecte și secvențe de evenimente care au loc în cadrul unui sistem sau aplicație.

Într-o diagramă de secvențe, obiectele sau componente sunt reprezentate ca fiind cadre rectangulare sau pătrate, iar evenimentele sau mesajele trimise între acestea sunt reprezentate ca săgeți care conectează obiectele. Pe măsură ce interacțiunile între obiecte sau componente au loc, acestea sunt reprezentate în ordinea cronologică a evenimentelor.

Diagrama de secvențe poate fi folosită pentru a comunica cu toți cei implicați într-un proiect, inclusiv cu dezvoltatorii, designerii, managerii și clienții. Aceasta poate fi utilă pentru a comunica planurile și obiectivele, pentru a identifica problemele și pentru a prezenta soluțiile posibile. De asemenea, poate fi utilizată pentru a valida modelele de comportament ale sistemului sau ale obiectelor și pentru a verifica dacă acestea funcționează conform specificațiilor și așteptărilor. Această diagramă poate fi, de asemenea, folosită pentru a dezvolta și testa scenarii de utilizare pentru un sistem sau aplicație.

În figura 9 putem observa o diagramă de secvență pentru autentificare. Putem vizualiza un actor (USER) și 2 obiecte (LOGIN și DB) care intracționează între ele. Pentru început eu fac un request către login cu datele introduse de utilizator (login și parola) așteptând să primească drept răspuns validarea acestora. LOGIN va crea o cerere la DB verificând astfel datele introduse dacă sunt veridice, se va returna mesaj de success sau de eroare și ulterior redirectăm unde avem nevoie. În cazul datelor incorecte dacă utilizatorul apasă forgot password atunci el trebuie să introducă email-ul pentru a reseta parola. După ce a introdus email-ul iarăși facem referire la baza de date (DB) și verificăm dacă email-ul există. Dacă există returnăm funcția SendMsgToEmail(), accesând linkul de pe email utilizatorului i se va propune să introducă parola nouă, după introducerea parolei noi se salvează în baza de date (DB) și se returnează un mesaj de success.

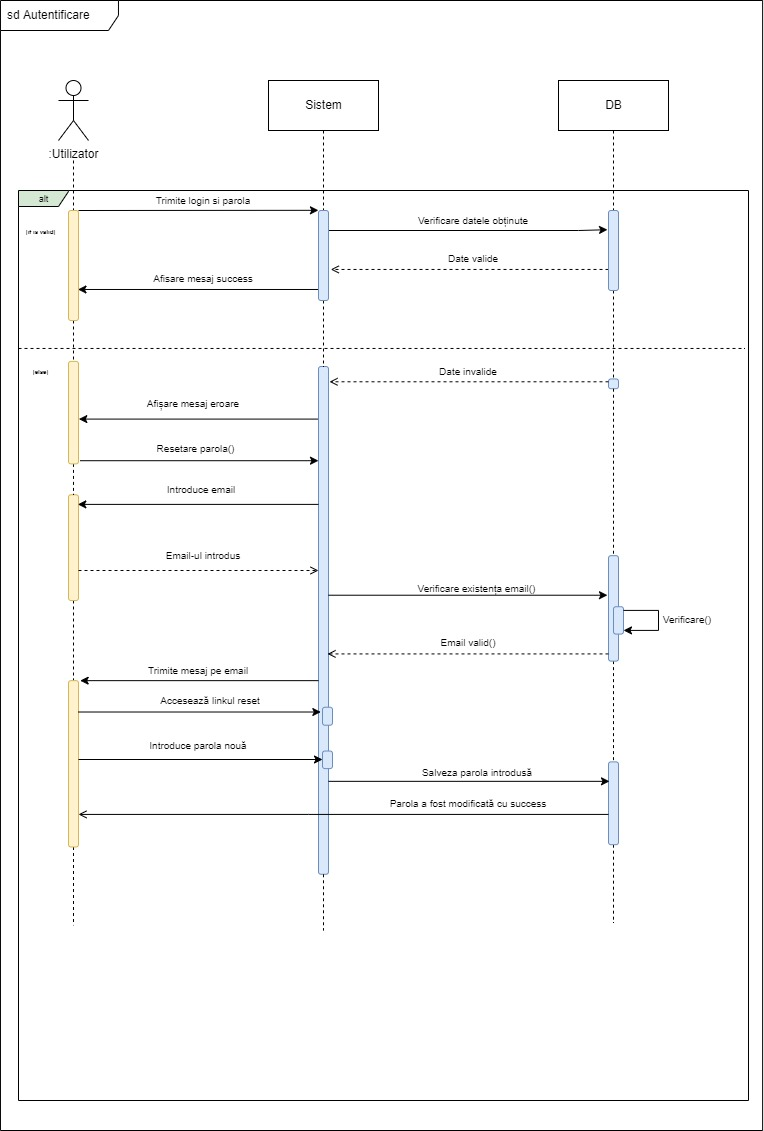


Figura 9 – Diagrama de secvență pentru procesul de autentificare

În următoare figura – 10, putem observa o diagramă de secvență pentru înregistrarea utilizatorului. Putem iarăși vizualiza un actor (USER) și 2 obiecte (Register și DB) care interacționează între ele. Pentru început introducem username-ul, ulterior verificăm cerințele pentru acel username (nr. de caractere, tipuri de simboluri ș.a.m.d.), apoi trimitem validarea către DB, verificăm dacă numele de utilizator nu este ocupat și ulterior dacă nu este întoarcem mesaj că este disponibil. După verificarea username-ului introducem parola cu ulterioara verificarea cerințelor de parolă. Dacă este valid introducem email-ul, apoi verificăm cerințele acestuia. După introducerea email-ului valid ne se propune să introducem captcha, trecem captcha și apoi trimitem datele către DB cu verificarea repetată a certitudinii datelor, apoi salvăm datele respective și introducerea mesajului și facem redirect către pagina necesară.

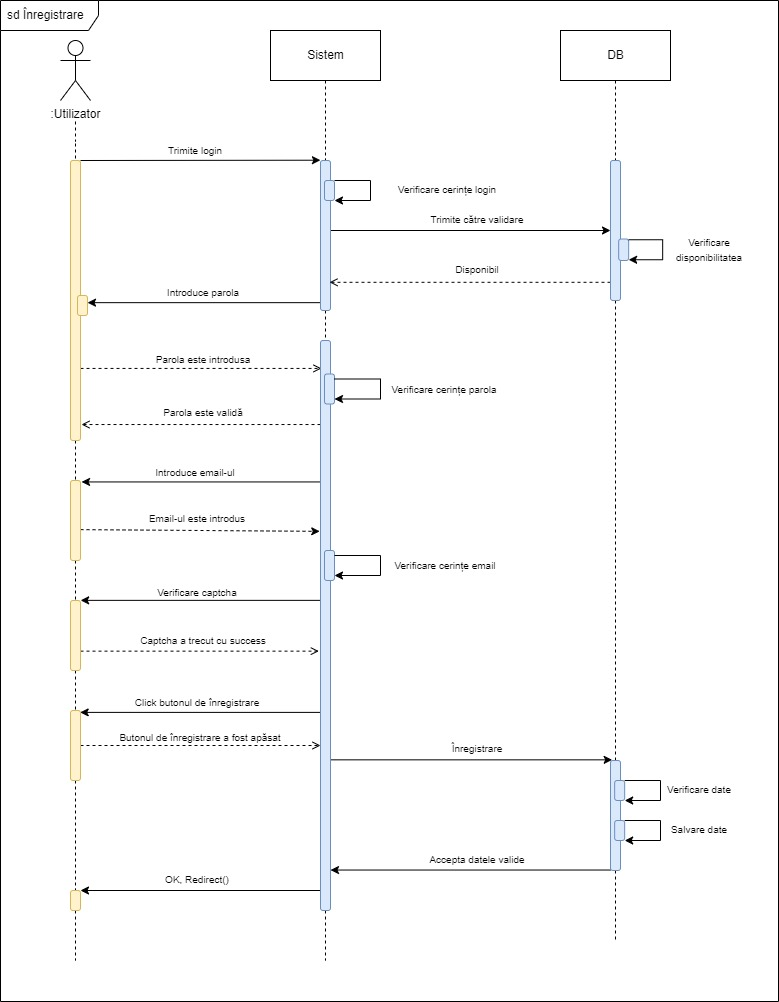


Figura 10 – Diagrama de secvență pentru procesul de înregistrare (Register)

# 2.1.5 Fluxurile de mesaje şi legăturile dintre componentele sistemului

Diagrama de colaborare (Collaboration Diagram) este o diagramă UML (Unified Modeling Language) utilizată pentru a modela interacțiunile între obiecte sau componente dintr-un sistem sau aplicație. Această diagramă poate fi utilizată pentru a modela scenarii de utilizare, interacțiuni între obiecte și fluxul de informații sau mesaje care circulă între acestea.

Într-o diagramă de colaborare, obiectele sau componente sunt reprezentate ca fiind cadre rectangulare sau pătrate, iar mesajele sau informațiile sunt reprezentate ca săgeți care conectează obiectele. Pe măsură ce interacțiunile între obiecte sau componente au loc, acestea sunt reprezentate în ordinea în care se întâmplă, astfel încât fluxul de informații sau mesaje să fie ușor de urmărit.

Diagrama de colaborare poate fi utilă pentru a comunica cu toți cei implicați într-un proiect, inclusiv cu dezvoltatorii, designerii, managerii și clienții. Aceasta poate fi utilizată pentru a identifica și clarifica interacțiunile dintre obiecte sau componente, pentru a valida modelele de comportament ale sistemului sau ale obiectelor și pentru a verifica dacă acestea funcționează conform specificațiilor și așteptărilor. De asemenea, poate fi folosită pentru a dezvolta și testa scenarii de utilizare pentru un sistem sau aplicație.

În figura 9 este reprezentat diagrama de colaborare - Access level care reprezintă nivelul de acces a aplicației. Prin această diagramă este modelat nivelul de specificare integrat în aplicație , care va delimita în două categorii accesul utilizatorilor la diverse categori ale sistemului. Va conține o categorie pentru utilizatorii care sunt înregistrați (care au access direct către funcționalitatea aplicației) și o altă categorie pentru utilizatorii neînregistrați (oaspeți) care vor putea vedea o landing page (o pagină unde vor putea vedea descrierea aplicației, descrierea funcționalului, design-ului ci nu și access direct la aplicație).

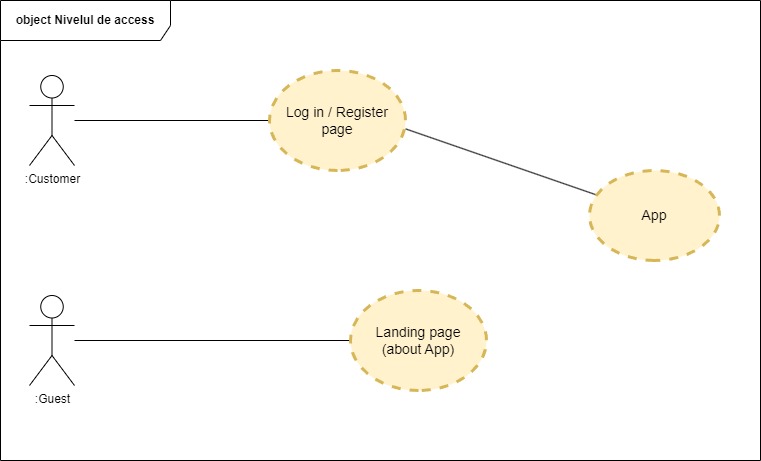


Figura 9 – Access level (nivel de specificare)

În figura 11 este reprezentată diagrama de colaborare nivel de exemplu, pentru procesul de logare. Actorul (sau utilizatorul) introduce login-ul și parola (1), apoi apasă butonul Login (1.1). După aceasta datele introduse (login-ul și parola) se verfică în baza de date (1.2), dacă totul este fain atunci chemăm ok (1.3) după ce redirecționăm către aplicație (1.4). Dacă utilizatorul va apăsa ResetPassword (2) el va trebui să introducă email-ul (2.1) după ce email-ul introdus se va verifica la veridicitate (2.2) după se va verifica dacă în baza de date există așa email (2.3), dacă există întoarcem că email-ul este valid (2.4) și trimitem un mesaj pe email-ul respectiv cu link-ul la resetare (2.5). Utilizatorul accesează acest link (2.6) și introduce parola nouă (2.7). După ce a introdus parola nouă (2.8) aceasta se salvează (2.9) și chemăm funcția password changed (3) ulterior redirecționând utilizatorul către aplicație (3.1).

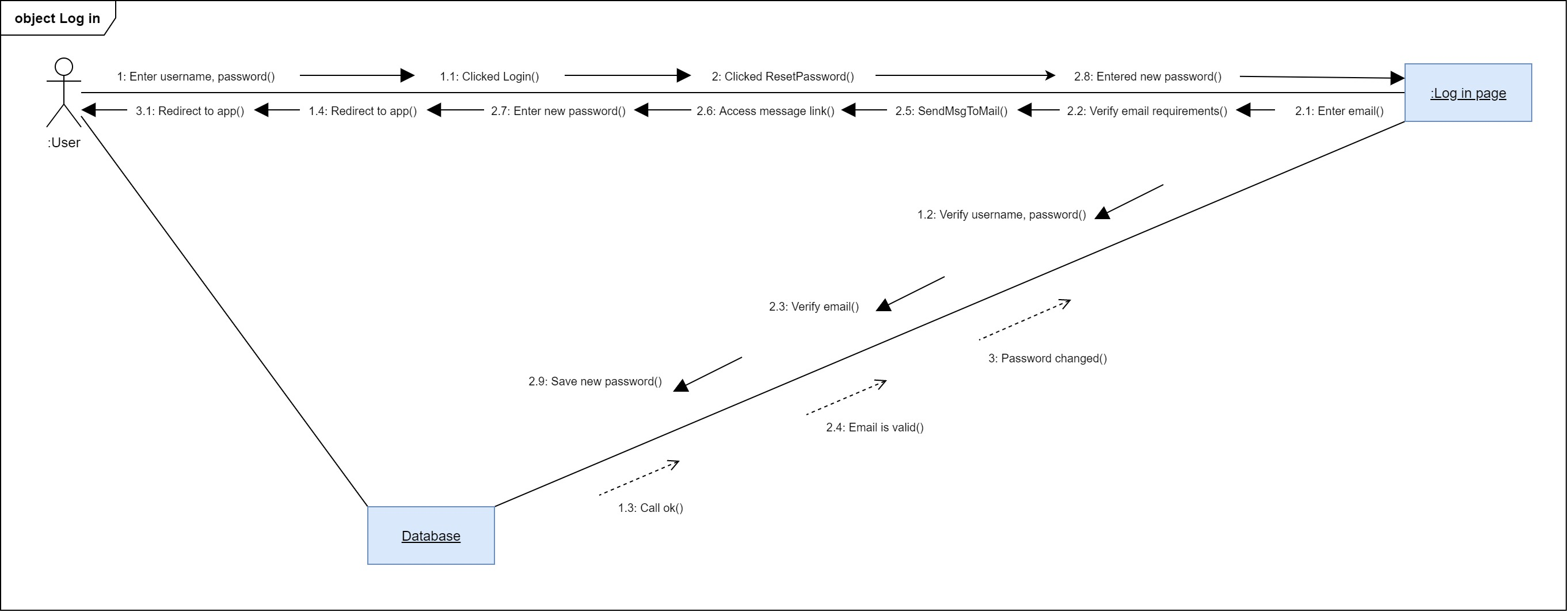


Figura 11 – Diagrama de colaborare la procesul de logare

# 2.2 Descrierea structurală a sistemului

Descrierea structurală a unui sistem reprezintă o analiză detaliată a componentelor sale și a modului în care acestea sunt organizate și interconectate între ele. Această descriere structurală poate fi realizată utilizând diferite modele și diagrame, cum ar fi diagrama de clasă, diagrama de pachet, diagrama de componentă, diagrama de distribuție etc.

Diagrama de clasă este unul dintre cele mai utilizate modele de descriere structurală a unui sistem, fiind utilizată pentru a reprezenta clasele și relațiile dintre acestea în cadrul sistemului. Aceasta poate fi folosită pentru a descrie componentele principale ale sistemului și modul în care acestea interacționează.

Diagrama de pachet este un alt model de descriere structurală utilizat pentru a organiza clasele în grupuri logice sau pachete. Aceasta poate fi folosită pentru a gestiona complexitatea sistemului prin organizarea componentelor într-un mod mai ușor de înțeles.

Diagrama de componentă este utilizată pentru a reprezenta componentele și interconexiunile acestora în cadrul unui sistem, în timp ce diagrama de distribuție este utilizată pentru a modela modul în care componentele sunt distribuite pe diferite noduri sau dispozitive fizice într-o rețea sau arhitectură distribuită.

Toate aceste modele și diagrame sunt utilizate pentru a descrie structura sistemului și pentru a ajuta la înțelegerea detaliată a componentelor sale, precum și a modului în care acestea interacționează și funcționează împreună.

În următoarea figură este prezentată diagrama de context de interacțiune cu postările. Putem observa nodul principal Posts (sau postări) și interacțiunea diferitor entități – Seller, Buyer, Admin, Moderator cu aceasta. Seller-ul sau vânzătorul poate vizualiza, crea, edita sau șterge anunțurile, după care el așteaptă un răspuns de la moderator sau admin care verifică postarea și-i dă un verdict. Buyer-ul sau cumpărătorul poate doar vizualiza postările adăugate de vânzător. Moderatorul poate vizualiza, verifica și modera postarea (approve or decline). Administratorul are acelea-și drepturi ca și Moderatorul, Buyer-ul și Seller-ul toți împreună. El poate modifica, șterge sau modera o postare.

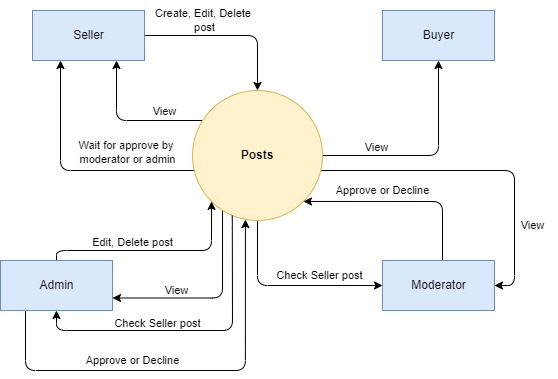


Figura 12 – Diagrama de context

# 2.2.1 Descrierea structurii statice a sistemului

Diagrama de clasă (Class Diagram) este unul dintre cele mai utilizate modele de descriere structurală în cadrul UML (Unified Modeling Language), utilizat pentru a reprezenta clasele, atributele, metodele și relațiile dintre acestea în cadrul unui sistem sau aplicație.

Diagrama de clasă este utilizată pentru a modela structura statică a unui sistem și poate fi utilizată pentru a descrie componentele principale ale sistemului și modul în care acestea interacționează. În cadrul unei diagrame de clasă, fiecare clasă este reprezentată sub formă de dreptunghi, iar atributele și metodele sale sunt reprezentate ca și câmpuri și metode asociate cu acea clasă.

Relațiile dintre clase sunt reprezentate prin săgeți, care indică tipul de relație, cum ar fi relația de asociere, relația de generalizare, relația de compoziție sau relația de agregare.

Diagrama de clasă poate fi utilizată pentru a modela sisteme complexe și pentru a ajuta la înțelegerea detaliată a componentelor sale, precum și a modului în care acestea interacționează și funcționează împreună. De asemenea, diagrama de clasă poate fi utilizată pentru a comunica cu dezvoltatorii, designerii, managerii și clienții pentru a valida modelele de structură ale sistemului și pentru a verifica dacă acestea funcționează conform specificațiilor și așteptărilor.

În figura 12 este reprezentat diagrama de clasă Nivelul de access care reprezintă nivelul de acces a aplicației. În figura respectivă putem observa 3 clase Guest, User și Admin care fac moștenire de la clasa Account. Guest (sau oaspete) se poate înregistra și autentifica, User (sau utilizator) se poate loga, schimba parola, restabili parola. De asemenea are unele atribute private cum ar fi: username, password, ip, auth\_date. Admin – este o clasă specială destinată administratorului. El de asemenea are username, password, ip, auth\_date care sunt protected dar pe lângă asta mai are access la modul privilegiat. În acest mod privilegiat el poate șterge un utilizator, poate edita, verifica activitatea utilizatorilor, actualiza aplicația, actualiza changelog. Observăm că are loc relația de compoziție dintre account și aplicație și admin-modul privilegiat și relația de moștenire dintre Guest, User, Admin și Account.

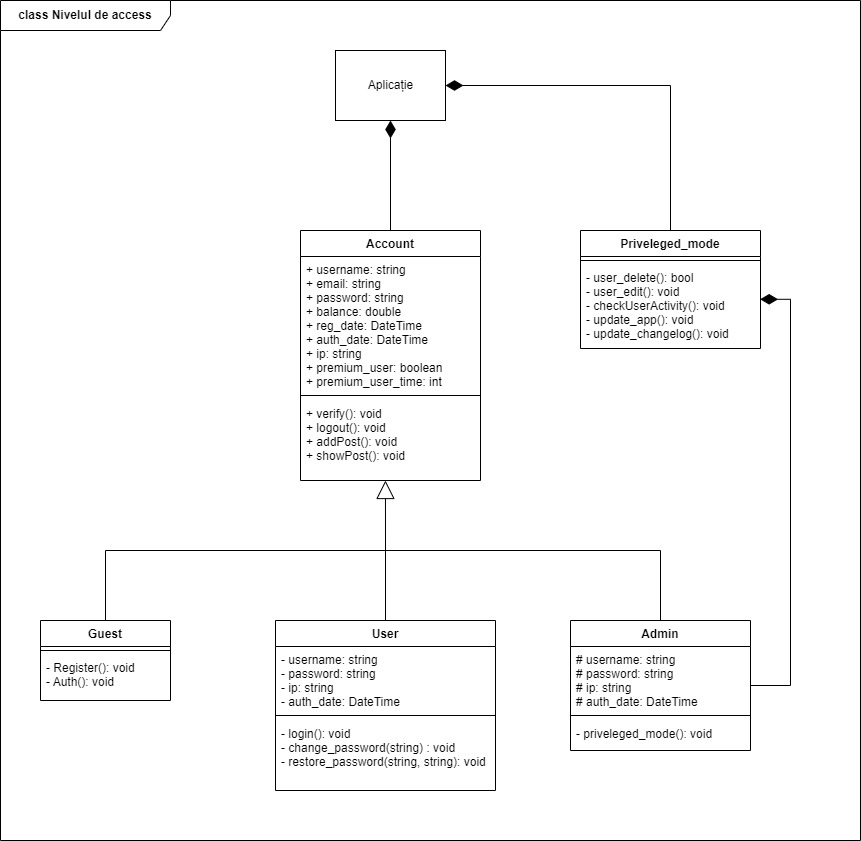


Figura 12 – Diagrama de clasă Nivele de access la aplicație

În figura ce urmează – 13 este reprezentat diagrama de clasa unde putem observa două clase – Post și AddPost. Putem observa relația de dependență cu nivel de access. Fiecare postare a un id, autor, nume, descriere, preț, când a fost creată și când a fost modificată. De asemenea are 3 funcții, show, edit și delete. AddPost este clasa care ne permite să adăugăm o postare. Utilizatorii pot adăuga postări care de asemenea au nume, descriere, preț, și data, putem de asemenea observa constructorul addPost care are 4 parametri de intrare (string, double, int, string).

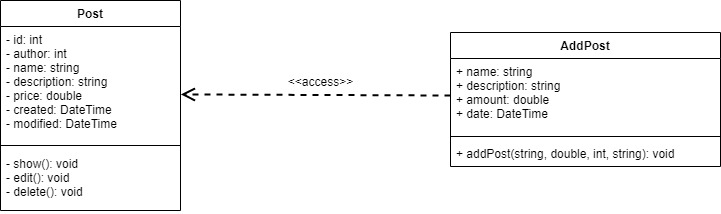


Figura 13 – Diagrama de clasa (Post)

După cum observăm din figura 14, avem 4 entități. Putem observa că User-ul se divizează în două tipuri (două sub-entități) – Buyer (Cumpărător) și Seller (Vânzător), care interacționează între ei. De asemenea putem observa că Cumpărătorul poate vizualiza, comenta, distribui postări, iar Vânzâtorul poate crea, edita, vizualiza, comenta și distribui. Deci Vânzătorul are funcțional asemănător cu a Cumpărătorului, cu ceva adițional.

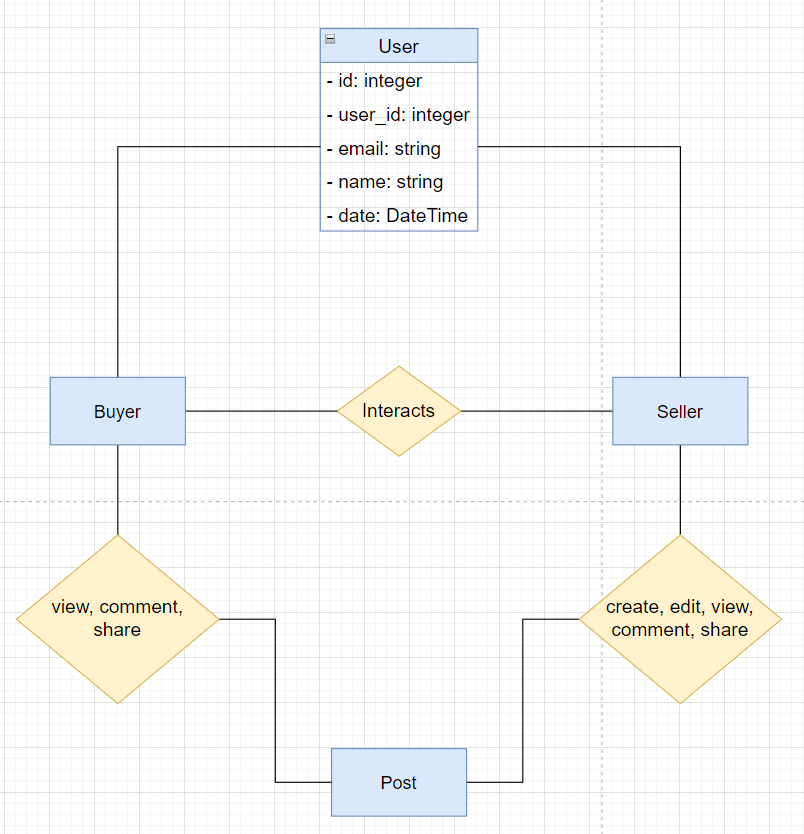


Figura 14 – Diagrama de clasa

# 2.2.2 Relatiile de dependență între componentele sistemului

Diagrama de componentă (Component Diagram) este un model utilizat în cadrul UML (Unified Modeling Language) pentru a reprezenta componente, interfețe, dependențe și relațiile dintre acestea în cadrul unui sistem sau aplicație.

Diagrama de componentă este utilizată pentru a modela structura fizică a unui sistem și poate fi utilizată pentru a descrie componentele principale ale sistemului și modul în care acestea interacționează. În cadrul unei diagrame de componentă, fiecare componentă este reprezentată sub formă de dreptunghi, iar interfețele sale sunt reprezentate sub formă de cercuri asociate cu acea componentă.

Relațiile dintre componente sunt reprezentate prin săgeți, care indică tipul de relație, cum ar fi relația de dependență, relația de asociere sau relația de compoziție.

Diagrama de componentă poate fi utilizată pentru a modela sisteme complexe și pentru a ajuta la înțelegerea detaliată a componentelor sale, precum și a modului în care acestea interacționează și funcționează împreună. De asemenea, diagrama de componentă poate fi utilizată pentru a comunica cu dezvoltatorii, designerii, managerii și clienții pentru a valida modelele de structură ale sistemului și pentru a verifica dacă acestea funcționează conform specificațiilor și așteptărilor.

În următoarea diagramă (fig. 14) putem vedea o diagramă de component *„Login / Register”.* În diagrama respectivă am reprezentat cum are loc procesul de autentificare și înregistrare. Puteți observa interacțiunea dintre componente. Login/Register page face referire la User Authentification unde este obligatoriu de verificat credentials (datele introduse), apoi User Authentification are nevoie obligatoriude users table, tabelul users din baza noastră de date. Apoi se formează user session și se accesează personal cabinet, care apoi face referință la content management system (infrastructura unde se este content-ul utilizatorului), care deja face referință la alte tabele din baza de date. Cu înregistrarea practic aceeași, face referință la User Creation care creeză însuși utilizatorul, și după aia e aceeași ca și la autentificare (access la users table din db, crearea sessiei, access la personal cabinet ș.a.m.d)

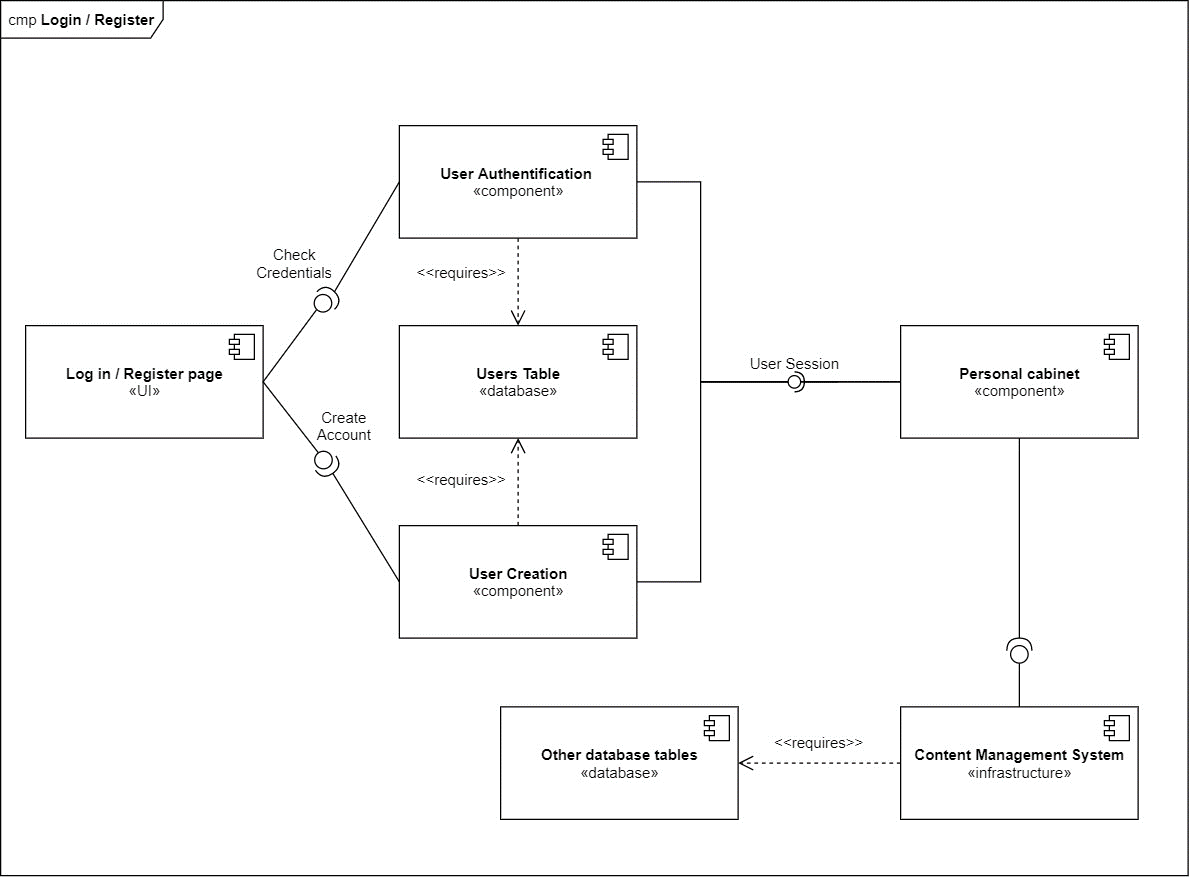


Figura 14 – Diagramă de component – Login / Register

În următoarea diagramă (fig. 15) putem vedea o diagramă de comportament „Posts”. În diagrama respectivă am reprezentat interacțiunea componentelor care fac referire la funcționalul de tranzacții. Putem vedea componentul Transactions, care face legătură cu baza de date MongoDB folosind stereotipul uses. De asemenea este obligatoriu ca să fie accesibil componentul Account care are informația utilizatorului. Posts are la rândul său 4 componente care nu pot lucra fără a fi conectate la Posts: AddPost (adăugarea unui anunț), GetPosts (afișarea tuturor anunțurilor), PostInfo (afișarea informației despre un anunț anumit), RemovePost (ștergerea unui anunț din baza de date).

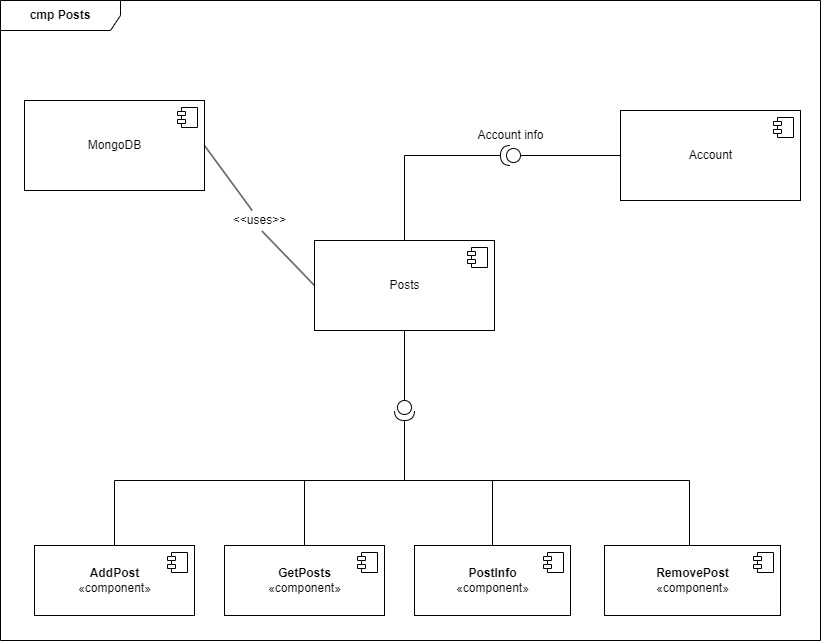


Figura 15 – Diagrama de component - Posts

# 2.2.3 Modelarea echipamentelor mediului de implementare

Diagrama de implementare (Deployment Diagram) este un model utilizat în cadrul UML (Unified Modeling Language) pentru a reprezenta modul în care componente fizice sau logice ale unui sistem sunt implementate și conectate într-un mediu de execuție.

Diagrama de implementare poate fi utilizată pentru a modela infrastructura de hardware și software necesară pentru a rula aplicația sau sistemul, cum ar fi servere, dispozitive de stocare, rețele și dispozitive periferice. În cadrul unei diagrame de implementare, componente precum hardware-ul și software-ul sunt reprezentate prin noduri și legăturile dintre acestea indică modul în care acestea sunt conectate și comunica între ele.

Prin utilizarea diagramelor de implementare, dezvoltatorii și arhitecții de sistem pot planifica și organiza modul în care componentele fizice și logice trebuie implementate și configurate pentru a asigura performanța, scalabilitatea și securitatea sistemului sau aplicației. De asemenea, diagramele de implementare pot fi folosite pentru a comunica cu echipele de dezvoltare, furnizorii de echipamente și clienții pentru a valida modelele de implementare ale sistemului și pentru a verifica dacă acestea funcționează conform specificațiilor și așteptărilor.

În următoarea diagramă (fig. 16) putem observa diagrama de implementare a sistemului. În diagrama dată am reprezentat interacțiunea dintre client «device» :UserClient (care are nodul web browser) și server «device» :HostMachine utilizând protocolul HTTP. În :HostMachine de asemenea putem vedea două noduri «execution environment» :WebServer utilizând tehnologia Node.JS, care va folosi framework-ul express.js (care are la rândul său application logic, express middlewares și route handler) și :DBServer folosind baza de date MongoDB. Legătura dintre WebServer și DBServer o facem prin MongoDB Node.JS driver.

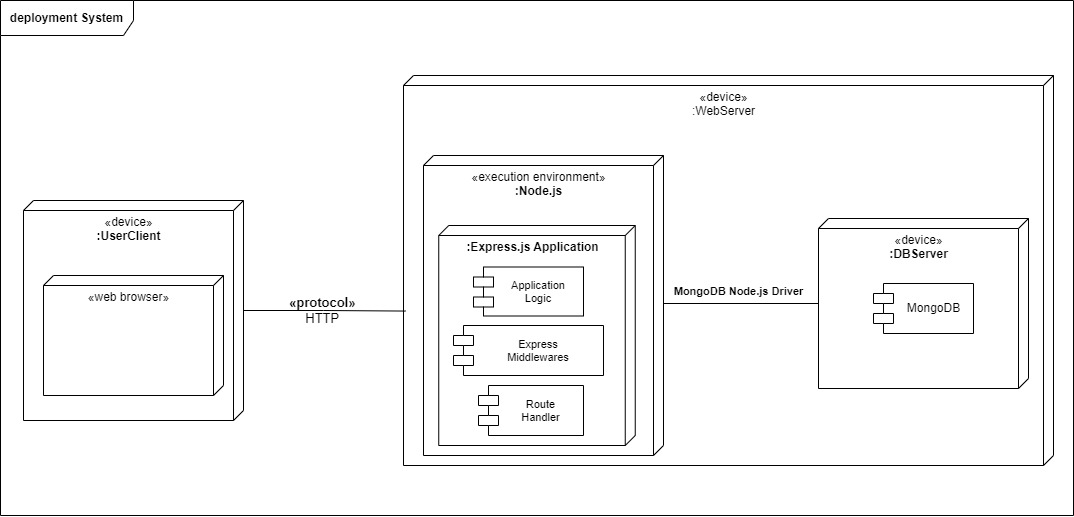


Figura 16 – Deplyment diagram - System

# 3 REALIZAREA SISTEMULUI

În lucrarea respectivă am implementat un șir de tehnologii. Platforma mea este implementată de la zero, a fost scris front-end-ul și backend-ul în limbajul de programare JavaScript (node.js).

Există mai multe motive pentru care Node.js este o soluție bună pentru dezvoltarea aplicațiilor web. Iată câteva dintre acestea:

1. Performanță ridicată: Node.js utilizează un model non-blocking de I/O, ceea ce înseamnă că poate procesa mai multe cereri simultan și poate gestiona un volum mare de trafic. Acest lucru face ca aplicațiile web dezvoltate cu Node.js să fie mai rapide și mai scalabile.
2. Utilizarea aceluiași limbaj de programare pe server și client: Node.js utilizează JavaScript ca limbaj de programare, ceea ce înseamnă că dezvoltatorii pot utiliza același limbaj pe server și client. Acest lucru face ca dezvoltarea aplicațiilor web să fie mai ușoară și mai eficientă.
3. Disponibilitatea unui ecosistem bogat de module și biblioteci: Node.js are un ecosistem bogat de module și biblioteci care fac posibilă dezvoltarea rapidă și eficientă a aplicațiilor web. Există module pentru aproape orice sarcină, de la gestionarea sesiunilor la autentificarea utilizatorilor și comunicarea cu bazele de date.
4. Comunitatea activă de dezvoltatori: Node.js are o comunitate activă și în creștere de dezvoltatori, care contribuie la dezvoltarea și îmbunătățirea continuă a tehnologiei. Această comunitate este un avantaj important pentru dezvoltatorii care doresc să învețe și să dezvolte aplicații cu Node.js.

Deci pe frontend a fost utilizat framework-ul Vue.JS de versiunea 3, iar pe backend Express.JS. De asemenea am utilizat MongoDB în calitate de bază de date.

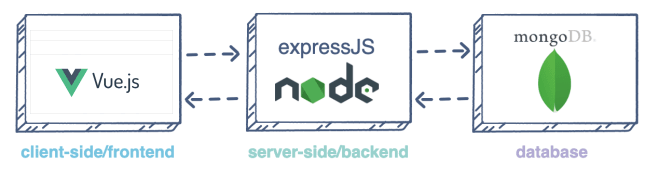


Figura 18 – Structura platformei create

**3.1 Vue.JS - Front end web framework**

Vue este un framework JavaScript proiectat pentru construirea de interfețe de utilizator reactive și eficiente. A fost creat de Evan You și a fost lansat pentru prima dată în 2014. Acesta se bazează pe paradigmă de componentă, adică construirea interfeței de utilizator prin compunerea unor componente reutilizabile.

Vue.js oferă o sintaxă elegantă și ușor de înțeles, ceea ce îl face foarte popular printre dezvoltatorii web. Are o curba de învățare relativ mică, ceea ce înseamnă că dezvoltatorii îl pot învăța rapid și îl pot folosi pentru a crea aplicații complexe cu ușurință.

O altă caracteristică importantă a Vue.js este reactivitatea sa. Datorită sistemului său de reactivitate, Vue.js poate detecta automat schimbările de stare și poate actualiza automat interfața de utilizator pentru a reflecta aceste schimbări. Această caracteristică face ca dezvoltarea cu Vue.js să fie foarte eficientă și productivă. Vue.js are numeroase avantaje care îl fac alegerea ideală pentru dezvoltarea aplicațiilor web moderne:

* Ușurința de utilizare: Vue.js este ușor de învățat și de utilizat, chiar și pentru dezvoltatorii începători. Sintaxa sa este simplă și intuitivă, ceea ce face ca codul să fie ușor de citit și de întreținut.
* Performanța ridicată: Vue.js oferă performanță ridicată prin intermediul sistemului său de reactivitate și prin utilizarea virtual DOM-ului. Acest lucru face ca aplicațiile să fie mai rapide și mai eficiente.
* Flexibilitatea: Vue.js este foarte flexibil, permițând dezvoltatorilor să îl utilizeze în diferite contexte și să îl integreze cu alte biblioteci sau cadre de lucru.
* Scalabilitatea: Vue.js este foarte scalabil, permițând dezvoltatorilor să creeze aplicații web de diferite dimensiuni și complexități.
* Comunitatea puternică: Vue.js are o comunitate puternică de dezvoltatori care oferă sprijin și resurse în mod constant. Există numeroase biblioteci și plugin-uri disponibile, care fac munca dezvoltatorilor mult mai ușoară și mai eficientă.
* Documentație excelentă: Vue.js are o documentație excelentă, care include exemple de cod, ghiduri și tutoriale. Aceasta face ca învățarea și utilizarea Vue.js să fie foarte ușoară.

În general, Vue.js este un framework excelent pentru construirea de interfețe de utilizator reactive și eficiente, cu o sintaxă ușor de înțeles și o comunitate puternică.

**3.1.1 Sintaxa șablonului**

Vue utilizează o sintaxă de șablon bazată pe HTML care vă permite să legați declarativ DOM-ul redat la datele instanței componentei de bază. Toate șabloanele Vue sunt HTML valide din punct de vedere sintactic, care pot fi analizate de browsere și analizoare HTML conforme cu specificațiile.

Sub capotă, Vue compilează șabloanele în cod JavaScript extrem de optimizat. În combinație cu sistemul de reactivitate, Vue își poate da seama în mod inteligent de numărul minim de componente de redat și de a aplica cantitatea minimă de manipulări DOM atunci când starea aplicației se schimbă.

**3.1.2 Interpolarea textului**

Eticheta mustață va fi înlocuită cu valoarea proprietății msg din instanța componentei corespunzătoare. De asemenea, va fi actualizat ori de câte ori msgproprietatea se schimbă.

Cea mai simplă formă de legare a datelor este interpolarea textului folosind sintaxa „Mustache” (acolade duble):

<template>

  <p class="greeting">

    Hello, {{ user.firstName }}, how are you today?

  </p>

</template>

<script setup>

const user = {

  firstName: 'Dan',

  lastName: 'Rotaru'

}

</script>

Figura 19 – Interpolarea textului în Vue.JS

**3.1.3 ​Composition API**

Composition API este o nouă caracteristică introdusă în Vue.js 3.0, care își propune să rezolve problemele cu care dezvoltatorii se confruntă în gestionarea codului pe măsură ce aplicațiile cresc în dimensiune și complexitate. Acesta oferă o modalitate mai modulară și mai flexibilă de a scrie codul în aplicațiile Vue.js.

În comparație cu Options API, Composition API oferă o modalitate mai structurată și mai organizată de a scrie codul. Cu Options API, codul este împărțit în proprietăți și metode, ceea ce poate duce la o structură dezordonată și confuză pe măsură ce aplicația crește în dimensiune. Composition API împarte codul în funcții, numite componenți, care pot fi reutilizate și combinate pentru a obține comportamente complexe. Composition API oferă, de asemenea, o serie de caracteristici utile, cum ar fi ref-uri, care permit accesul la obiecte și proprietăți din alte componente, și provide / inject, care oferă o modalitate mai eficientă de a partaja informații între componente.

În figura de mai jos (fig. 20) am reprezentat diferența dintre option API (vue 2) și Composition Api (vue 3). După cum puteți observa codul reprezentat din partea dreaptă (Composition API) este mai mic, mai citabil și mai bine înțeles decât codul din partea stângă.

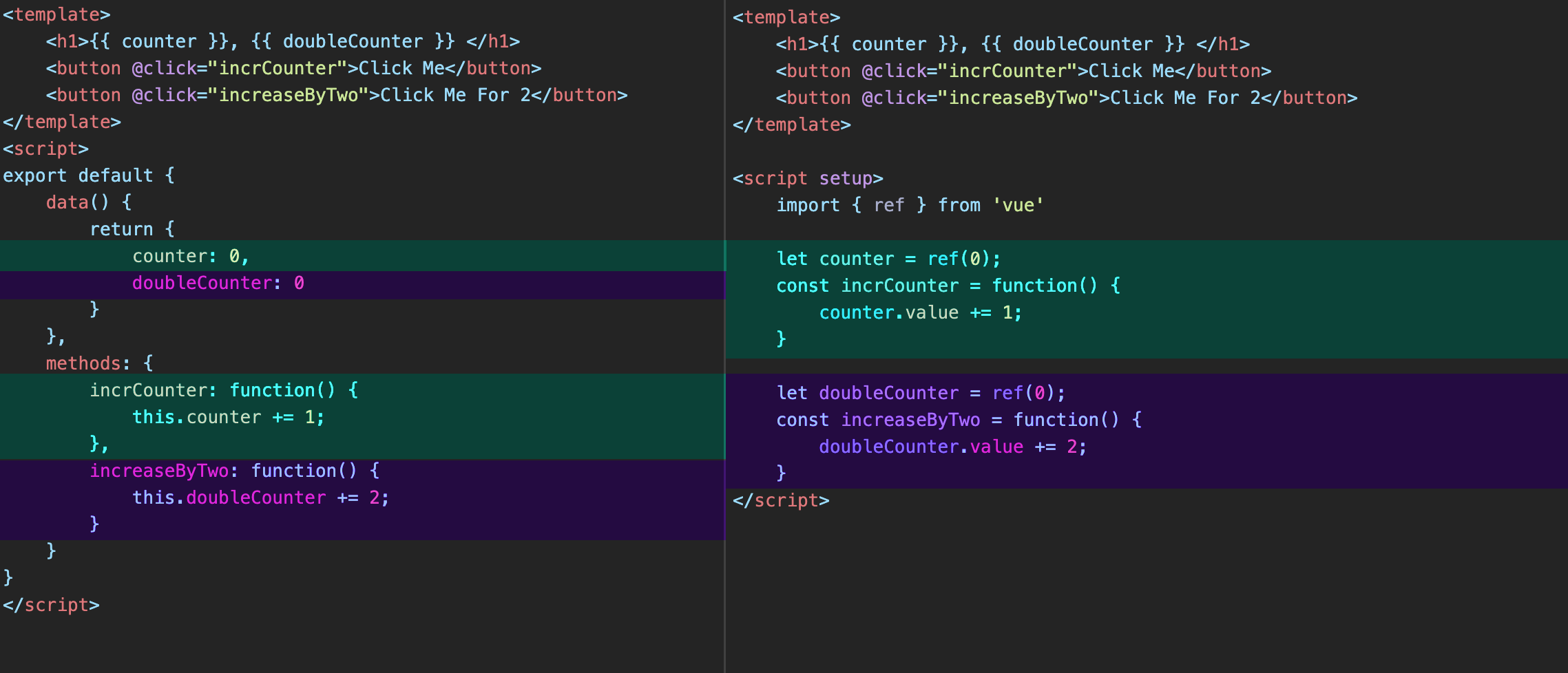


Figura 20 – Option API (vue 2) vs Composition API (vue 3)

Comparativ cu Option API, Composition API vine cu multe avantaje, care sunt:

1. **Reutilizarea codului**: Composition API vă permite să definiți logica componentelor în funcții și să o reutilizați în alte componente.
2. **Scalabilitate**: Composition API este mai scalabil decât Option API, deoarece vă permite să organizați mai bine codul și să separați logica în funcții mai mici și mai ușor de gestionat.
3. **Testarea mai ușoară**: Composition API facilitează testarea, deoarece logica componentelor este împărțită în funcții, ceea ce face mai ușoară izolarea logicii în timpul testării.
4. **Îmbunătățirea performanței**: Composition API poate ajuta la îmbunătățirea performanței aplicației, deoarece permite optimizarea reprezentării grafice.
5. **Cod mai curat**: Composition API vă permite să scrieți cod mai curat și mai ușor de citit, deoarece componentele pot fi organizate într-un mod logic și structurat.

În concluzie, Composition API este o modalitate mai bună de a scrie cod în Vue.js decât Option API, datorită avantajelor pe care le aduce în termeni de reutilizare a codului, scalabilitate, testare, performanță și claritate. Composition API face ca dezvoltarea aplicațiilor Vue.js să fie mai modulară, mai flexibilă și mai ușor de gestionat pe măsură ce aplicațiile devin mai complexe.

**3.1.4 Componente – caracteristica de bază a framework-ului**

În Vue totul este divizat pe componente, fiecare component își are o logică oarecare, un funcțional anumit. Spre exemplu să luăm componentul Rating creat în aplicația mea:

<template>

  <div class="rating">

    <ul class="list-inline mb-0">

      <li v-for="starClass in starClasses" :key="starClass" class="list-inline-item me-0 small">

        <i :class="starClass"></i>

      </li>

      <li class="list-inline-item ms-2 h6 fw-medium mb-0">{{ stars }}/5.0</li>

    </ul>

  </div>

</template>

<script setup>

import {defineProps, computed, toRefs} from 'vue';

const props = defineProps({

  stars: {

    type: Number

  }

});

const { stars } = toRefs(props);

const starClasses = computed(() => {

  const classes = [];

  const filledStars = Math.floor(stars.value);

  for (let i = 0; i < filledStars; i++) {

    classes.push('fas fa-star text-warning');

  }

  if (stars.value % 1 !== 0) {

    classes.push('fas fa-star-half-alt text-warning');

  }

  const remainingStars = 5 - filledStars - (stars.value % 1 !== 0 ? 1 : 0);

  for (let i = 0; i < remainingStars; i++) {

    classes.push('far fa-star text-warning');

  }

  return classes;

});

</script>

Figura 20 – Structura unui component Vue - Rating

Aici puteți observa că componentul are 2 secțiuni principale:

1. Template – seсțiunea unde se scrie html layout a componentului
2. Script – secțiunea unde are loc însăși careva logică a componentului

De asemenea se poate de utilizat și Style – secțiunea în care stilizează elementele HTML

**3.1.5 Vuex – model de management a stărilor sistemului**

Vuex este o librărie de management al stării pentru aplicațiile Vue.js. Aceasta oferă o soluție centralizată pentru gestionarea stării în aplicațiile Vue.js, permițând programatorilor să își gestioneze cu ușurință datele și să păstreze starea aplicației într-un singur loc.

Vuex oferă un model de stocare bazat pe evenimente și utilizează un flux de date unidirecțional, ceea ce înseamnă că datele se deplasează într-o singură direcție prin aplicație. Acest model face ușor de înțeles cum funcționează datele și cum acestea se propagă prin componentele aplicației.

În ceea ce privește utilizarea Vuex cu Vue.js, programatorii pot defini starea aplicației lor într-un singur loc și pot accesa aceste date din orice componentă a aplicației. Acest lucru simplifică gestionarea stării și face mai ușoară dezvoltarea aplicațiilor complexe.

Cu Vuex, programatorii pot gestiona starea aplicației lor într-un mod predictibil și eficient, făcând astfel dezvoltarea aplicațiilor Vue.js mai rapidă și mai ușoară. Aceasta face ca Vuex să fie o soluție populară și larg utilizată pentru dezvoltarea de aplicații web bazate pe Vue.js.

Vuex ne ajută să ne ocupăm de managementul de stat partajat cu costul mai multor concepte și standard. Mai jos puteți observa un exemplu de vuex store aplicație.

import Vue from 'vue'

import Vuex from 'vuex'

Vue.use(Vuex)

const store = new Vuex.Store({

  state: {

*// Define your state here*

  },

  mutations: {

*// Define your mutations here*

  },

  actions: {

*// Define your actions here*

  },

  getters: {

*// Define your getters here*

  }

})

export default store

Figura 21 – Vuex – Definirea stărilor

În acest exemplu, este definit un store Vuex care conține trei proprietăți principale: state, mutations și actions, precum și getters.

* **State** reprezintă starea aplicației și conține variabilele de starea care vor fi stocate în store-ul Vuex.
* **Mutations** reprezintă funcțiile care modifică starea aplicației. Acestea sunt funcții purtătoare de nume care iau starea curentă și un set de argumente și modifică starea în funcție de acestea.
* **Actions** reprezintă funcțiile care inițiază modificarea stării aplicației prin mutații. Acestea sunt funcții care primesc o instanță a obiectului store și un set de argumente și apoi declanșează mutații corespunzătoare.
* **Getters** reprezintă funcțiile care accesează starea aplicației și returnează valori derivate din aceasta. Acestea sunt funcții purtătoare de nume care primesc starea curentă și returnează o valoare derivată din aceasta.

Acestea sunt elementele de bază ale unui store Vuex și pot fi extinse și personalizate în funcție de nevoile aplicației.

**3.1.6 Axios – interacțiunea frontend – backend prin trimiterea cererilor HTTP**

Axios este o librărie JavaScript folosită pentru a face cereri HTTP către serverul web și a obține date în format JSON sau alte formate. Aceasta este o librărie foarte populară și larg utilizată în cadrul aplicațiilor web moderne, în special pentru dezvoltarea aplicațiilor web bazate pe React și Vue.js.

Axios este foarte ușor de utilizat și oferă o interfață simplă pentru a face cereri HTTP și a manipula răspunsurile serverului. Aceasta oferă, de asemenea, funcționalitate pentru a configura și personaliza cererile, precum și a trata erorile de cerere și de răspuns.

Un alt avantaj al utilizării Axios este că aceasta este compatibilă cu majoritatea browserelor moderne și poate fi utilizată într-o gamă largă de medii de dezvoltare, inclusiv Node.js.

În ceea ce privește integrarea cu Express.js, Axios poate fi utilizată pentru a face cereri către serverul Express.js și a obține date de la acesta. Acest lucru face Axios o opțiune populară pentru a construi aplicații web complexe și performante care utilizează Express.js și alte tehnologii web moderne.

Mai jos este reprezentat un exemplu de cereri create prin intermediul bibliotecii Axios [5]:

const axios = require('axios');

*// Make a request for a user with a given ID*

axios.get('/user?ID=12345')

  .then(function (response) {

*// handle success*

    console.log(response);

  })

  .catch(function (error) {

*// handle error*

    console.log(error);

  })

  .finally(function () {

*// always executed*

  });

*// Optionally the request above could also be done as*

axios.get('/user', {

    params: {

      ID: 12345

    }

  })

  .then(function (response) {

    console.log(response);

  })

  .catch(function (error) {

    console.log(error);

  })

  .finally(function () {

*// always executed*

  });

*// Want to use async/await? Add the `async` keyword to your outer function/method.*

async function getUser() {

  try {

    const response = await axios.get('/user?ID=12345');

    console.log(response);

  } catch (error) {

    console.error(error);

  }

}

Figura 22 – Crearea cererilor HTTP utilizând librăria Axios

**3.2 Express.JS – Back end web framework**

Express.js este un framework back-end popular pentru Node.js, folosit pentru a construi aplicații web și API-uri. Este ușor de învățat și de utilizat, iar comunitatea sa activă și bogată face ca acesta să fie o alegere populară pentru dezvoltatorii de aplicații web.

În proiectul meu, am decis să folosesc Express.js datorită flexibilității sale și a capacitatea sa de a gestiona o varietate de funcționalități dincolo de crearea simplă a serverului web. Express.js a fost utilizat pentru a gestiona rutele de navigare ale aplicației și pentru a procesa cererile utilizatorilor.

O altă caracteristică importantă a Express.js este middleware-ul. Middleware-ul este un concept cheie în Express.js, care permite dezvoltatorilor să gestioneze diverse aspecte ale aplicației înainte sau după rutele de navigare. Prin intermediul middleware-ului, am putut să efectuez validarea datelor utilizatorului, să implementez autentificarea și autorizarea, să gestionez erorile și multe alte funcționalități.

Express.js a permis, de asemenea, integrarea ușoară cu alte module și biblioteci Node.js. De exemplu, am putut să integrez baza de date MongoDB în aplicație, folosind biblioteca Mongoose, și să utilizez promisiuni și async/await pentru a face cereri către baza de date și pentru a procesa rezultatele.

Pe lângă aceste avantaje, Express.js a fost și ușor de extins. Am putut să adaug funcționalități suplimentare prin intermediul pachetelor npm sau prin crearea propriilor module.

**3.2.1 Rutarea**

Rutarea se referă la modul în care punctele finale (URI) ale unei aplicații răspund la solicitările clientului. Pentru o introducere în rutare, consultați rutare de bază .

Definiți rutarea folosind metode ale appobiectului Express care corespund metodelor HTTP; de exemplu, app.get()pentru a gestiona cererile GET și app.postpentru a gestiona cererile POST. Pentru o listă completă, consultați app.METHOD . De asemenea, puteți utiliza app.all() pentru a gestiona toate metodele HTTP și app.use() pentru a specifica middleware ca funcție de apel invers (consultați Utilizarea middleware pentru detalii).

Aceste metode de rutare specifică o funcție de apel invers (numită uneori „funcții de gestionare”) apelată atunci când aplicația primește o solicitare către ruta (punctul final) și metoda HTTP specificate. Cu alte cuvinte, aplicația „ascultă” cererile care se potrivesc rutelor și metodelor specificate, iar atunci când detectează o potrivire, apelează funcția de apel invers specificată.

De fapt, metodele de rutare pot avea mai multe funcții de apel invers ca argumente. Cu mai multe funcții de apel invers, este important să oferiți nextca argument pentru funcția de apel invers și apoi să apelați next() în corpul funcției pentru a preda controlul următorului apel invers.

Următorul cod este un exemplu de rută foarte simplă.

const express = require('express')

const app = express()

*// respond with "hello world" when a GET request is made to the homepage*

app.get('/', (req, res) => {

  res.send('hello world')

})

Figura 23 – Crearea cererii HTTP - GET

**3.2.2 Metode de traseu**

O metodă de rută este derivată dintr-una dintre metodele HTTP și este atașată unei instanțe a clasei express. Următorul cod este un exemplu de rute care sunt definite pentru metodele GET și POST către rădăcina aplicației. [4]

*// GET method route*

app.get('/', (req, res) => {

  res.send('GET request to the homepage')

})

*// POST method route*

app.post('/', (req, res) => {

  res.send('POST request to the homepage')

})

*// DELETE method route*

app.delete('/', (req, res) => {

  res.send('DELETE request to the homepage')

})

*// PUT method route*

app.put('/', (req, res) => {

  res.send('PUT request to the homepage')

})

*// PATCH method route*

app.patch('/', (req, res) => {

  res.send('PATCH request to the homepage')

})

*// OPTIONS method route*

app.options('/', (req, res) => {

  res.send('OPTIONS request to the homepage')

})

*// HEAD method route*

app.head('/', (req, res) => {

  res.send('HEAD request to the homepage')

})

*// ALL method route*

app.all('/', (req, res) => {

  res.send('ALL request types to the homepage')

})

Figura 24 – Metodelele HTTP - Express

**3.2.3 Parametrii rutei**

Parametrii rutei sunt denumite segmente URL care sunt utilizate pentru a capta valorile specificate la poziția lor în adresa URL. Valorile capturate sunt populate în req.paramsobiect, cu numele parametrului rutei specificat în cale ca cheile respective.

Pentru a defini rute cu parametri de rută, pur și simplu specificați parametrii rutei în calea rutei, așa cum se arată mai jos.

app.get('/users/:userId/books/:bookId', (req, res) => {

  res.send(req.params)

})

Figura 25 – Parametrii rutei users

**3.2.4 Funcțiile middleware**

Funcțiile middleware sunt funcții care au acces la obiectul cerere ( req), la obiectul răspuns ( res) și la nextfuncția din ciclul cerere-răspuns al aplicației. Funcția nexteste o funcție din routerul Express care, atunci când este invocată, execută middleware-ul succesiv middleware-ului curent. Funcțiile middleware pot îndeplini următoarele sarcini:

* Executați orice cod.
* Efectuați modificări la obiectele de solicitare și răspuns.
* Încheiați ciclul cerere-răspuns.
* Apelați următorul middleware din stivă.

Începând cu Express 5, funcțiile middleware care returnează o Promisiune vor apela next(value) atunci când resping sau aruncă o eroare. nextva fi apelat fie cu valoarea respinsă, fie cu Eroare aruncată.

Iată un exemplu de aplicație simplă „Hello World” Express. Restul acestui articol va defini și adăuga trei funcții middleware la aplicație: una numită myLoggercare imprimă un mesaj de jurnal simplu, una numită requestTimecare afișează marcajul de timp al cererii HTTP și una numită validateCookiescare validează cookie-urile primite.

const express = require('express')

const app = express()

const myLogger = function (req, res, next) {

  console.log('LOGGED')

  next()

}

app.use(myLogger)

app.get('/', (req, res) => {

  res.send('Hello World!')

})

app.listen(3000)

Figura 26 – Utilizarea middleware express.js

De fiecare dată când aplicația primește o solicitare, imprimă mesajul „LOGGED” pe terminal. Ordinea de încărcare a middleware-ului este importantă: funcțiile middleware care sunt încărcate primele sunt, de asemenea, executate mai întâi. Dacă myLoggereste încărcat după traseul către calea rădăcină, cererea nu ajunge niciodată la ea și aplicația nu afișează „LOGGED”, deoarece handlerul de rută al căii rădăcină încheie ciclul cerere-răspuns. Funcția middleware myLogger pur și simplu imprimă un mesaj, apoi transmite cererea următoarei funcții middleware din stivă apelând funcția next().

**3.3 MongoDB – sistem de gestiune a bazelor de date**

MongoDB este un sistem de gestiune a bazelor de date (SGBD) orientat spre documente, care utilizează formatul JSON pentru stocarea datelor și permite dezvoltatorilor să creeze aplicații scalabile și performante. Acesta a fost dezvoltat de compania MongoDB Inc. și este disponibil sub licență open-source.

MongoDB utilizează o abordare diferită față de sistemele de gestiune a bazelor de date relaționale (RDBMS) tradiționale. În loc de tabele și înregistrări, MongoDB utilizează colecții și documente. Un document poate fi comparat cu un obiect JSON și poate conține orice număr de perechi cheie-valoare, inclusiv tablouri și alte documente încorporate.

Unul dintre avantajele majore ale utilizării MongoDB este scalabilitatea sa. MongoDB utilizează o arhitectură distribuită, ceea ce înseamnă că poate fi utilizat pe mai multe noduri pentru a gestiona volumul mare de date. Acesta poate fi, de asemenea, configurat pentru a oferi replicare și sharding, permițând astfel distribuirea datelor și a sarcinilor de procesare pe mai multe servere.

MongoDB este de asemenea bine cunoscut pentru performanța sa. Bazele de date MongoDB sunt stocate în format binar, ceea ce permite o manipulare rapidă a datelor. De asemenea, suportă indexarea și interogarea eficientă a datelor, oferind astfel o performanță excelentă în ceea ce privește citirea și scrierea datelor.

În general, MongoDB este o soluție puternică pentru gestionarea datelor în aplicațiile moderne. Este scalabil, performant și flexibil, permițând dezvoltatorilor să creeze aplicații robuste și scalabile care pot gestiona volume mari de date.

În următoarea figură (fig. 27) puteți observa un exemplu de document creat în MongoDB. Putem observa că însăși documentul e de formatul JSON, ușor de citit și de înțeles. Putem observa câteva câmpuri, așa ca \_id – care este implicit pentru orice document, firstname, lastname și address care are un sub JSON cu adresa, orașul, statul și zip codul.

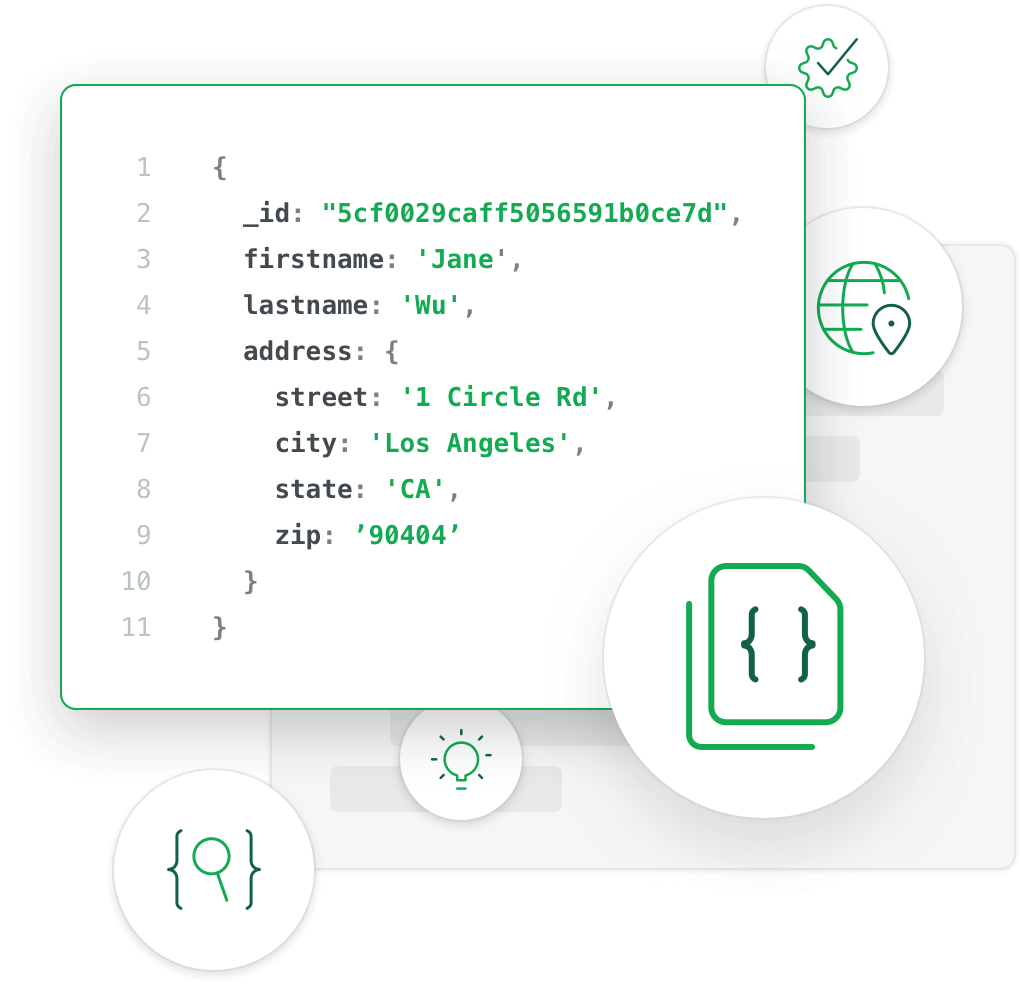


Figura 27 – Exemplu de document în MongoDB

**3.3.1 Mongoose**

Mongoose este o bibliotecă JavaScript pentru MongoDB, care oferă o interfață mai ușor de folosit pentru a interacționa cu baza de date MongoDB folosind Node.js. Acesta oferă un sistem de definire a schemelor pentru colecțiile MongoDB, care simplifică procesul de definire a structurii datelor și de validare a datelor introduse de utilizator.

Mongoose oferă, de asemenea, multe alte funcții utile, cum ar fi crearea, citirea, actualizarea și ștergerea documentelor, crearea de interogări complexe și gestionarea tranzacțiilor. De asemenea, permite definirea de modele pentru obiecte de tipul Schema, care permite o structurare mai clară și mai ușor de utilizat a datelor în aplicații node.js. Mongoose este o unealtă foarte populară printre dezvoltatorii Node.js și este utilizată în multe proiecte mari. Oferă o interfață intuitivă pentru programare și este ușor de utilizat, ceea ce îl face o alegere ideală pentru oricine lucrează cu MongoDB și Node.js.

În codul de mai jos puteți observa un exemplu de conexiune cu baza de date MongoDB, crearea și introducerea a două modele (Scheme) userSchema și postSchema.

*// Requiring module*

const mongoose = require('mongoose');

*// Connecting to database*

mongoose.connect('mongodb://localhost:27017/mydatabase,

    {

        useNewUrlParser: true,

        useUnifiedTopology: true,

        useFindAndModify: false

    });

*// Creating Schemas*

const userSchema = new mongoose.Schema({

    username: String,

    email: String

})

const postSchema = new mongoose.Schema({

    title: String,

    postedBy: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User"

    }

})

*// Creating models from userSchema and postSchema*

const User = mongoose.model('User', userSchema);

const Post = mongoose.model('Post', postSchema);

*// Query to find and show all the posts*

Post.find()

    .then(p => console.log(p))

    .catch(error => console.log(error));

Figura 28 – Crearea schemelor Mongoose

**3.4 Git – sistem control al versiuniilor**

Git este un sistem de control al versiunilor pentru gestionarea codului sursă al unui proiect. A fost creat de Linus Torvalds în 2005 și este utilizat în mod obișnuit de dezvoltatorii de software pentru a urmări modificările, colabora și sincroniza codul sursă între diferite echipamente.

Git este un sistem distribuit de control al versiunilor, ceea ce înseamnă că fiecare copie locală a unui depozit Git conține întregul istoric al proiectului, precum și toate modificările și versiunile anterioare. Aceasta face ca Git să fie un instrument puternic pentru lucrul cu proiecte complexe și permite dezvoltatorilor să lucreze la o copie locală a codului sursă, fără a afecta dezvoltarea altor colaboratori.

Git are o serie de caracteristici utile, cum ar fi gestionarea ramurilor (branching), care permite dezvoltatorilor să creeze ramuri separate ale codului sursă pentru a dezvolta funcționalități noi sau pentru a remedia probleme fără a afecta ramura principală a codului. De asemenea, Git are o gamă largă de instrumente și funcții de colaborare, care permit dezvoltatorilor să lucreze împreună la același cod sursă și să își sincronizeze modificările într-un mod eficient.

În figura ce urmează (fig. 29) am reprezentat comenzile de bază a sistemului de versionare – git.

În primul rând inițiem repozitoriu dacă nu e inițializat cu comanda – git init.

Apoi, folosim comanda git add pentru adăugarea unui sau mai multor fișiere în git.

Apoi, folosim comanda git commit pentru a comite modificările executate în git.

După care facem git push și modificările noastre vor apărea pe repozitoriu de git – așa ca github ș.a.

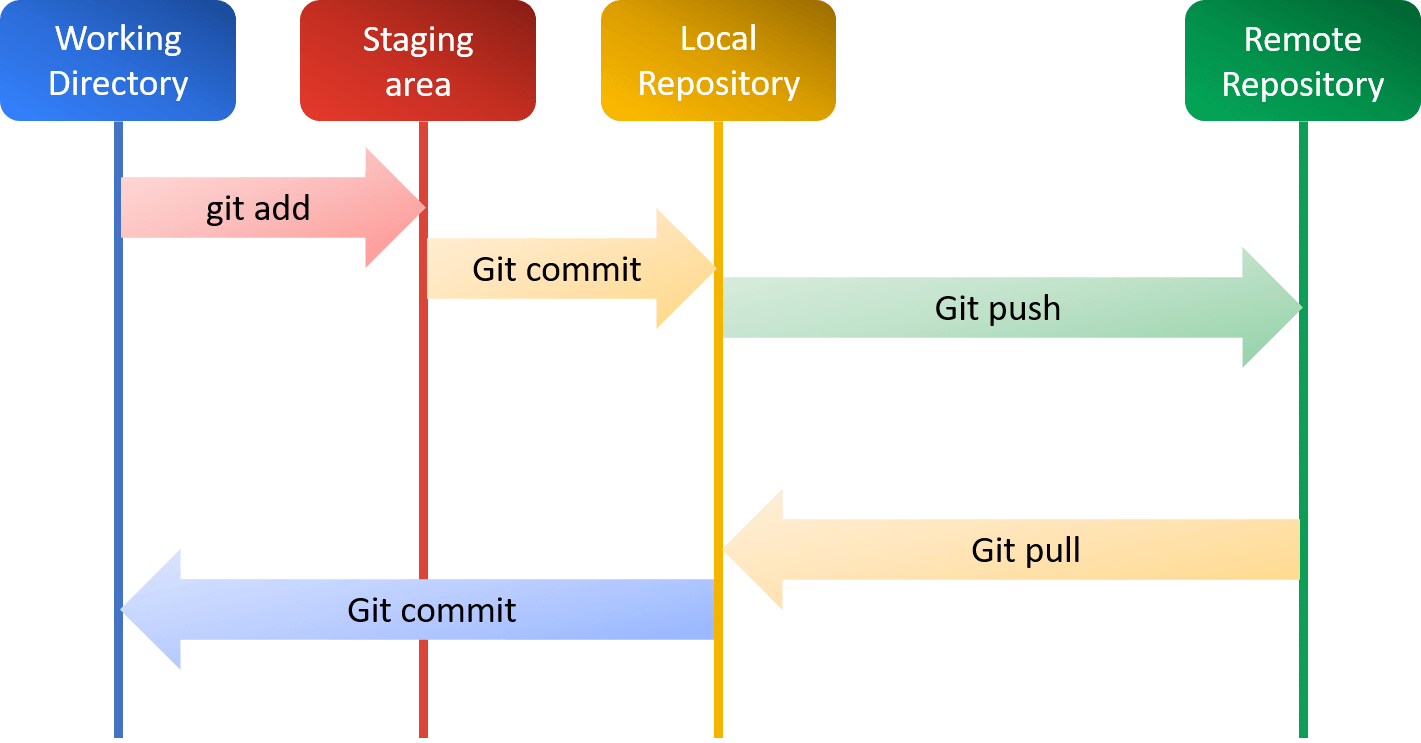


Figura 29 – Comenzile de bază Git

# 3.5 Descrierea la nivel de cod pe module

În acest capitol voi reprezenta descrierea codului pe module, de asemenea voi reprezenta câteva secvențe de cod pentru a fi mai clar.

Să începem cu structura proiectului realizat. Mai jos puteți observa cum este structurat proiectul meu. Puteți observa 2 module principale (foldere) – backend și frontend. Ele sunt configurate aparte și codul nu se intersectează între ele. Este o practică des folosită pentru aplicații NU de tipul SSR (Server Side Rendering) – ceea ce este a mea. Puteți observa o structură des utilizată a aplicațiilor Express.JS (în combinație cu mongoose) și a aplicațiilor scrise în Vue.JS (în combinație cu Vuex, Vue Router). Fișierul din root (app.bat) rulează ambele procese – frontend-ul și backend-ul.

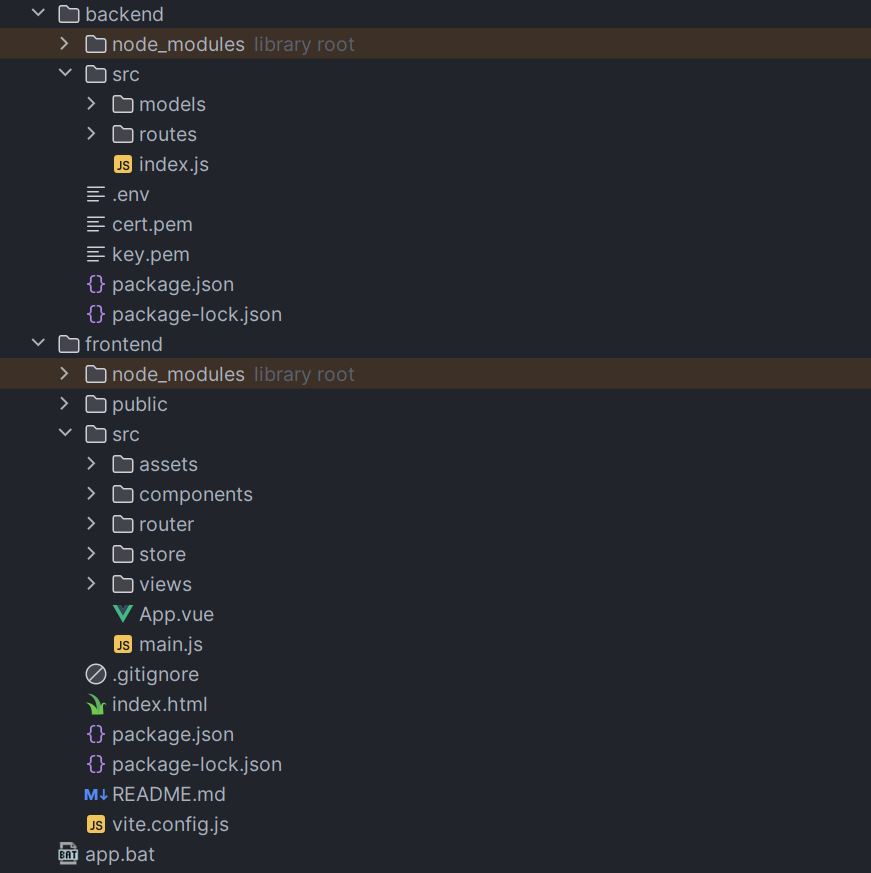


Figura 30 – Structura proiectului realizat

Primul exemplu de cod pe care vreau să-l prezint este fișierul principal din backend (src/index.js). În acest fișier se inițiază instanța de server și se face conexiune la baza de date. De asemenea se implementează ruterile create pentru user și posts.

const https = require('https');

const express = require('express');

const cookieSession = require('cookie-session');

const mongoose = require('mongoose');

const userRouter = require('./routes/user');

const postRouter = require('./routes/post');

const app = express();

const cors = require('cors');

const fs = require('fs');

require('dotenv').config();

app.use(

  cors({

    origin: 'http://localhost:5000',

    credentials: true,

  }),

);

app.use(

  cookieSession({

    name: process.env.SECRET\_NAME,

    keys: [process.env.SECRET\_KEY],

    secret: process.env.SECRET\_WORD,

    httpOnly: false,

    SameSite: 'None',

    maxAge: 365 \* 24 \* 60 \* 60 \* 1000, *// 1 year*

  }),

);

app.use(express.json());

app.use(express.urlencoded({extended: true}));

app.set('trust proxy', true);

app.use((req, res, next) => {

  res.setHeader('Content-type', 'application/json; charset=UTF-8');

  next();

});

const port = process.env.PORT || 3000;

app.listen(port, () => console.log('http://localhost:' + port));

mongoose.set('strictQuery', true);

mongoose.connect(process.env.MONGO\_URI + '', {serverSelectionTimeoutMS: 5000}, function (err) {

  if (err) {

    console.log('MongoDB Connection Error!');

    app.use((req, res) => {

      return res.json({error: 'MongoDB Connection Error'});

    });

    return;

  }

  const routes = [

    {

      method: "GET",

      url: "http://localhost:3000/api/user/check",

    },

    {

      method: "GET",

      url: "http://localhost:3000/api/post/get",

    }

  ]

  app.get('/', (req, res) => {

    return res.json({success: true, database: 'Connected', routes});

  });

  app.use('/api/user', userRouter);

  app.use('/api/post', postRouter);

});

Figura 31 – Inițierea serverului (backend)

În exemplul prezentat mai jos puteți observa crearea unei Scheme utilizate pentru baza de date MongoDB. Puteți observa careva câmpuri, așa ca userId, email, name, avatar, password, access, userType, regType, githubId. Este o structură standart pentru crearea unei scheme mongodb.

const { Schema, model, mongoose } = require('mongoose');

const AutoIncrement = require('mongoose-sequence')(mongoose);

const schema = new Schema({

    userId: {

        type: Number,

        default: 0

    },

    email: {

        type: String,

        required: true,

        trim: true

    },

    name: {

        type: String,

        required: false,

    },

    avatar: {

      type: String,

      default: 'https://placehold.co/512x512/EEE/999'

    },

    password: {

        type: String,

        default: null

    },

    access: {

        type: Number,

        required: true,

        default: 0

    },

    userType: {

        type: Number,

        required: true,

        default: 0

    },

    regType: {

        type: Number,

        default: 0

    },

    githubId: {

        type: Number,

        default: null

    }

}, { versionKey: false, timestamps: true });

schema.plugin(AutoIncrement, { inc\_field: 'userId' });

module.exports = model('User', schema);

Figura 32 – Schema pentru documentul Users

În figura ce urmează (fig. 33) am reprezentat main.js – punctul principal din partea de fronend a platformei. Putem observa că este folosit vuex (store), vue router, biblioteca Bootstrap, și conexiunea stilurilor din assets/scss/main.scss – care conține toate stilurile platformei.

import { createApp } from 'vue'

import App from '@/App.vue'

import router from '@/router'

import store from '@/store'

import "bootstrap"

import auth from '@/store/auth'

import './assets/scss/main.scss'

const app = createApp(App)

app.use(auth.auth)

app.use(router, store)

app.mount('#app')

Figura 33 – Main.js – inițierea Vue App

În figura ce urmează (fig. 34) este reprezentată Layout-ul principal (App) a platformei Vue. Putem observa că global conectez Header și Footer, iar în main se conectează RouterView – utilizat pentru rutarea paginilor. În acest loc se va face render la conținutul paginilor.

<script setup>

import { RouterView } from 'vue-router'

import Header from '@/components/Header.vue'

import Footer from '@/components/Footer.vue'

</script>

<template>

  <Header/>

  <main>

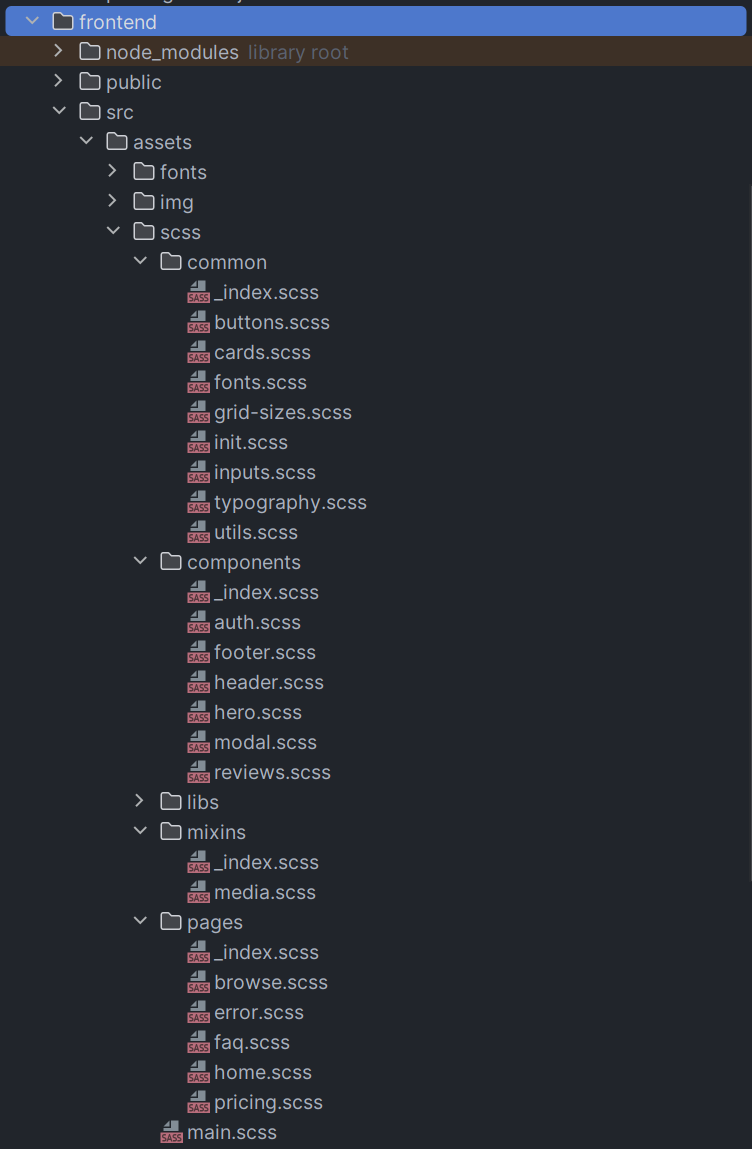
    <RouterView />

  </main>

  <Footer/>

</template>

Figura 34 – App.vue – Base Layout pentru toată platforma



În figura alăturată (fig.35) puteți observa structura stilurilor create. E important să avem o structură curată și clară ca pe viitor să putem face modificări ușor și fără mult timp de analizare. Asta ne permite să avem o scalabilitate mare – fapt ce va influența pozitiv în viitor.

Figura 35 – Structura stilurilor (SCSS)

# 3.6 Testarea sistemului

Testarea sistemului este un aspect esențial în dezvoltarea unei aplicații, iar în acest subcapitol voi descrie metoda de testare, instrumentele folosite și rezultatele obținute în cadrul proiectului meu. Pentru a testa sistemul dezvoltat, am folosit o combinație de teste automate și manuale.

Testele automate au fost implementate cu ajutorul framework-ului Jest, o platformă de testare a codului JavaScript, care oferă o serie de funcționalități pentru a testa diverse scenarii ale aplicației.

Am creat teste pentru diverse componente ale aplicației, inclusiv interacțiunile dintre acestea și cu baza de date MongoDB. În ceea ce privește testele manuale, am efectuat teste de utilizare a aplicației pentru a evalua experiența utilizatorului și pentru a identifica problemele de interfață sau de funcționalitate.

Am creat scenarii de utilizare realiste și am evaluat cum se comportă aplicația în aceste situații. De asemenea am folosit instrumentul Postman pentru testarea cererilor la server

# 3.6.1 Jest

Jest este un framework de testare pentru aplicații JavaScript, dezvoltat și întreținut de către Facebook. Acesta este utilizat în special pentru a testa codul sursă al aplicațiilor React, dar poate fi utilizat și pentru alte aplicații JavaScript.

Jest vine cu o varietate de funcții de testare, inclusiv testarea unitară, testarea integrării și testarea componentelor. Acesta oferă, de asemenea, o gamă largă de asertori pentru a verifica diverse condiții și funcții ale codului.

O caracteristică importantă a lui Jest este faptul că este foarte ușor de configurat și de utilizat. Este integrat cu Babel, ceea ce înseamnă că poate fi utilizat pentru testarea codului JavaScript scris în ES6 sau în orice alt limbaj de programare care poate fi compilat la JavaScript. Jest oferă și o interfață de linie de comandă foarte ușor de utilizat pentru rularea testelor.

Un alt avantaj al lui Jest este viteza sa de execuție. Acesta este conceput să ruleze teste foarte rapid, cu ajutorul unor funcții de caching inteligente și a unui proces de paralelizare. Acest lucru face ca testarea să fie mai rapidă și mai eficientă, ceea ce este important în dezvoltarea de software.

În concluzie, Jest este un framework de testare foarte popular și eficient pentru aplicațiile JavaScript, care oferă o gamă largă de funcții și asertori, este ușor de configurat și de utilizat, și rulează teste foarte rapid.

const mongoose = require('mongoose');

const { MongoMemoryServer } = require('mongodb-memory-server');

const User = require('./models/User'); *// exemplu model de utilizator*

let mongoServer;

beforeAll(async () => {

  mongoServer = await MongoMemoryServer.create();

  const uri = mongoServer.getUri();

  await mongoose.connect(uri);

});

afterAll(async () => {

  await mongoose.disconnect();

  await mongoServer.stop();

});

describe('User model', () => {

  beforeEach(async () => {

*// sterge toate datele din baza de date*

    await User.deleteMany({});

  });

  it('should save a user', async () => {

    const user = new User({

      fullName: 'Dan Rotaru',

      email: 'rotaru.dan@isa.utm.md',

      password: 'password'

    });

    await user.save();

    expect(user.isNew).toBe(false); *// verifica daca utilizatorul a fost salvat cu succes*

  });

  it('should find a user by email', async () => {

    const user = new User({

      fullName: 'Dan Rotaru',

      email: 'rotaru.dan@isa.utm.md',

      password: 'password'

    });

    await user.save();

    const foundUser = await User.findOne({ email: 'rotaru.dan@isa.utm.md' });

    expect(foundUser.name).toEqual(user.name); *// verifica daca utilizatorul a fost gasit cu succes dupa email*

  });

});

Figura 36 – Utilizarea Jest

În acest exemplu, am creat o bază de date MongoDB temporară folosind mongodb-memory-server. Apoi am definit un model de utilizator și am scris teste Jest pentru a verifica dacă utilizatorii sunt salvați și găsiți cu succes în baza de date.

# 3.6.2 Postman

Postman este o unealtă puternică și versatilă pentru testarea API-urilor și a aplicațiilor web în general. Oferă o interfață intuitivă și ușor de utilizat, care permite utilizatorilor să trimită cereri HTTP și să primească răspunsuri în timp real.



Figura 36 – Postman – testarea cererilor HTTP

În ceea ce privește testarea unui sistem, Postman este o alegere populară pentru că poate fi folosit atât pentru teste manuale cât și pentru automatizare. Prin automatizarea testelor cu Postman, putem să economisim mult timp și să îmbunătățim calitatea testelor noastre. Pentru a începe testarea cu Postman, trebuie să încărcați o colecție de teste care conține toate cererile necesare pentru a testa sistemul. Aceste cereri ar trebui să acopere toate funcționalitățile sistemului, astfel încât să puteți fi siguri că fiecare componentă a fost testată corespunzător. Una dintre caracteristicile importante ale Postman este suportul pentru variabile de mediu, care ne permit să definim o valoare o dată și să o folosim în întreaga colecție de teste. Acest lucru ne poate economisi timp și reduce riscul de a introduce erori umane.

Atunci când testăm cu Postman, putem utiliza și teste automate pentru a verifica răspunsurile sistemului. Putem defini aserțiuni care verifică dacă răspunsurile sistemului sunt cele așteptate și dacă respectă anumite condiții, cum ar fi status code-ul și conținutul răspunsului. Dacă apare o problemă în timpul testării, Postman oferă și o funcționalitate de debug, care ne permite să analizăm cererea și răspunsul și să identificăm rapid problemele.

Testarea cu Postman este un proces puternic și flexibil, care poate fi utilizat pentru a testa aplicațiile web cu ușurință. Prin definirea și automatizarea testelor, putem economisi mult timp și ne putem asigura că sistemul funcționează corespunzător și că respectă cerințele noastre.

# 4 DOCUMENTAREA PRODUSULUI REALIZAT

Descrieți cerințele tehnice pentu aplicație, cum se gasește aplicația, cum se instalează.

În acest capitol se descrie utilizarea aplicației la nivel de administrator și la nivel de utilizator foarte detaliat, ca un manual de utilizare a aplicației.

Descrierea fiecărui butonaș, meniu, toate funcțiile.

# 5 ESTIMAREA COSTURILOR PROIECTULUI

Evaluarea cheltuielilor asociate dezvoltării unei aplicații software este un domeniu mai puțin structurat, care se bazează pe estimări și aproximații. Cu toate acestea, există o serie de metode care permit calcularea costului total al unui proiect software pe baza unui număr limitat de factori de cost relevanți. Estimarea unui proiect în mare parte reprezintă bugetul acelui proiect. Estimarea costurilor reprezintă o componentă critică a oricărui proiect software. O estimare precisă poate asigura o planificare mai bună a resurselor și o abordare mai eficientă a proiectului, în timp ce o estimare greșită poate duce la costuri suplimentare și întârzieri în proiect. În general, estimarea costurilor unei aplicații software se bazează pe evaluarea cheltuielilor asociate cu dezvoltarea, implementarea, testarea și întreținerea aplicației. Estimarea costurilor poate fi mai puțin formalizată decât alte aspecte ale proiectului software, cum ar fi specificațiile tehnice sau design-ul aplicației, deoarece este nevoie de estimări și aproximări.

Există mai multe metode de estimare, pe care le putem utiliza în cadrul unui proiect pentru estimarea costurilor, timpului și resursele necesare:

* Metoda de estimare a punctelor de funcționalitate – se bazează pe numărul de funcționalități sau de cerințe ale aplicației, estimându-se numărul de ore necesare pentru implementarea acestora.
* Metoda de estimare bazată pe experiență – se bazează pe experiența anterioară a dezvoltatorilor în ceea ce privește proiecte similare și se ajustează pentru a reflecta cerințele specifice ale proiectului actual.
* Metoda de estimare pe bază de expertiză – se bazează pe părerea unor experți din cadrul industriei care au experiență în proiecte similare.
* Metode de estimare de tip “bottom-up” – care, se bazează pe estimarea individuală a fiecărei sarcini sau activități din cadrul proiectului și adăugarea acestora pentru a obține o estimare totală.
* Metoda de estimare prin comparare - se bazează pe compararea costurilor, timpului și resurselor necesare pentru proiecte similare finalizate anterior.

Este important să ținem cont de specificul proiectului și de experiența și cunoștințele echipei pentru a alege metoda cea mai potrivită pentru nevoile dvs. Estimarea costurilor poate fi realizată utilizând unelte software specializate sau prin intermediul unor calcule manuale. Indiferent de metoda utilizată, este important să se ia în considerare toți factorii care pot influența costurile, cum ar fi nivelul de complexitate al aplicației, cerințele tehnice, timpul necesar și resursele disponibile.

De asemenea, este important să se țină cont de factorii care pot duce la creșterea sau scăderea costurilor, cum ar fi fluctuațiile de pe piața forței de muncă sau modificările aduse cerințelor de proiect.

Un aspect important al estimării costurilor este reprezentat de identificarea și cuantificarea riscurilor asociate cu proiectul. Riscurile pot include întârzieri în livrarea proiectului, depășirea bugetului sau probleme tehnice neprevăzute. Identificarea și cuantificarea acestor riscuri poate ajuta la estimarea costurilor suplimentare necesare pentru a le gestiona.

# 5.1 Definirea diagramei WBS

WBS este o descompunere ierarhică și incrementală a proiectului în faze, livrabile și pachete de lucru. Este o structură arborescentă , care arată o subdiviziune a efortului necesar pentru atingerea unui obiectiv, de exemplu, un program, un proiect și un contract. Într-un proiect sau contract, WBS este dezvoltat pornind de la obiectivul final și subdivându-l succesiv în componente gestionabile în ceea ce privește dimensiunea, durata și responsabilitatea (de exemplu, sisteme, subsisteme, componente, sarcini , subsarcini și muncă). pachete) care includ toți pașii necesari atingerii obiectivului.

WBS este organizat în jurul produselor primare ale proiectului (sau rezultatelor planificate) în loc de munca necesară pentru producerea produselor (acțiuni planificate). Întrucât rezultatele planificate sunt scopurile dorite ale proiectului, ele formează un set relativ stabil de categorii în care pot fi colectate costurile acțiunilor planificate necesare pentru realizarea lor. Un WBS bine conceput facilitează atribuirea fiecărei activități de proiect unui singur element terminal al WBS. Pe lângă funcția sa în contabilitatea costurilor, WBS ajută și la maparea cerințelor de la un nivel de specificație a sistemului la altul, de exemplu, o matrice de referință încrucișată care mapează cerințele funcționale la documente de proiectare de nivel înalt sau de nivel scăzut. WBS poate fi afișat orizontal în conturformă sau pe verticală ca o structură arborescentă (ca o organigramă).

Dezvoltarea WBS are loc în mod normal la începutul unui proiect și precede planificarea detaliată a proiectului și a sarcinilor. Realizarea unui WBS este primul pas în dezvoltarea unui [program de proiect](https://www.projectmanager.com/project-scheduling). Acesta definește toată munca care trebuie finalizată (și în ce ordine) pentru a atinge scopurile și obiectivele proiectului. Vizualând proiectul în acest mod, puteți înțelege domeniul de aplicare al proiectului și puteți aloca resurse pentru toate sarcinile proiectului.

Structura de defalcare a muncii din managementul de proiect definește vizual părți gestionabile ale unui proiect pe care o echipă le poate înțelege, deoarece fiecare parte a structurii de defalcare a muncii oferă detalii suplimentare.  Scopul WBS în managementul proiectelor este de a face un proiect mare gestionabil ajută la:

* elaborarea unui program;
* determinarea costului proiectului;
* setarea dependenței;
* scrierea unei declarație de lucru;
* atribuirea responsabilităților și clarificarea rolurilor;
* urmărirea progresul unui proiect;

# 5.2 Decompoziția lucrării proiectului, crearea WBS

În figura de mai jos (figura 99) este reprezentată o tehnică de construcție a structurii de defalcare a lucrării care demonstrează regula 100% și tehnica „elaborarii progresive”. La WBS Nivelul 1, arată 100 de unități de lucru ca sfera totală a unui proiect de proiectare și construire a unei biciclete personalizate. La WBS Nivelul 2, cele 100 de unități sunt împărțite în șapte elemente. Numărul de unități alocate fiecărui element de lucru poate fi bazat pe efort sau cost; nu este o estimare a duratei sarcinii.

Cele mai mari trei elemente ale WBS Nivelul 2 sunt subdivizate în continuare la Nivelul 3. Cele mai mari două elemente de la Nivelul 3 reprezintă fiecare doar 17% din sfera totală a proiectului. Aceste elemente mai mari ar putea fi subdivizate în continuare folosind tehnica de elaborare progresivă descrisă mai sus. Acesta este un exemplu de abordare bazată pe produs (care ar putea fi produs final sau livrabil sau bazat pe muncă), în comparație cu abordarea în etape (care ar putea fi etape închise într-un ciclu de viață formal de dezvoltare a sistemelor sau evenimente forțate (de exemplu, trimestrial). actualizări sau o rebugetare a unui an fiscal), sau o abordare bazată pe abilități/roluri.

Designul WBS poate fi susținut de software (de exemplu, o foaie de calcul) pentru a permite acumularea automată a valorilor punctelor. Estimările efortului sau costurilor pot fi elaborate prin discuții între membrii echipei de proiect. Această tehnică de colaborare oferă o perspectivă mai bună asupra definițiilor domeniului de aplicare, a ipotezelor subiacente și a consensului cu privire la nivelul de granularitate necesar pentru gestionarea proiectelor.

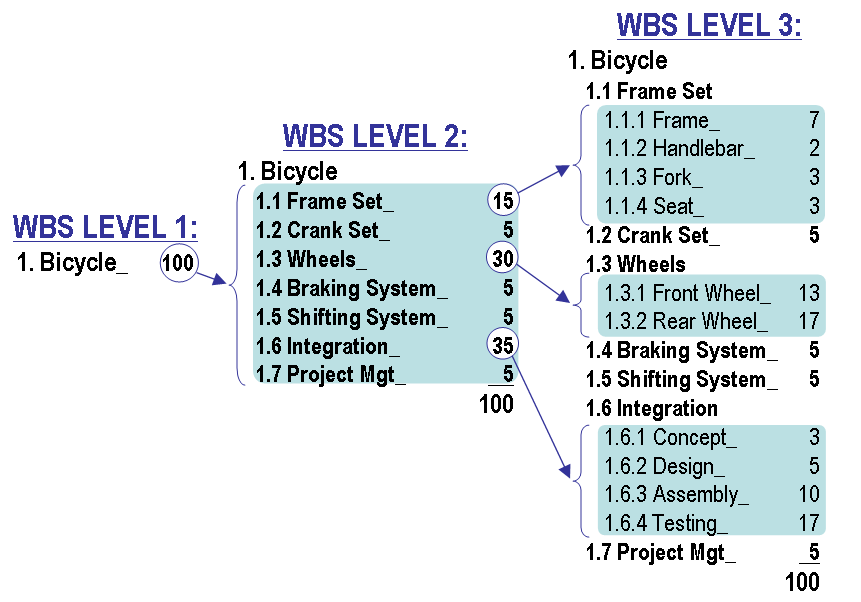


Figura 99 – Construirea diagramei WBS

Mai jos, în figura 100 am reprezentat diagrama WBS pentru proiectarea produsului software. Putem observa 5 entități de bază – Management de proiect, Cerințele produsului, Proiectarea design-ului, Dezvoltarea produsului și Testarea acestuia. La management de proiect avem nevoie de planificare, meetinguri (apeluri și discuții referitor la proiect), supraveghere (sau supervision) – procesul care implică o întâlnire regulată a managerului și interacțiunea cu lucrătorii, administrare. La cerințele produsului avem analiza cerințelor utilizaorilor, clasificarea cerințelor, documentarea cerințelor. La proiectarea desing-ului avem design software, client, server design și documnetarea design-ului. În entitatea de dezvoltare avem 4 ramuri – software, client, server, scrierea documentației. Testarea are 2 subramuri – testarea software, scrierea raportelor de testare.

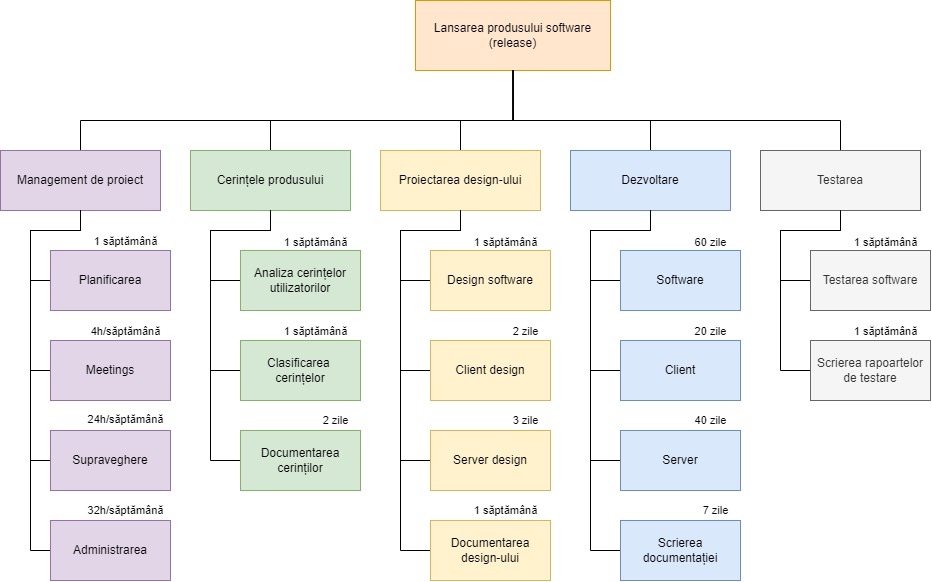


Figura 100 – Diagrama WBS de proiectare a produsului software

# 5.3 SDLC - Ciclul de viață al dezvoltării software-ului

SDLC (în engleză – The Software Development Life Cycle sau română – Ciclul de viață al dezvoltării software-ului) este un proces urmat pentru un proiect software, în cadrul unei organizații software. Acesta constă într-un plan detaliat care descrie modul de dezvoltare, întreținere, înlocuire și modificare sau îmbunătățire a unui software specific. Ciclul de viață definește o metodologie pentru îmbunătățirea calității software-ului și a procesului general de dezvoltare. Următoarea figură este o reprezentare grafică a diferitelor etape ale unui SDLC tipic. Din aceasta putem observa următoarele etape:

* Planificare
* Analiză
* Design
* Implementare
* Testare și integrare
* Mentenanță



Figura 101 – Reprezentarea ciclului de dezvoltare software (SDLC)

Planificarea – etapa de planificare este probabil cea mai importantă etapă de dezvoltare. Aceasta ajută la definirea problemei și domeniului de aplicare a oricăror sisteme existente, precum și la determinarea obiectivelor pentru noile lor sisteme. Dezvoltând o schiță eficientă pentru viitorul ciclu de dezvoltare, teoretic vor surprinde probleme înainte de a afecta dezvoltarea. Poate cel mai important, etapa de planificare stabilește calendarul proiectului, care poate fi de o importanță cheie dacă dezvoltarea este pentru un produs comercial care trebuie trimis pe piață până la un anumit timp. Planificarea cerintelor de asigurare a calitatii si identificarea riscurilor asociate proiectului se face si in faza de planificare. Rezultatul studiului de fezabilitate tehnică este definirea diferitelor abordări tehnice care pot fi urmate pentru implementarea cu succes a proiectului cu riscuri minime. Dezvoltatorii vor sublinia mai întâi detaliile aplicației generale, alături de aspecte, cum ar fi:

* Interfețe cu utilizatorul
* Interfețe de sistem
* Cerințe de rețea și rețea
* Baze de date

A două etapă este etapa de analiză – în această etapă a ciclului de viață al dezvoltării software, specialiștii colectează meticulos cerințe precise de la client pentru a prezenta o soluție adaptată nevoilor acestuia. Orice neclarități trebuie elucidate numai în această etapă. Faza de analiză adună, de asemenea, cerințele de afaceri și identifică orice riscuri potențiale. Acest pas în SDLC include, de asemenea, un studiu de fezabilitate, care definește toate punctele forte și punctele slabe ale proiectului pentru a evalua viabilitatea globală a proiectului.‍

Obiectivele pe care le atingem în această etapă sunt identificate ca sistemul de funcții de care afacere. Pentru aceasta, dezvoltatorii de software realizează trei activități principale:

* enumerarea nevoilor sau cerințelor afacerii
* dezvoltarea diagramelor de proces sau a unei conducte de dezvoltare
* efectuarea analizei

Etapa de analiză include:

* clarificarea detaliilor specifice necesare pentru dezvoltarea software-ului
* determinarea ideilor prototip inițiale:
* ce funcții ar putea fi cele mai potrivite pentru noul produs

ce USP-uri (Unique Selling Points) ar trebui să aibă viitorul tău software pentru a concura bine pe piață.

A treia etapă este etapa de arhitectură și design – în această etapă ne ocupăm de design-ul și arhitectura produsului software. Această fază de proiectare pune o bază pentru următorul pas al ciclului de viață, care este dezvoltarea. De asemenea în această etapă se presupune încă doi pași – Proiectare la nivel înalt (HLD) și Proiectare la nivel scăzut (LLD). Rezultatul primului este arhitectura viitoare a unui produs software, în timp ce pasul LLD descrie cum ar trebui să funcționeze fiecare caracteristică a produsului. De asemenea, în această fază este dezvoltată specificația bazei de date pentru a decide gestionarea și stocarea datelor pentru procesarea, preluarea sau evaluarea viitoare. Printre altele, echipa de dezvoltare va analiza componentele de bază, structura, procesarea și procedurile pentru ca sistemul să atingă obiectivul declarat. O abordare de proiectare definește în mod clar toate modulele arhitecturale ale produsului împreună cu comunicarea și reprezentarea fluxului de date cu modulele externe și terțe (dacă există). Designul intern al tuturor modulelor arhitecturii propuse ar trebui să fie clar definit cu cele mai mici detalii din DDS.

Următoarea etapă este etapa de implementare – în această etapă ne ocupăm de însăși implementarea produsului, începe dezvoltarea efectivă a produsului software. Etapa de implementare sau dezvoltare este partea în care dezvoltatorii scriu de fapt cod și construiesc aplicația conform documentelor de proiectare anterioare și specificațiilor subliniate. Codul programului produsului este construit conform specificațiilor documentului de proiectare. În teorie, toată planificarea anterioară și subliniată ar trebui să facă faza reală de dezvoltare relativ simplă. Dezvoltatorii vor urma orice instrucțiuni de codare așa cum sunt definite de organizație și vor folosi diferite instrumente, cum ar fi compilatoare, depanatoare și interpreți. Dezvoltatorul de sine stătător alege un limbaj potrivit de dezvoltare, analizând cerințele sistemului și limbajele cele mai potrivite pentru implementarea acestor cerințe.

Testarea și integrarea – în etapa respectivă ne ocupăm cu testarea softului. Construirea de software nu este sfârșitul. Acum trebuie testat pentru a vă asigura că nu există erori și că experiența utilizatorului final nu va fi afectată negativ în niciun moment. Această etapă este de asemenea foarte importantă, ca în final utilizatorii obișnuiți să nu observe careva bug-uri sau neclarități în timpul folosirii softului. E important ca aceste bug-uri să fie identificate din timp și corectate până a ajunge la “production”. În timpul etapei de testare, dezvoltatorii își vor parcurge software-ul cu un pieptene cu dinți fini, notând eventualele erori sau defecte care trebuie urmărite, remediate și, ulterior, retestate.

Această etapă este de obicei un subset al tuturor etapelor, ca în modelele moderne SDLC, activitățile de testare sunt implicate în cea mai mare parte în toate etapele SDLC. Totuși, această etapă se referă la doar etapa de testare a produsului în care defectele produsului sunt raportate, urmărite, remediate și retestate, până când produsul atinge standardele de calitate definite în SRS.

Mentenanța – nu e suficient de creat produsul, pe lângă asta mai trebuie de întreținut. Mentinanța este de asemenea o etapă foarte importantă, fără de care nu există nici un produs real. Da, testerii au testat softul, însă ei tot nu pot lua în calcul oricare scenariu, și de aceea totuși pot apărea careva neclarități sau bug-uri pe production. După ce s-a depistat acestea, noi ca dezvoltatorii trebuie să facem fix cât mai curând posibil. SDLC nu se termină când software-ul ajunge pe piață. Dezvoltatorii trebuie să treacă acum într-un mod de întreținere și să înceapă să practice orice activități necesare pentru a gestiona problemele raportate de utilizatorii finali. În plus, dezvoltatorii sunt responsabili pentru implementarea oricăror modificări de care software-ul ar putea avea nevoie după implementare.

Odată ce produsul este testat și gata de a fi implementat, acesta este lansat oficial pe piața corespunzătoare. Uneori, implementarea produsului are loc în etape, conform strategiei de afaceri a acelei organizații. Produsul poate fi lansat mai întâi într-un segment limitat și testat în mediul de afaceri real (UAT- Testare de acceptare a utilizatorilor).

Apoi, pe baza feedback-ului, produsul poate fi lansat așa cum este sau cu îmbunătățiri sugerate în segmentul de piață vizat. După ce produsul este lansat pe piață, întreținerea acestuia se face pentru baza de clienți existentă.

Aceasta poate include gestionarea erorilor reziduale care nu au putut fi corectate înainte de lansare sau rezolvarea de noi probleme care apar din cauza rapoartelor utilizatorilor. Sistemele mai mari pot necesita etape de întreținere mai lungi în comparație cu sistemele mai mici.

# 5.4 Costurile proiectului

Cum am menționat în capitolul 5.1 – un program software poate avea diferite cheltuieli, asociate cu dezvoltarea, implementarea, testarea și întreținerea aplicației. De asemenea avem și cheltuieli de promovare și marketing, care, la fel sunt unele din cele mai costisitoare. Desigur nu trebuie să uităm și de cheltuielile pentru serviciile web (hosting). Pentru platforma noastră nu e destul un simplu hosting web. Pentru că ea are partea de backend (nodejs), partea de frontend și stocarea bazei de date MongoDB noi avem nevoie de un server, un server VPS. Ce este un VPS și pentru ce este folosit?

Un server privat virtual (VPS) poate fi folosit pentru a găzdui site-uri web (e-commerce, conținut, media) și aplicații software (portale, extranet, soluții colaborative, wiki, CRM). Spre deosebire de găzduirea partajată, datele sunt izolate pe o mașină virtuală care este dedicată utilizatorului.

VPS este un compromis perfect între planurile de găzduire web și serverele fizice, combinând fiabilitatea cu performanța ridicată a unui server dedicat — dar fără dificultatea de a gestiona hardware-ul fizic al unui server.

Ok, deci haideți să începem cu cheltuielile cele mai mici, cheltuielile pe găzduirea platformei mele pe web, așa ca ea să fie disponibilă oricărui utilizator, din orice colț al lumii, având doar conexiune la internet.

În următorul tabel am prezentat o listă de hosting providers, și am făcut o comparație la preț și specificații – memoria operativă, memoria pe disk, bandwidth (traficul maxim admisibil lunar)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hosting Provider | VPS Plan | RAM | Disk Space | Bandwidth/mo | $/mo |
| Namecheap | Pulsar | 2 GB | 40 GB SSD | 1 TB | 6.88 $ |
| A2 Hosting | LIFT 4 | 4 GB | 150 GB SSD | 2 TB | 39.99 $ |
| FASTCOMET | Cloud 1 | 2 GB | 50 GB SSD | 2 TB | 46.16 $ |
| ScalaHosting | Build #1 | 4 GB | 50 GB SSD | Unlimited | 29.95 $ |
| Contabo | CLOUD VPS S | 8 GB | 50 GB NVME | 32 TB | 5.99 $ |
| DreamHOST | VPS BASIC | 1 GB | 30 GB SSD | Unlimited | 9.99 $ |
| AccuWebHosting | GP.1 | 1 GB | 30 GB SSD | 500 GB | 7.99 $ |
| ovhcloud | Essential | 2 GB | 40 GB NVME | Unlimited | 6.44 $ |

Figura 102 – Comparația VPS hosting-urilor

Cheltuielile lunare bazate pe server (VPS), sunt minore în comparație cu alte cheltuieli. Spre exemplu cele mai mari cheltuieli vor fi la etapa de dezvoltare și promovare – marketing. Însăși existența proiectului depinde nu doar de hosting, ci și de numărul de utilizatori, sau investitori care vor fi interesați de aceasta.

Cheltuielile bazate pe dezvoltare, implementare și testarea platformei – ar fi unele din cele mai costisitoare din punct de vedere financiar și durată. Ele depind de mai mulți factori, inclusiv dimensiunea și complexitatea proiectului, tehnologiile utilizate, timpul necesar pentru a finaliza proiectul și resursele implicate. În această categorie se include nemijlocit salariu programatorilor, testerilor, design-erilor. Da, pentru început platforma poate nu fi foarte mare, să nu aibă un foarte mare șir de funcționalități, însă cu timpul așteptările și cerințele față de proiect cresc. Numărul personalului depinde însăși de faptul cât de complexă va fi platforma. În general, aceste cheltuieli includ:

* Costul dezvoltării software-ului: Acesta este costul asociat cu angajarea dezvoltatorilor software pentru a crea platforma. Acest cost poate varia în funcție de numărul de dezvoltatori implicați, experiența lor și tarifele lor orare.
* Costul infrastructurii hardware: Acesta este costul asociat cu achiziționarea și instalarea echipamentelor hardware necesare pentru a susține platforma. Acest cost poate include servere, stocare, rețea și alte echipamente necesare pentru a asigura funcționarea adecvată a platformei.
* Costul software-ului și al licențelor: Acesta este costul asociat cu achiziționarea și utilizarea software-ului necesar pentru dezvoltarea, implementarea și testarea platformei. Acest cost poate include licențe pentru baze de date, medii de dezvoltare integrat (IDE), biblioteci și alte instrumente de dezvoltare software.
* Costul testării și asigurării calității: Acesta este costul asociat cu testarea platformei pentru a verifica funcționalitatea corectă, performanța și securitatea. Acest cost poate include testarea manuală și automată, achiziționarea și configurarea unor instrumente de testare și angajarea unor experți în testarea software-ului.
* Costul formării și instruirii utilizatorilor finali: Acesta este costul asociat cu instruirea utilizatorilor finali pentru a utiliza platforma. Acest cost poate include crearea de documentație, tutoriale și alte materiale de formare și angajarea unor specialiști în instruirea utilizatorilor.

Cheltuielile bazate pe marketing și promovare sunt esențiale pentru a asigura succesul unei platforme, deoarece acestea ajută la creșterea vizibilității și la atragerea utilizatorilor. Aceste cheltuieli pot include:

Publicitate online: Acesta este costul asociat cu plasarea de anunțuri plătite pe platforme online precum Google Ads, Facebook Ads sau LinkedIn Ads, în funcție de publicul țintă și de obiectivele de marketing.

Marketing prin email: Acesta este costul asociat cu crearea și trimiterea de campanii de email marketing către potențiali utilizatori. Aceste campanii pot include newslettere, oferte speciale și alte materiale de marketing.

Participarea la evenimente și conferințe: Acesta este costul asociat cu participarea la evenimente și conferințe relevante pentru platforma, precum târguri și expoziții. Aceasta poate include costuri de înregistrare, costuri de călătorie și de cazare și costuri de promovare la eveniment.

Marketing prin social media: Acesta este costul asociat cu crearea și gestionarea prezenței platformei pe platformele de social media, precum Facebook, Instagram sau Twitter. Acest cost poate include costul de creare a conținutului și de alocare a timpului necesar pentru gestionarea conturilor.

Relații publice și comunicare: Acesta este costul asociat cu gestionarea relațiilor cu presa și comunicarea cu publicul. Acest cost poate include costul de creare și distribuire a comunicatelor de presă și costul de a angaja o agenție de relații publice.

Este important să se evalueze cu atenție cheltuielile de marketing și promovare pentru a asigura că acestea sunt efective în atingerea obiectivelor de marketing și de afaceri. Pentru a face acest lucru, poate fi util să se creeze o strategie de marketing clară și să se monitorizeze cu atenție performanța campaniilor de marketing și promovare pentru a ajusta și optimiza bugetul alocat acestora.

În concluzie, estimarea costurilor proiectului este un proces important în planificarea și implementarea cu succes a unui proiect. Prin estimarea costurilor, se poate estima costul total al proiectului și se pot identifica și aloca resursele financiare necesare pentru a finaliza proiectul la timp și în conformitate cu specificațiile stabilite. Este important să se ia în considerare toți factorii relevanți pentru a estima costurile cât mai precis posibil, inclusiv dimensiunea și complexitatea proiectului, tehnologiile utilizate, timpul necesar pentru a finaliza proiectul și resursele implicate. De asemenea, este important să se evalueze cu atenție costurile pe durata întregului ciclu de viață al proiectului, incluzând cheltuielile cu dezvoltarea, implementarea, testarea, marketingul și promovarea, precum și costurile de întreținere și de suport după finalizarea proiectului. În plus, este important să se monitorizeze și să se actualizeze estimările costurilor pe parcursul proiectului, pentru a asigura o gestionare eficientă a resurselor financiare și pentru a lua decizii informate privind bugetul.

# CONCLUZII

Aici scrieți concluzii pentru tot raportul ( pentru fiecare capitol din raport)

**BIBLIOGRAFIE**

1. https://www.if-koubou.com/articles/how-to/what-is-github-and-what-is-it-used-for.html

2. <https://www.scritub.com/stiinta/informatica/MODELAREA-SISTEMELOR-INFORMATI12121131520.php>

3. <https://vuejs.org/guide/introduction.html>

4. <https://expressjs.com/en/guide/routing.html>

5. https://axios-http.com/docs/example

# ANEXA A

Denumire anexă

În anexă se scrie cod, tabele, imagini, informație care nu este inclusă în context, dar la care se va face referință.

Dacă în anexe vor fi figuri ele se vor nota: Figura A.1- Denumire figură.

Dacă în anexe vor fi tabele ele se vor nota: Tabelul A.1- Denumire tabel