UNIVERSITATEA TEHNICĂ„GHEORGHE ASACHI” DIN IAȘI
FACULTATEA DE AUTOMATICĂȘI CALCULATOARE
Domeniul: Calculatoare și Tehnologia Informației
Programul de studii: Tehnologia Informației

UrHomie - Platformă web distribuită destinată serviciilor de Home Maintenance

Coordonator științific:
ș.I. dr.ing. Aflori Cristian

Absolvent:
Gherasim Dragoș-George

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII PROIECTULUI DE DIPLOMĂ

Subsemnatul Gherasim Dragoş-George legitimat cu CI seria ZC nr. 451401, CNP 5001213045351 autorul lucrării UrHomie - Platformă web distribuită destinată serviciilor de Home Maintenance elaborată în vederea susţinerii examenului de finalizare a studiilor de licență, programul de studii Tehnologia Informaţiei organizat de către Facultatea de Automatică şi Calculatoare din cadrul Universităţii Tehnice „Gheorghe Asachi” din Iaşi, sesiunea iulie 2025 a anului universitar 2024-2025, luând în considerare conţinutul Art. 34 din Codul de etică universitară al Universităţii Tehnice „Gheorghe Asachi” din Iaşi (Manualul Procedurilor, UTI.POM.02 - Funcţionarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române (legea 8/1996) şi a convenţiilor internaţionale privind drepturile de autor.

Data
29.06.2025

Semnătura

Cuprins

Rezumat

Într-o societate tot mai digitalizată, utilizatorii caută soluții rapide și sigure pentru a accesa diverse servicii, inclusiv din sfera *home maintenance* (întreținere, reparații, curățenie etc.). În România, numărul platformelor specializate dedicate acestui domeniu este redus, iar cele existente nu oferă o experiență digitală suficient de personalizată, intuitivă și sigură. Motivația dezvoltării proiectului *UrHomie* derivă din această realitate și vizează realizarea unei soluții digitale adaptate contextului local, care să sprijine atât clientii, cât și micii furnizori de servicii.

Obiectivele principale presupun dezvoltarea unei platforme web moderne, fiabile și ușor de utilizat, care facilitează accesul utilizatorilor la servicii specializate și oferă vizibilitate online furnizorilor independenți. *UrHomie* se bazează pe o arhitectură distribuită, compusă din microservicii dezvoltate în *C#*, *Python* și *Java*, utilizând framework-uri precum *ASP.NET Core Web API*, *Spring Boot* și *FastAPI*. Comunicarea între componente este orchestrată printr-o combinație de tehnologii moderne, adaptate fiecărui scenariu specific: *WebSocket* pentru interacțiuni bidirecționale în timp real, *gRPC* pentru apeluri rapide și eficiente între microservicii, *REST* pentru expunerea convențională a resurselor web și *RabbitMQ* pentru schimbul asincron de mesaje, asigurând decuplarea și reziliența componentelor.

Platforma integrează baze de date relaționale (*MariaDB*) și non-relaționale (*MongoDB*), oferind flexibilitate în stocarea și procesarea datelor. Pentru containerizare se utilizează *Docker*, iar rutarea și conversia protocoalelor este asigurată prin *NGINX* și *Envoy*. Platforma oferă funcționalități esențiale, precum: înregistrarea utilizatorilor, autentificare și autorizare pe baza token-urilor *JWT*, gestionarea completă a profilului (pentru clienți și furnizori de servicii), căutarea serviciilor de tip *home maintenance*, adăugarea de servicii de către furnizori, gestionarea celor existente și un sistem de programare a serviciilor, prin care clienții pot face rezervări.

Prin arhitectura sa modulară și scalabilă, *UrHomie* are potențialul de a deveni un ecosistem digital complet dedicat serviciilor de *home maintenance*.

Cuvinte-cheie: Digitalizare, Home Maintenance, Platformă Web Distribuită, Microservicii, REST API.

Capitolul 1. Introducere

1.1. Contextul și importanța temei alese.

Într-o societate în continuă dezvoltare tehnologică, accesul rapid și sigur la diverse servicii, inclusiv cele de *home maintenance*, devine o necesitate din ce în ce mai accentuată. În România, oferta de platforme specializate în acest domeniu este limitată, iar cele existente nu reușesc să ofere o experiență digitală personalizată, intuitivă și sigură. Această situație evidențiază o oportunitate semnificativă pentru dezvoltarea unor soluții adaptate contextului local, care să sprijine atât clienții, cât și micii furnizori de servicii.

Digitalizarea serviciilor de întreținere a locuinței poate aduce multiple beneficii, precum eficientizarea proceselor, reducerea timpului de așteptare și creșterea transparenței în relația dintre clienți și furnizori. Totodată, o astfel de platformă poate contribui la formalizarea activităților desfășurate de furnizorii independenți, oferindu-le acestora o vizibilitate mai mare și acces la o bază mai largă de clienți. Prin urmare, dezvoltarea unei platforme web în acest domeniu răspunde unei nevoi reale a pieței și sprijină incluziunea digitală și dezvoltarea economică locală [1].

1.2. Obiectivele generale și specifice ale lucrării

Lucrarea de față își propune să contribuie la digitalizarea serviciilor de *home maintenance* în România, prin dezvoltarea unei platforme web inovatoare care să faciliteze conexiunea între clienți și furnizori de servicii. Această inițiativă răspunde unei nevoi reale identificate pe piața locală, unde oferta de platforme specializate este limitată, iar cele existente nu oferă o experiență digitală suficient de personalizată și sigură.

1.2.1. Obiectiv general

Dezvoltarea unei platforme web scalabile și securizate, bazată pe sisteme distribuite și arhitectură pe microservicii, care să permită interacțiunea eficientă între clienți și furnizori de servicii de întreținere a locuinței, oferind o experiență digitală intuitivă și personalizată.

1.2.2. Obiective specifice

1. Identificarea și documentarea nevoilor clienților și a furnizorilor de servicii, pentru a asigura o aliniere optimă între funcționalitățile platformei și așteptările acestora;

2. Elaborarea unei arhitecturi distribuite, bazată pe microservicii, care să asigure modularitate, scalabilitate și ușurință în mentenanță;

3. Implementarea serviciilor individuale în cadrul arhitecturii distribuite, utilizând tehnologii moderne și mecanisme de comunicare eficiente între componente;

4. Integrarea unei soluții hibride de stocare a datelor, care să combine baze de date relaționale și non-relaționale, în funcție de specificul fiecărei componente;

5. Dezvoltarea unui sistem securizat de autentificare și autorizare, bazat pe token-uri, care să asigure protecția datelor și a sesiunilor utilizatorilor;

6. Crearea unei interfețe web moderne, „prietenoase” și adaptabile la diferite dispozitive, care să ofere o experiență accesibilă și intuitivă pentru utilizatori [2];

7. Realizarea testelor unitare și de integrare pentru a asigura funcționarea corectă a componentelor și a întregului sistem;

8. Evaluarea performanței aplicației în scenarii relevante de utilizare, cu scopul de a identifica posibile optimizări;

9. Redactarea documentației tehnice și funcționale a platformei, pentru a facilita întreținerea, utilizarea și extinderea ulterioară.

1.3. Metodologia de cercetare utilizată

În etapa inițială a proiectului, am realizat o documentare amplă axată pe identificarea celor mai bune practici în proiectarea arhitecturilor scalabile, utilizând surse relevante, actualizate și adaptate cerințelor curente ale industriei IT. În acest sens, am analizat mai multe proiecte open-source disponibile pe platforma *GitHub* [3], cu precădere cele care utilizează arhitecturi bazate pe microservicii și tehnologii moderne. De asemenea, m-am documentat din resurse valide, precum articole de specialitate, cărți în format digital și documentații oficiale, pentru a înțelege în profunzime principiile de proiectare, avantajele și provocările asociate unei astfel de arhitecturi.

Pe tot parcursul proiectului, am menținut o colaborare strânsă cu domnul profesor îndrumător, discutând periodic despre progresul realizat și deciziile tehnice adoptate. Feedback-ul oferit de acesta a fost esențial în validarea alegerilor stack-ului și în orientarea dezvoltării platformei, colaborarea noastră asigurând alinierea proiectului la cerințele academice și la nevoile reale ale utilizatorilor.

1.4. Structura generală a lucrării

Lucrarea de față este structurată în mod logic și progresiv, astfel încât să reflecte analiza teoretică și implementarea practică a platformei *UrHomie*. Primul capitol, *Introducere*, oferă contextul și motivația alegerii temei, definind obiectivele urmărite și metodologia de cercetare adoptată. În continuare, capitolul *Fundamentarea teoretică și soluții similare* explorează conceptele relevante pentru arhitectura sistemelor distribuite, designul orientat pe microservicii și experiența utilizatorului (*UI/UX*), punând accent și pe soluțiile deja existente în domeniul *home maintenance*, identificând limitările și oportunitățile neacoperite.

Capitolul central, *Soluția propusă*, constituie nucleul lucrării și detaliază procesul de proiectare și implementare al aplicației, de la cerințele utilizatorilor până la alegerea tehnologiilor și arhitectura sistemului. Sunt incluse modelarea bazelor de date, descrierea componentelor software dezvoltate, justificarea deciziilor tehnice și capturi de ecran din aplicația funcțională. Urmează *Testarea soluției și rezultate experimentale*, unde sunt prezentate metodele de validare aplicate sistemului, inclusiv testarea funcțională, de performanță și scenarii de utilizare, susținute prin tabele, grafice și interpretări.

Ultima parte a lucrării cuprinde *Concluziile*, unde sunt sintetizate contribuțiile personale, impactul soluției dezvoltate și posibile direcții de extindere viitoare. În încheiere, *Bibliografia* conține sursele academice și tehnice consultate, iar *Anexele* includ diagrame *UML*, fragmente de cod relevante și alte resurse care susțin înțelegerea și replicabilitatea proiectului.

Capitolul 2. Fundamentarea teoretică și soluții similare

2.1. Fundamente teoretice și concepte cheie privind platformele de servicii pentru întreținerea locuinței

În contextul dezvoltării platformei *UrHomie*, adoptarea unei arhitecturi web distribuite reprezintă o alegere strategică, menită să asigure scalabilitate, reziliență și performanță în gestionarea relațiilor dintre clienți și furnizorii de servicii de *home maintenance*. O astfel de arhitectură permite distribuirea sarcinilor între multiple componente autonome, facilitând adaptarea rapidă la cerințele variate ale utilizatorilor și la volumul fluctuant de solicitări.

Unul dintre avantajele majore ale sistemelor distribuite este capacitatea de scalare orizontală, prin adăugarea de noi noduri sau servicii pentru a gestiona creșterea cererii, fără a compromite performanța generală a sistemului [4]. Această flexibilitate este esențială pentru platforme precum *UrHomie*, care trebuie să răspundă eficient atât nevoilor clienților, cât și ale furnizorilor. De asemenea, arhitectura distribuită oferă toleranță la erori, asigurând continuitatea serviciilor chiar și în cazul unor defecțiuni ale anumitor componente.

2.1.1. Arhitectura bazată pe microservicii

Arhitectura bazată pe microservicii reprezintă o abordare modernă în dezvoltarea aplicațiilor, în care sistemul este divizat în componente mici, autonome și independente. Principiile fundamentale ale microserviciilor includ [5]:

- *Autonomy*, mai exact fiecare serviciu funcționează independent, având propriul ciclu de viață și propriile date;
- *Loose Coupling*, deoarece microserviciile interacționează între ele prin intermediul unor interfețe bine definite, reducând dependențele și facilitând modificările locale fără a afecta întregul sistem;
- *Fault Tolerance*, prin faptul că sistemul este proiectat astfel încât eșecurile unui microserviciu să nu afecteze funcționarea celorlalte componente;
- *Composability*, ceea ce înseamnă că microserviciile pot fi combinate pentru a crea funcționalități mai complexe, permițând reutilizarea componentelor existente.

2.1.2. Principiile SOLID în proiectarea software

Principiile *SOLID* reprezintă un set de cinci reguli fundamentale în programarea orientată pe obiecte, menite să îmbunătățească calitatea codului și să faciliteze întreținerea și extensibilitatea aplicațiilor. Aceste principii sunt [6]:

- *S (Single Responsibility Principle)* - o clasă ar trebui să fie responsabilă pentru un singur aspect al funcționalității sistemului;
- *O (Open/Closed Principle)* - entitățile software, precum clasele, modulele sau funcțiile, ar trebui concepute astfel încât să poată fi extinse cu noi comportamente, fără a fi necesară modificarea codului existent;
- *L (Liskov Substitution Principle)* - clasele derivate trebuie să poată fi utilizate în locul claselor de bază fără a compromite funcționalitatea sau comportamentul corect al programului;

- *I (Interface Segregation Principle)* - este recomandat ca interfețele să fie cât mai specifice scopului lor, astfel încât fiecare modul să depindă doar de metodele de care are nevoie, evitându-se interfețele generale care conțin funcționalități irelevante pentru anumite componente;

- *D (Dependency Inversion Principle)* - modulele de nivel înalt nu ar trebui să depindă de modulele de nivel jos, ambele ar trebui să depindă de abstracții.

2.1.3. reDesign Patterns în dezvoltarea software

Design patterns sunt soluții general acceptate pentru probleme recurente în dezvoltarea software ce oferă un cadru standardizat care facilitează comunicarea între dezvoltatori și promovează reutilizarea codului. În dezvoltarea platformei *UrHomie*, aplicarea unor *design patterns* consacrate a fost esențială pentru asigurarea unei arhitecturi flexibile și scalabile. Printre aceste șabloane de proiectare se numără:

- *Dependency Injection (DI),* un *pattern* care promovează inversarea controlului prin injectarea dependențelor unei clase din exterior, reducând astfel cuplajul între componente și facilitând testarea și întreținerea codului;

- *Event-Driven Architecture (EDA*), un model arhitectural în care componentele comunică asincron prin evenimente transmise prin cozi de mesaje, facilitând decuplarea, scalabilitatea și reactivitatea sistemului;

- *Repository Pattern*, reprezintă un șablon structural care oferă o abstractizare a accesului la date, separând logica de acces la date de logica de business a aplicației, ceea ce contribuie la o arhitectură mai curată și modulară;

- *Saga Pattern*, utilizat pentru gestionarea tranzacțiilor distribuite în arhitecturile bazate pe microservicii, *pattern* ce permite menținerea consistenței datelor prin definirea unei serii de tranzacții locale, fiecare cu o acțiune de compensare în caz de eșec.

2.1.4. Stilul arhitectural REST

REST (Representational State Transfer) este un stil arhitectural pentru dezvoltarea serviciilor web, care se bazează pe un set de constrângeri și principii pentru a crea sisteme scalabile și ușor de întreținut. Principiile fundamentale ale *REST* includ [7]:

- *Statelessness*, adică fiecare cerere de la client către server trebuie să conțină toate informațiile necesare pentru a înțelege și procesa cererea;

- *Uniform Interface*, ce presupune ideea că se utilizează o interfață uniformă pentru a interacționa cu resursele, de obicei prin metodele *HTTP* standard (*GET, POST, PUT, PATCH, DELETE*);

- *Client-Server Architecture,* ce vizează separarea clară între partea de client (interfața utilizatorului) și partea de server (logica de procesare și stocare a datelor), permițând dezvoltarea, întreținerea și scalarea independentă a fiecărei părți.

- *Cacheability*, deoarece răspunsurile trebuie să fie explicit marcate ca fiind *cacheable* sau *non-cacheable* pentru a îmbunătăți performanța.

Toate conceptele teoretice descrise anterior au fost aplicate concret în dezvoltarea platformei *UrHomie*. Arhitectura distribuită și modelul bazat pe microservicii au stat la baza organizării aplicației în patru componente independente (*autentificare*, *user management*, *catalog servicii*, *rezervări*), fiecare rulând autonom în containere dedicate și comunicând prin protocoale

standardizate (*REST, gRPC, WebSocket*). Principiile *SOLID* au fost urmate în proiectarea claselor și serviciilor interne, în special prin separarea responsabilităților (*SRP*), folosirea interfețelor specializate și injectarea dependențelor. De asemenea, *design patterns* precum *Dependency Injection, Repository Pattern* și *Event-Driven Architecture* au fost utilizate în mod practic pentru a reduce cuplajul, a izola accesul la date și a permite comunicarea asincronă între componente prin *RabbitMQ*. Consumul de resurse prin *REST* a fost standardizat conform principiilor *RESTful* (*stateless, cacheable, uniform interface*), iar pentru operațiile critice distribuite, a fost implementat un mecanism de tip *Saga*, care asigură consistența tranzacțiilor între microserviciile implicate.

2.2. Evaluarea abordărilor actuale din literatura de specialitate

Pentru a înțelege mai bine contextul și direcțiile actuale în dezvoltarea platformelor de gestionare a serviciilor de *home maintenance,* este importantă o analiză a cercetărilor academice și studiilor relevante din domeniu. Aceast lucru permite evidențierea tendințelor emergente, a tehnologiilor frecvent utilizate și a principalelor provocări întâlnite în proiectarea acestor soluții.

Un exemplu de referință este aplicația *Home Repairs* [8], dezvoltată de cercetători de la *Middle East College*. Aceasta oferă o gamă largă de servicii de întreținere, precum reparații electrice, montaj de mobilier și curățenie. Aplicația permite utilizatorilor să selecteze serviciile dorite, să specifice preferințele și să evalueze calitatea serviciilor primite, facilitând astfel o interacțiune eficientă între clienți și furnizori. Studiul evidențiază importanța aplicațiilor mobile în facilitarea accesului la servicii de întreținere și în îmbunătățirea experienței utilizatorilor.

2.3. Analiza soluțiilor/aplicațiilor existente din perspectiva temei alese

Acest subcapitol vizează o analiză a platformelor existente în domeniul serviciilor de întreținere a locuinței, atât la nivel național, cât și internațional, cu scopul de a evidenția inovațiile propuse de acestea, precum și limitările întâmpinate în practică.

În contextul platformelor naționale dedicate întreținerii locuinței, *Home Maintenance România* [9] se remarcă drept o companie specializată în furnizarea de servicii de mentenanță pentru imobile și cartiere rezidențiale. Printre serviciile oferite se numără amenajările peisagistice, instalarea sistemelor de irigații automatizate, precum și lucrările de construcții civile. O altă inițiativă relevantă este *Servicii24.ro* [10], o platformă care funcționează ca un marketplace digital, facilitând conexiunea directă între furnizorii de servicii și clienți. Aceasta oferă un cadru eficient pentru identificarea și contractarea rapidă a specialiștilor în diverse domenii de întreținere și reparații.

Pe plan internațional**,** *TaskRabbit* [13] este una dintre cele mai cunoscute platforme online care conectează utilizatorii cu furnizori locali pentru o gamă variată de servicii. Printre acestea se numără reparațiile casnice, montajul de mobilier și serviciile de curățenie. Utilizatorii pot alege furnizorii în funcție de criterii precum prețul, recenziile altor clienți și disponibilitatea, toate operațiunile fiind gestionate facil prin intermediul aplicației dedicate. O altă platformă relevantă este *Handy* [14], care oferă servicii de curățenie și reparații pentru locuințe, punând accent pe accesibilitatea și eficiența procesului de rezervare. Utilizatorii pot programa rapid intervențiile unor profesioniști verificați pentru sarcini diverse, de la montarea televizoarelor până la instalarea corpurilor de iluminat. Platforma se distinge printr-un sistem de rezervare simplificat și prin opțiunea de plată securizată online.

2.4. Lacune identificate în stadiul actual de dezvoltare și în soluțiile existente

O analiză critică a limitărilor și provocărilor cu care se confruntă platformele și aplicațiile actuale din domeniul serviciilor de întreținere a locuinței se dovedește necesară pentru evidențierea nevoii de inovație. Această evaluare fundamentează justificarea dezvoltării unei soluții noi, capabile să răspundă mai eficient cerințelor utilizatorilor.

Analiza platformelor locale existente în domeniul serviciilor de întreținere a locuinței relevă o serie de limitări importante. *Home Maintenance România*, de exemplu, oferă servicii variate în acest sector, însă prezentarea lor se realizează într-un mod tradițional, lipsind o platformă digitală interactivă care să faciliteze comunicarea directă și eficientă între clienți și furnizori. Pe de altă parte, *Servicii24.ro*, deși acoperă o gamă extinsă de servicii, inclusiv cele legate de întreținerea locuinței, nu reușește să ofere o experiență digitală complet personalizată. Platforma nu dispune de funcționalități avansate, precum gestionarea programărilor sau urmărirea în timp real a progresului serviciilor, ceea ce limitează nivelul de interacțiune și control al utilizatorilor.

2.5. Conturarea cerințelor aplicației în raport cu nevoile identificate

2.5.1. Cerințe funcționale

Un prim set de cerințe funcționale vizează procesul de gestionare a utilizatorilor, începând cu etapa de înregistrare și autentificare. Utilizatorii se pot înregistra în platformă prin completarea unui formular care solicită date personale de bază, precum adresa de e-mail, o parolă securizată, numărul de telefon, adresa de domiciliu, etc. În cazul în care utilizatorul optează pentru rolul de furnizor de servicii, este necesară completarea unor câmpuri suplimentare, precum nivelul de educație, certificările relevante, o scurtă descriere a experienței profesionale, etc. Odată finalizat procesul de înregistrare, autentificarea se realizează prin introducerea adresei de e-mail și a parolei, iar utilizatorul este redirecționat automat către pagina principală corespunzătoare rolului său.

După autentificare, platforma oferă funcționalități adaptate rolului fiecărui utilizator. Atât clienții, cât și furnizorii de servicii au acces la o secțiune de profil personal, unde pot vizualiza și actualiza informațiile proprii. În plus, clienții beneficiază de o interfață intuitivă care le permite să efectueze o căutare avansată a serviciilor disponibile, prin intermediul unui câmp de tip *search bar*. Acesta oferă potriviri contextuale, identificând servicii relevante pe baza cuvintelor-cheie introduse, indiferent dacă acestea apar în titlu, descriere, domeniu sau locație. Furnizorii de servicii, în schimb, au la dispoziție un *dashboard* specializat, prin intermediul căruia pot adăuga, modifica sau elimina serviciile oferite, pot vizualiza rezervările primite de la clienți și pot gestiona întregul flux de programare – inclusiv acceptarea, respingerea sau marcarea unei rezervări ca finalizată. Această diferențiere a funcționalităților, în funcție de rol, contribuie la o experiență personalizată și eficientă pentru fiecare tip de utilizator.

Nu în ultimul rând, aplicația va pune accent pe o interfață personalizată și prietenoasă cu utilizatorul. Navigarea va fi intuitivă, cu meniuri clar structurate și acces rapid la funcționalitățile principale. Elemente vizuale coerente, cum ar fi o paletă cromatică echilibrată, fonturi lizibile și iconuri sugestive, vor contribui la orientarea ușoară a utilizatorului. De asemenea, aplicația va furniza feedback vizual pentru acțiunile întreprinse și va păstra utilizatorul conectat pentru o perioadă extinsă, facilitând astfel o experiență de utilizare continuă și fără întreruperi frecvente de autentificare.

2.5.2. Cerințe tehnice

Pentru a garanta o experiență optimă, aplicația trebuie să îndeplinească o serie de cerințe tehnice esențiale. În ceea ce privește performanța și scalabilitatea, sistemul trebuie să asigure timpi de răspuns rapizi pentru toate operațiunile, inclusiv în condiții de trafic intens sau de utilizare simultană de către un număr mare de utilizatori. Arhitectura tehnică va fi proiectată astfel încât să permită scalarea orizontală, oferind posibilitatea extinderii capacității aplicației fără a compromite

performanța.

Din perspectiva securității, aplicația va integra mecanisme solide pentru protejarea datelor sensibile ale utilizatorilor, precum criptarea parolelor și validarea riguroasă a datelor introduse. Autentificarea și autorizarea se vor baza pe protocoale robuste, menite să prevină accesul neautorizat și eventualele breșe de securitate. Disponibilitatea și fiabilitatea sistemului sunt, de asemenea, esențiale. Aplicația va trebui să funcționeze cu un timp de disponibilitate de cel puțin 99.9%, reducând la minimum perioadele de indisponibilitate. În plus, vor fi implementate strategii de backup și mecanisme eficiente de recuperare în caz de dezastru, astfel încât pierderile de date sau funcționalitate să fie prevenite sau remediate rapid. Pentru a susține administrarea eficientă a unei platforme distribuite, este necesară implementarea unor mecanisme de observabilitate, care să permită monitorizarea continuă a comportamentului sistemului. Acestea includ logarea evenimentelor relevante, colectarea de metrici de performanță și trasabilitatea tranzacțiilor între microservicii. Astfel, posibilele probleme pot fi detectate și diagnosticate rapid, contribuind la menținerea unui nivel ridicat de fiabilitate și la optimizarea permanentă a aplicației.

Capitolul 3. Soluția propusă

3.1. Reiterarea problemei și a strategiei de rezolvare

În contextul digital actual, intermedierea eficientă între cerere și ofertă în domenii diverse, de la servicii de curățenie la activități profesionale specializate, este esențială pentru a răspunde rapid și adecvat nevoilor utilizatorilor. Cu toate acestea, multe dintre platformele actuale dedicate serviciilor de *home maintenance* se confruntă cu limitări precum lipsa unei interfețe moderne și interactive, personalizare redusă a experienței utilizatorului și integrare deficitară cu tehnologii contemporane.

Strategia de rezolvare propusă urmărește dezvoltarea unei soluții web moderne, care să răspundă cerințelor funcționale esențiale privind gestionarea utilizatorilor, personalizarea interfeței și facilitarea interacțiunii dintre clienți și furnizorii de servicii. Arhitectura distribuită adoptată permite separarea clară a responsabilităților între componentele aplicației și susține integrarea unor funcționalități avansate precum înregistrarea diferențiată în funcție de rol, căutarea contextuală a serviciilor, gestionarea rezervărilor și menținerea unei sesiuni active. Prin orientarea pe modularitate, ușurință în utilizare și eficiență operațională, soluția propusă își propune să ofere o experiență adaptată nevoilor reale ale utilizatorilor, într-un mediu sigur și intuitiv.

3.2. Idei originale, soluții noi

Deși soluția propusă nu introduce idei originale sau noi în materie de arhitectură software, alegerea unor tehnologii moderne și a unor mecanisme avansate contribuie semnificativ la diferențierea platformei *UrHomie* în raport cu aplicații similare. Accentul cade pe integrarea unor soluții deja validate în industrie, adaptate însă specificului platformei, astfel încât să asigure modularitate, scalabilitate, reziliență și o experiență de utilizare optimă. Aceste decizii arhitecturale și tehnice influențează în mod direct calitatea și performanța funcționalităților implementate.

Un exemplu concret în acest sens este alegerea unui microserviciu dedicat pentru autentificare, autorizare și înregistrare, implementat ca server *gRPC*, ceea ce oferă avantajul unei comunicări rapide și eficiente între componentele *backend*. Pentru a permite integrarea acestui serviciu într-un context web, platforma utilizează un *proxy* compatibil *gRPC-Web* (precum *Envoy*), care intermediază comunicarea între *frontend*-ul bazat pe browser și microserviciul *gRPC*.

În aceeași direcție, o altă decizie arhitecturală relevantă este implementarea unui mecanism asincron de orchestrare a procesului de înregistrare, bazat pe modelul *Saga*. Acesta permite coordonarea tranzacțiilor distribuite între microserviciile implicate, oferind posibilitatea de compensare în cazul unor eșecuri și menținând astfel consistența datelor. În plus, pentru a elimina dependența de interogări recurente (*polling*), practică ineficientă în sisteme distribuite, platforma integrează un mecanism de notificare în timp real prin *WebSocket*, prin care *frontend*-ul este informat imediat asupra rezultatului unei operațiuni complexe, cum ar fi finalizarea procesului de înregistrare sau apariția unei erori.

3.3. Cerințele utilizatorului

Pentru definirea direcției de dezvoltare a platformei *UrHomie*, a fost realizată o analiză a nevoilor și așteptărilor utilizatorilor finali – atât clienți, cât și furnizori de servicii. Aceste cerințe au fost formulate în urma evaluării limitărilor soluțiilor existente și a tendințelor actuale în interacțiunea utilizator-software. Scopul este de a crea o aplicație care oferă funcționalitate și o experiență coerentă, accesibilă și relevantă.

Cerințe ale clienților (consumatori de servicii):

- Posibilitatea de a crea rapid un cont și de a se autentifica în siguranță;

- Acces la un catalog organizat de servicii, cu opțiuni clare de filtrare și căutare;
- Acces la informații detaliate despre fiecare serviciu și furnizor (preț, durată, recenzii);
- Posibilitatea de a rezerva online un serviciu într-un interval convenabil;
- Confirmări clare și notificări despre starea solicitărilor și rezervărilor;
- Modalități simple și sigure de plată;
- Acces la un istoric al comenzilor și posibilitatea de a evalua experiențele anterioare;
- O interfață simplă și prietenoasă, adaptată și dispozitivelor mobile.

Cerințe ale furnizorilor de servicii:

- Posibilitatea de a crea un cont specializat și de a furniza informații complete despre serviciile oferite;
- Un spațiu dedicat pentru administrarea serviciilor disponibile (adăugare, modificare, ștergere);
- Vizualizarea și gestionarea comenzilor primite, inclusiv posibilitatea de a accepta/refuza programări;
- Acces la evaluările primite și la statistici despre activitatea proprie;
- Protecția datelor proprii și transparență în relația cu clienții.

Cerințe generale pentru toți utilizatorii:

- Interacțiune intuitivă și feedback vizual imediat pentru fiecare acțiune efectuată;
- Timp redus de așteptare pentru răspunsurile din aplicație;
- Siguranța datelor personale și confidențialitatea tranzacțiilor.

3.4. Arhitectura sistemului

Arhitectura platformei *UrHomie* este de tip distribuit și este structurată în jurul unui model orientat pe microservicii, fiecare responsabil pentru un set bine delimitat de funcționalități. La nivelul interfeței, aplicația utilizează un client web dezvoltat în *React*, care comunică cu microserviciile *backend* prin intermediul unui *reverse proxy*, *NGINX*. Pentru a facilita integrarea apelurilor *gRPC,* clientul dispune de suport pentru *gRPC-Web*, ceea ce înseamnă că transmite cererile sub forma unor mesaje codificate cu *application/grpc-web* prin protocolul *HTTP/1.1*. Aceste cereri sunt interceptate de *proxy-ul Envoy*, care se ocupă de conversia lor în *HTTP/2*, protocolul nativ al *gRPC*, și, la nevoie, face conversia inversă pentru a răspunde clientului într-un format compatibil cu limitările browserelor actuale. În plus, pentru a evita mecanismele de tip *polling*, aplicația utilizează un canal *WebSocket* prin care notificările sunt transmise în timp real către *frontend*, oferind astfel utilizatorului un feedback imediat asupra statusului acțiunilor efectuate. Arhitectura sistemului este reprezentată în diagrama de mai jos, care evidențiază interacțiunile directe dintre componente și fluxurile de date corespunzătoare:

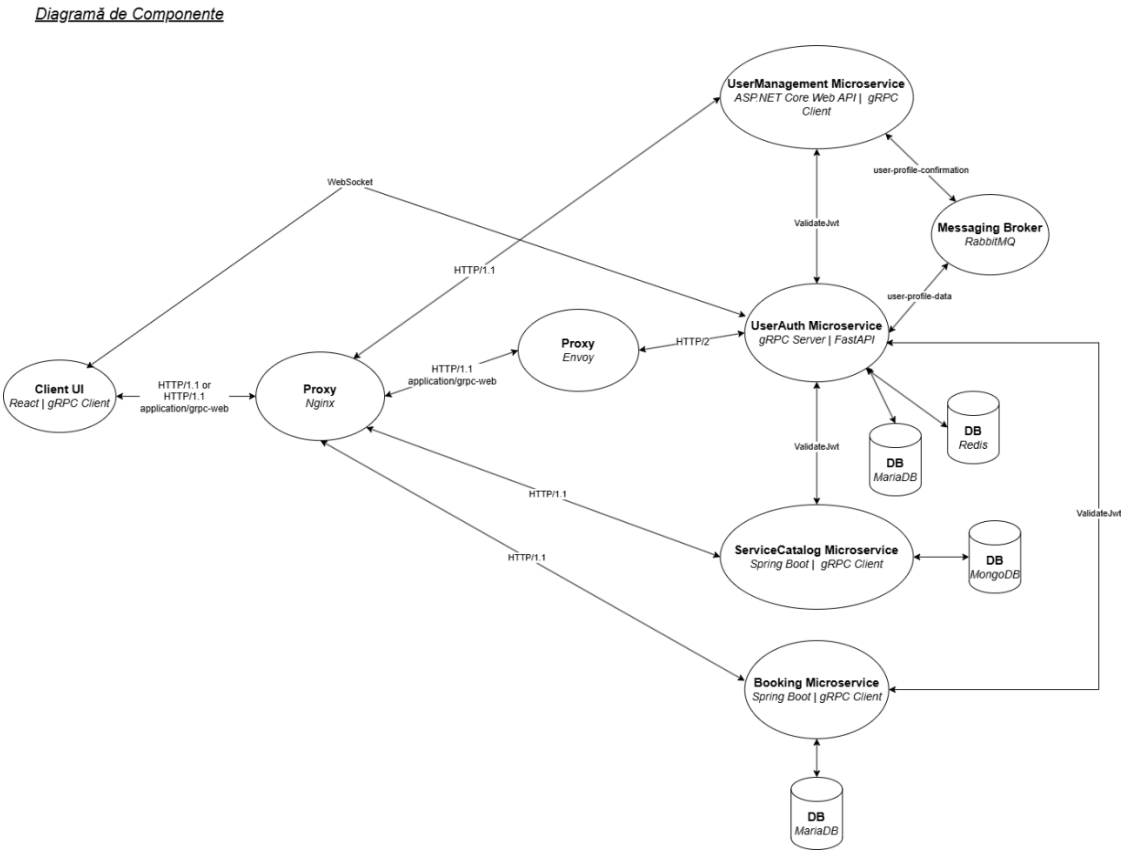


Figura 1: Diagramă logică a componentelor din platforma UrHomie

Backend-ul este compus din patru microservicii principale: *UserAuth*, *UserManagement*, *ServiceCatalog* și *Booking*, fiecare construit folosind tehnologii adecvate cerințelor funcționale specifice. Microserviciul *UserAuth*, implementat în *Python* folosind *FastAPI* și *gRPC*, se ocupă de autentificare, autorizare și validarea token-urilor *JWT*. Acesta comunică direct cu o bază de date relațională (*MariaDB*) pentru persistență și utilizează *Redis* pentru stocări temporare în procesele asincrone. Validarea token-urilor emise este accesată de celelalte microservicii, consolidând mecanismul de securitate centralizat. *UserManagement*, dezvoltat în *ASP.NET Core Web API*, este responsabil pentru gestionarea profilului utilizatorilor și interacționează cu *UserAuth* prin mesaje transmise asincron prin *RabbitMQ*, în cadrul unui model de tip *Saga*.

În paralel, *ServiceCatalog* și *Booking* sunt microservicii *Spring Boot* care gestionează serviciile disponibile și respectiv programările efectuate de clienți. *ServiceCatalog* persistă datele în *MongoDB*, reflectând natura flexibilă și variabilă a informațiilor asociate serviciilor oferite, în timp ce *Booking* utilizează *MariaDB* pentru a gestiona structuri relaționale mai rigide. Pentru a asigura securitatea *request*-urilor către aceste microservicii, fiecare endpoint este protejat prin *JWT*, iar validarea token-urilor este delegată către microserviciul *UserAuth*, fiecare componentă acționând ca un client *gRPC* pentru a efectua apelul *ValidateJwt*.

3.4.1. Componenta Frontend

Interfața utilizator (*frontend*-ul) din cadrul platformei *UrHomie* este dezvoltată sub forma unei aplicații single-page, construită utilizând biblioteca *React*. Aceasta permite o dezvoltare modulară și rapidă a componentelor, fiind ideală pentru aplicații web moderne care necesită interactivitate crescută și integrare strânsă cu servicii *backend*.

3.4.1.1. Structura pe Componente

Aplicația *frontend UrHomie* este structurată în jurul unor rute și componente majore, fiecare asociată unui set specific de funcționalități adaptate rolului utilizatorului (*Client* sau *Service Provider*). Arhitectura este bazată pe o organizare modulară, cu layout-uri și navigare diferențiată pe roluri.

- *Landing page*, pagina publică principală, destinată utilizatorilor neautentificați, de unde pot accesa formularele de *login* și *înregistrare*. Deși ruta implicită a aplicației este */home*, utilizatorii care nu sunt autentificați și încearcă să acceseze funcționalități protejate vor fi redirecționați automat către această pagină de întâmpinare;

- *Login Page,* gestionat printr-un formular simplu, securizat prin validări locale și apeluri *gRPC* către microserviciul de autentificare;

- *Register Forms*, este structurată în doi pași: mai întâi completarea informațiilor generale (pentru toți utilizatorii), apoi completarea detaliilor suplimentare pentru furnizorii de servicii (educație, certificări, experiență etc.);

- *Home Page*, după autentificare, utilizatorul este redirecționat către pagina *Home*, personalizată în funcție de rol. Clienții au acces la o interfață de explorare a serviciilor disponibile, inclusiv un *search bar* avansat, care returnează rezultate relevante în funcție de titlu, descriere, locație sau domeniu. Pentru furnizorii de servicii, *Home Page* are un conținut diferit: afișează o vizualizare sumarizată a serviciilor proprii, alături de acțiuni rapide precum „*Add new service*” sau „*View all*”. Atunci când furnizorul alege una dintre opțiunile specifice rolului în *navigation bar*, este redirecționat către *Dashboard*, unde are acces la o interfață completă pentru gestionarea activității sale;

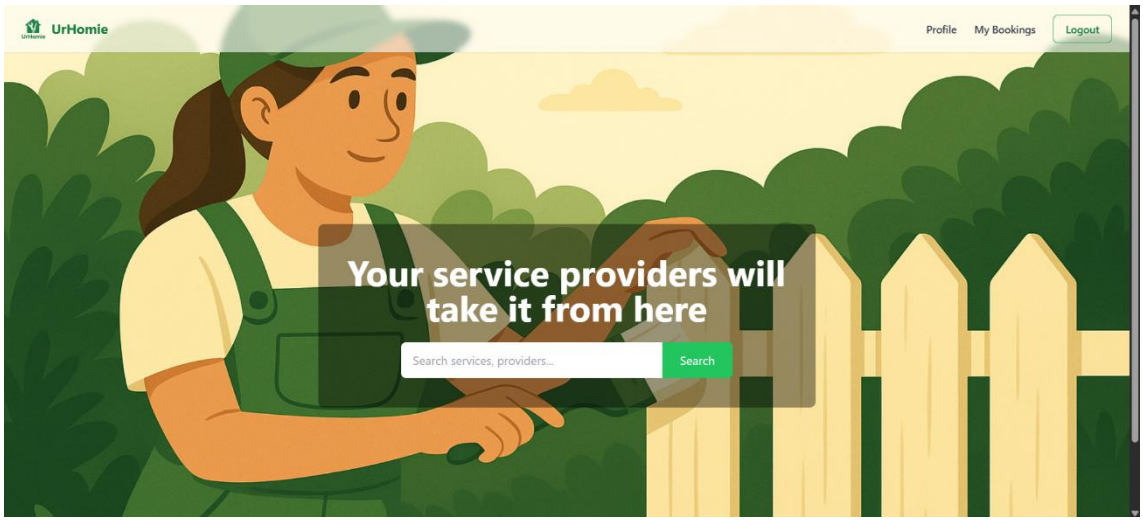


Figura 2: Home Page din perspectiva unui client autentificat

- *Search Results Page*, accesibilă clienților și afișează serviciile care corespund cuvintelelor-cheie introduse în bara de căutare. Rezultatele sunt prezentate în format specific, cu posibilitate de paginare, iar în lipsa potrivirilor se oferă feedback vizual;

- *Profile Page*, secțiune protejată, accesibilă doar utilizatorilor autentificați, unde aceștia pot vizualiza și edita datele personale. Structura formularului se adaptează rolului utilizatorului, permițând actualizarea câmpurilor relevante pentru fiecare categorie;

- *My Bookings Page*, se pot vizualiza rezervările efectuate, din perspectiva clientului. Fiecare înregistrare include detalii despre serviciu, statusul actual, data programării și un *modal* pentru afișarea detaliilor suplimentare. În funcție de status, clientul poate efectua acțiuni precum anularea rezervării, disponibilă pentru stările ***PENDING*** și ***CONFIRMED***. Anularea se face printr-un formular tip *modal*, unde utilizatorul introduce un mesaj justificativ, care va fi ulterior vizibil în detaliile rezervării.

- *Dashboard*, furnizorii de servicii autentificați beneficiază de un *dashboard* dedicat, care le oferă acces la funcționalitățile necesare pentru gestionarea completă a activității din platformă. Navigarea este susținută de un layout si un *navigation bar* specific, adaptat acestui rol. Fiecare secțiune este organizată în jurul unei funcționalități cheie, descrise mai jos:

    - *My Services Page,* afișează lista tuturor serviciilor create de furnizor, sub formă de carduri ce includ informații precum titlul, prețul, locația și durata estimată. Selectarea unui serviciu de interes redirecționează utilizatorul către *Service Details Page*;

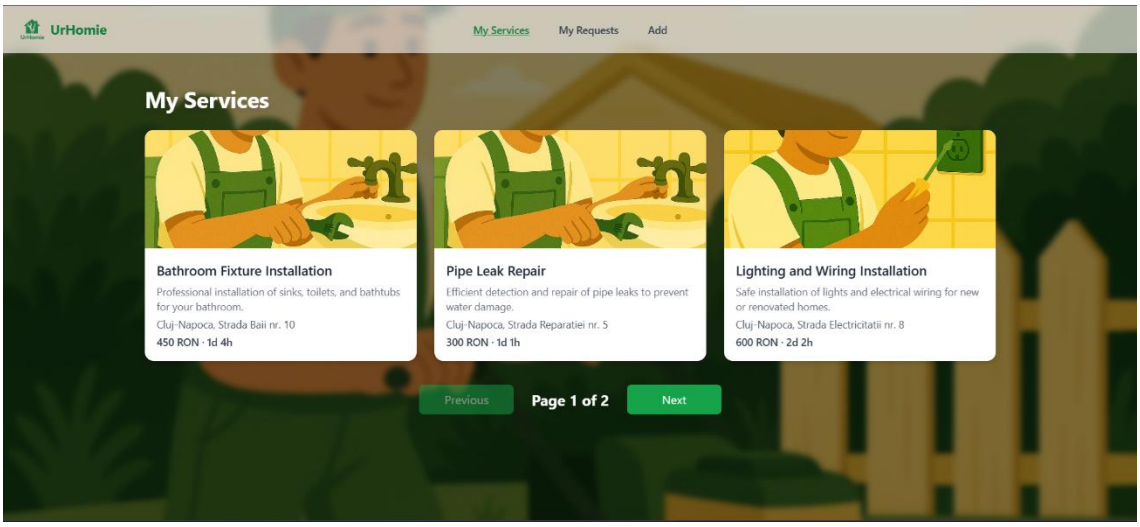


Figura 3: My Services Page

    - *My Requests Page,* se pot vizualiza cererile de rezervare primite de la clienți, afișate sub formă de carduri cu informații esențiale despre serviciul rezervat, client, locație și statusul curent. Furnizorul poate întreprinde acțiuni în funcție de starea rezervării: în cazul cererilor ***PENDING***, are posibilitatea de a accepta sau respinge solicitarea, cea din urmă fiind însoțită de un *modal* dedicat pentru introducerea unui mesaj justificativ. Pentru cererile ***CONFIRMED***, este disponibilă acțiunea de marcarea ca finalizată, iar toate cererile, indiferent de status, pot fi vizualizate în detaliu printr-un *modal* complet, care include și motivul anulării dacă este cazul;

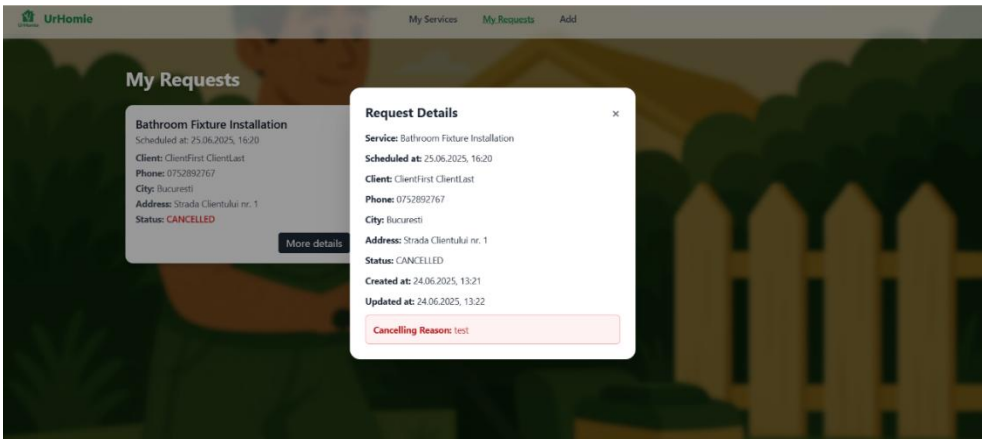


Figura 4: My Requests Page, acesarea opțiunii „More details" pe un request CANCELLED

- *Add Service Form*, include un formular pentru adăugarea unui nou serviciu, structurat dinamic în funcție de categoria selectată. Formularul cuprinde câmpuri personalizate, validări și opțiuni clare de completare, facilitând publicarea rapidă a serviciului în platformă.

- *Service Details Page,* oferă o interfață adaptată rolului utilizatorului. În cazul furnizorului, aceasta permite vizualizarea completă a informațiilor și realizarea acțiunilor de modificare a câmpurilor editabile, precum locația, prețul sau durata estimată, precum și ștergerea definitivă a serviciului din platformă. Pentru clienți, aceeași pagină este accesibilă din rezultatele căutării și include, suplimentar, informații despre furnizorul serviciului și posibilitatea de a efectua o rezervare. Procesul de rezervare este inițiat printr-un *modal* dedicat, unde clientul completează detaliile necesare programării, într-o interfață simplă și intuitivă;

- *Not Found Page,* afișată atunci când utilizatorul accesează o rută inexistentă sau o resursă invalidă. Aceasta oferă un mesaj clar de *eroare 404*, alături de opțiuni vizibile pentru a reveni la pagina anterioară sau pentru a naviga direct către *Home Page*.

3.4.1.2. Tehnologii utilizate

Interfața aplicației *UrHomie* este dezvoltată utilizând *React*, o bibliotecă modernă pentru construirea interfețelor interactive, care permite un control granular asupra stării aplicației și o organizare componentizată a codului. Pentru stilizare, se utilizează *Tailwind CSS*, un *framework utility-first* care oferă rapiditate în dezvoltare și o estetică coerentă, responsivă și ușor de personalizat.

Pentru a permite integrarea eficientă între *frontend* și serviciile *backend*, aplicația utilizează un set de tehnologii moderne de comunicare:

- *gRPC-Web*, pentru apelurile către microserviciul de autentificare, adaptat astfel încât să funcționeze în medii browser;

- *WebSocket*, pentru recepționarea notificărilor în timp real, în special în contextul confirmării înregistrării;

- *REST API,* utilizat pentru comunicarea cu microserviciile care gestionează profilurile utilizatorilor, serviciile oferite de furnizori și rezervările efectuate de clienți, permițând *operațiuni* de accesare, actualizare și administrare a acestor date.

3.4.1.3. Integrarea gRPC-Web în React

Pentru a permite comunicarea eficientă între aplicația *frontend* și microserviciul de autentificare scris în *gRPC*, platforma *UrHomie* utilizează *gRPC-Web*, o extensie a *framework-ului gRPC*, concepută special pentru a funcționa în browserele moderne. Acest protocol permite apeluri bidirecționale de tip *RPC* între un client *JavaScript* și un server *gRPC*, fără a fi nevoie de *fallback-uri* pe *REST* sau *polling*. Aplicația *React* este configurată ca un client *gRPC-Web* prin instalarea pachetelor necesare, precum *grpc-web* și *google-protobuf*, și generarea fișierelor client direct din fișierele *.proto* [13] (vezi Anexa 1). Acest proces se realizează cu ajutorul compilatorului *protoc*, utilizând comanda:

```
protoc -I=. user_auth.proto \
  --js_out=import_style=commonjs:. \
  --grpc-web_out=import_style=typescript,mode=grpcwebtext:.
```

Figura 5: Cod bash pentru compilarea fișierului .proto în React

Comanda generează fișiere *TypeScript* și *JavaScript* compatibile cu *gRPC-Web*, care sunt apoi importate în aplicație pentru a apela direct metodele declarate în fișierul *.proto*, precum *SignUp*, *LogIn*, *RefreshToken* etc. Pentru ca aceste apeluri să funcționeze în browser, comunicarea este direcționată printr-un reverse *proxy NGINX*, care trimite cererile către un *proxy Envoy* configurat special pentru conversia dintre *application/grpc-web* (folosit de browser) și *HTTP/2* (necesar serverului *gRPC*). Astfel, se asigură compatibilitate completă cu mediul de execuție web, fără a compromite performanța sau securitatea comunicației.

3.4.1.4. Gestionarea sesiunii și tratarea automată a erorilor de autentificare

Pentru a înțelege mai clar mecanismul de autentificare și gestionare a sesiunii în cadrul aplicației *UrHomie*, în Anexa 3 este prezentată o diagramă de activitate care detaliază pașii principali ai procesului de autentificare. Aceasta ilustrează interacțiunea dintre *frontend* și microserviciul *user-auth-ms*, atât în cazul autentificării directe (*login*), cât și în cazul reîmprospătării automate a sesiunii prin *refreshToken*. Diagrama reflectă modul în care sunt verificate credențialele, cum sunt generate și gestionate *token-urile JWT* și cum este actualizat contextul utilizatorului în aplicația *React*.

Pentru gestionarea autentificării și menținerea sesiunii active, aplicația utilizează un context global numit *AuthContext*. Acesta centralizează informațiile esențiale ale utilizatorului, precum *accessToken*, *refreshToken* (stocat în *cookie*), *userId* și rolul asociat. La inițializarea aplicației, se declanșează un apel *gRPC* către metoda *RefreshToken*, care validează sesiunea și generează un nou *token* de acces, dacă este cazul. În caz contrar, utilizatorul este considerat delogat și redirecționat corespunzător. *Token-ul* de acces este păstrat în memoria aplicației (*React state*) și utilizat în toate cererile către *endpoint-urile* securizate sub forma unui *Bearer Token*, adăugat automat în antetul fiecărui *request*.

Pentru trimiterea cererilor către *backend*, aplicația utilizează un client *Axios* configurat centralizat. Acesta conține un interceptor care verifică automat dacă serverul răspunde cu *eroarea 401 Unauthorized*, presupunând astfel că *token-ul* de acces a expirat. În acest caz, se încearcă reîmprospătarea sesiunii printr-un nou apel *gRPC* folosind *RefreshToken*. Dacă acest proces reușește, se repetă cererea inițială fără intervenție suplimentară din partea utilizatorului. În caz contrar, sesiunea este încheiată, iar utilizatorul este redirecționat către pagina de *login*.

Pentru o tratare unitară și *user-friendly* a codurilor de eroare venite de la server (precum *403, 404, 422*), aplicația folosește o funcție dedicată. Aceasta permite afișarea unor mesaje sugestive, personalizate pentru fiecare tip de eroare, și oferă posibilitatea de a declanșa acțiuni

automate, precum *logout-ul* sau navigarea către pagini dedicate (ex: „*not-found”*). Astfel, aplicația oferă o experiență robustă și coerentă în fața erorilor de rețea sau autentificare, menținând siguranța și continuitatea fluxului de utilizare.

3.4.1.5. Securitate și protecție

Aplicația implementează măsuri clare de securitate pentru protejarea sesiunii utilizatorilor. *Token-ul* de acces este stocat exclusiv în memorie, în timp ce *refresh token-ul* este salvat în *cookie*, o abordare menită să reducă riscul atacurilor de tip *XSS*. Accesul la rutele sensibile este controlat prin mecanismul *ProtectedRoute*, care verifică în mod constant prezența unui *token* valid și a unui rol autorizat. În plus, interfața previne executarea acțiunilor nepermise și redirecționează automat utilizatorul către pagina de autentificare în cazul în care sesiunea expiră sau *token-ul* devine invalid.

3.4.2. Componente proxy în arhitectura sistemului

Pentru a asigura o comunicare sigură, scalabilă și compatibilă între *frontend-ul React* și microserviciile *backend*, aplicația *UrHomie* utilizează două componente esențiale în infrastructura de rețea: *NGINX* și *Envoy*. Acestea joacă un rol critic în gestionarea rutării cererilor, asigurarea compatibilității între protocoale și configurarea corespunzătoare a politicilor *CORS*. *NGINX* acționează ca *reverse proxy* principal, expunând *endpoint-urile backend-ului* către *frontend* și aplicând setările de securitate necesare. *Envoy* completează arhitectura prin conversia protocoalelor *HTTP/1.1* și *HTTP/2*, permițând integrarea fluentă a *gRPC-Web* în contextul browserului.

3.4.2.1. NGINX – Reverse proxy și gestionarea CORS

Componenta *NGINX* este utilizată ca *reverse proxy*, având rolul de intermediar între client și serviciile *backend*, redirecționând cererile primite către microserviciile corespunzătoare și aplicând politicile de securitate, inclusiv configurarea *CORS*. În cadrul fișierului de configurare *nginx.conf*, sunt definite mai multe locații (ex: */UserAuthentication/, /api/user-management/*, etc.), fiecare mapată către o destinație *backend* internă. De asemenea, *NGINX* adaugă automat anteturi specifice pentru a permite partajarea resurselor între origini diferite (*CORS*), esențială pentru aplicațiile *frontend* care rulează în alt domeniu sau port față de serviciile *backend*.

NGINX gestionează în mod explicit cererile *preflight* (*OPTIONS*) printr-o variabilă dedicată (*$cors_preflight*) și răspunde cu anteturi detaliate, acceptând metodele *HTTP* relevante și anteturile necesare pentru interacțiunile moderne (inclusiv cele specifice *gRPC-Web*). Configurația include și activarea credențialelor (*Access-Control-Allow-Credentials*) pentru a permite utilizarea *cookie-urilor* (ex*: refresh token*) (vezi Anexa 4 pentru configurația din *NGINX*).

3.4.2.2. Envoy – Convertor de protocol pentru gRPC-Web

Envoy este un *proxy* avansat utilizat pentru a realiza conversia între cererile *HTTP/1.1* (transmise de browser prin *gRPC-Web*) și *HTTP/2* (protocolul nativ al *gRPC*). Această componentă este crucială pentru a permite *frontend-ului* să interacționeze eficient cu microserviciile *gRPC* implementate în *backend*, fără a compromite performanța sau compatibilitatea cu standardele browserelor.

Configurația *envoy.yaml* definește un *listener* care acceptă conexiuni *HTTP/1.1* din partea *NGINX*, apoi transformă aceste cereri în *HTTP/2* pentru a comunica cu serviciile *gRPC*. În acest mod, *Envoy* acționează ca un adaptor între tehnologiile moderne de browser și ecosistemul *gRPC*, eliminând necesitatea unor soluții intermediare complexe sau *workaround-uri*.

3.4.3. Componenta UserAuth Microservice

Microserviciul *UserAuth* are un rol central în gestionarea identității și a sesiunilor utilizatorilor. Acesta rulează un server *gRPC* (*grpc.aio*) care implementează o interfață definită pe baza unui fișier *.proto* compilat anterior, ce conține definițiile metodelor *LogIn*, *SignUp*, *ValidateJwt*, *RefreshToken* și *LogOut*. În paralel, microserviciul utilizează *FastAPI* pentru expunerea unui canal *WebSocket* prin care *frontend-ul* este notificat în timp real asupra finalizării procesului de înregistrare. Logica de orchestrare a înregistrării este implementată asincron, folosind un *pattern* de tip *Saga*, în combinație cu *RabbitMQ* (pentru schimbul de mesaje) și *Redis* (pentru persistarea temporară a stării operațiunilor).

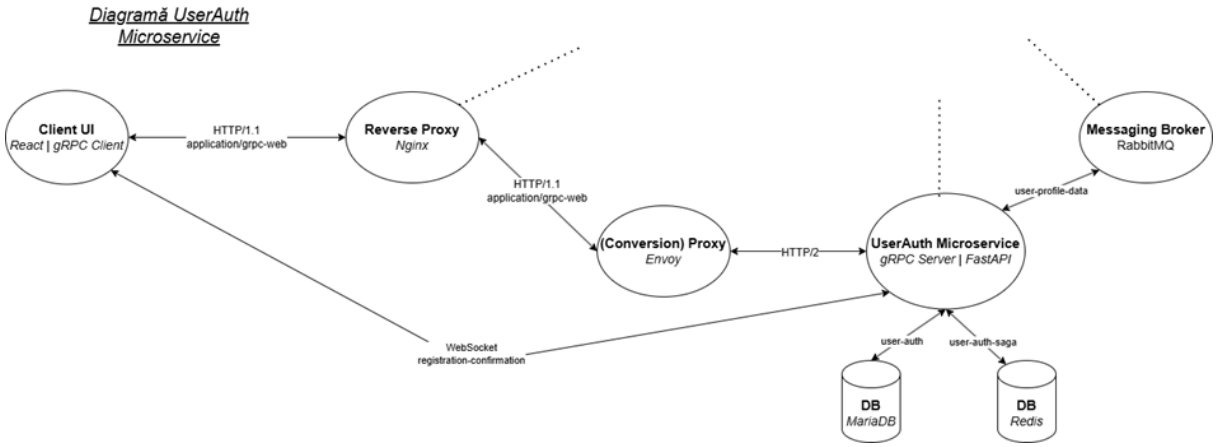


Figura 6: Diagrama UserAuth Microservice

3.4.3.1. Functionalități principale

UserAuth implementează cinci metode *gRPC* principale:

- ***LogIn***: validează credențialele și returnează un access *token* (*JWT*), în timp ce *refreshToken-ul* este transmis sub formă de *cookie* securizat (*HTTP-only*), protejat împotriva accesului din *JavaScript* (vezi Anexa 5);
- ***SignUp***: inițiază înregistrarea unui nou utilizator și publică un mesaj pe un exchange *RabbitMQ*, solicitând crearea unui profil în microserviciul *UserManagement*;
- ***ValidateJwt***: permite validarea *token-ului JWT* pentru a obține detalii despre utilizator și rolul acestuia;
- ***RefreshToken***: extrage *refresh token-ul* din *cookie* și generează un nou access *token* dacă acesta este valid;
- ***LogOut***: invalidează sesiunea prin ștergerea explicită a *cookie-ului*.

3.4.3.2. Orchestrarea înregistrării și gestiunea statusului (Saga)

Procesul de orchestrare a înregistrării este gestionat printr-un mecanism de tip *Saga*, care coordonează pașii necesari creării complete a unui utilizator și a profilului său. După crearea inițială a contului, microserviciul inițiază o instanță *Saga* în *Redis*, setând statusul operațiunii ca „*pending*”. Ulterior, publică un mesaj *CreateUserProfileMessage* pe un exchange de tip *fanout* în *RabbitMQ*, conform specificației oferite de *MassTransit*, pentru a notifica microserviciul *UserManagement* despre necesitatea creării profilului. Ulterior, microserviciul rămâne în așteptarea evenimentelor de tip răspuns, care pot fi fie *UserProfileCreated*, fie *UserProfileCreationFailed*. În funcție de

evenimentul recepționat, statusul corespunzător al instanței de *Saga* este actualizat în *Redis*, fiind marcat ca *completed* în caz de succes sau *failed* în caz de eșec. Dacă este detectat un eșec, sistemul declanșează automat mecanismul de compensare, constând în ștergerea contului de utilizator creat anterior.

Pentru a informa *frontend-ul* cu privire la starea înregistrării, *UserAuth* expune un *endpoint WebSocket (/user-auth/registration-status)*. *Frontend-ul* deschide o conexiune la acest canal și ascultă modificările asociate propriului *correlationId*. Odată ce statusul din *Redis* devine *completed* sau *failed*, se trimite un *payload JSON* corespunzător, iar intrarea este ștearsă pentru a evita aglomerarea bazei temporare. (Vezi Anexa 2, diagramă de activitate *Saga*)

3.4.3.3. Persistența datelor

Microserviciul *UserAuth* persistă datele esențiale ale utilizatorilor în tabela *user_account*, care reprezintă punctul central al gestiunii conturilor. Această tabelă conține următoarele atribute principale:

- *id*: identificator unic de tip *BIGINT*, utilizat drept cheie primară;
- *email*: adresa de e-mail a utilizatorului, garantată ca unică;
- *password*: parolă stocată în formă de *hash*, folosind algoritmul *bcrypt*;
- *role*: valoare de tip *ENUM* (*CLIENT, SERVICE_PROVIDER*), utilizată pentru gestionarea drepturilor de acces.

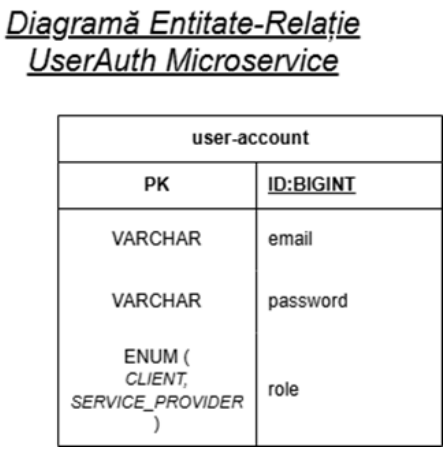


Figura 7: Diagrama Entitate-Relație UserAuth Microservice

3.4.3.4. Mecanisme de Securitate

Securitatea reprezintă un pilon fundamental în arhitectura microserviciului *UserAuth*, fiind asigurată printr-o serie de măsuri aplicate în toate etapele procesului de autentificare și gestionare a sesiunii. La autentificare, sistemul generează două tipuri de *token-uri*: *access token* și *refresh token*. *Access token-ul* este returnat în răspunsul *gRPC* și este utilizat de client în cererile către *endpoint-urile* securizate, fiind păstrat temporar în memoria aplicației *frontend*. În schimb, *refresh token-ul* este transmis sub formă de *cookie HTTP-only*, cu atribute restrictive precum *SameSite=Lax* și *Path=/*, pentru a preveni atacuri de tip *Cross-Site Request Forgery* (*CSRF*) și accesul din *JavaScript*.

Validarea *token-urilor* este gestionată riguros: *access token-ul* este verificat la fiecare cerere, iar *refresh token-ul* este evaluat separat printr-o metodă dedicată (*RefreshToken*), doar în scopul reînnoirii sesiunii. În cazul în care un *token* este expirat, invalid sau manipulat, serviciul răspunde automat cu statusul *UNAUTHENTICATED*, declanșând în *frontend* procesul de *logout* și

redirecționare. În plus, datele sensibile, precum parolele, sunt stocate exclusiv sub formă de *hash*, folosind algoritmul *bcrypt*, care asigură protecție împotriva atacurilor de tip dicționar sau *brute-force*. În eventualitatea unui eșec în procesul de înregistrare (de exemplu, eșecul de creare a profilului), microserviciul declanșează automat o acțiune compensatorie de ștergere a contului, prevenind astfel păstrarea unor înregistrări incomplete sau potențial exploatabile.

3.4.4. Componenta UserManagement Microservice

Microserviciul ***UserManagement*** este implementat ca un *REST API* în *ASP.NET Core Web API* și are rolul de a gestiona datele de profil extinse ale utilizatorilor, fie clienți sau furnizori de servicii. Pe lângă această funcționalitate principală, microserviciul acționează ca un consumator de mesaje *RabbitMQ*, intervenind în procesul asincron de creare a profilului după înregistrare, și joacă, totodată, rolul de client *gRPC*, apelând microserviciul *UserAuth* pentru validarea *token-urilor JWT* necesare autorizării accesului la *endpoint-urile* protejate.

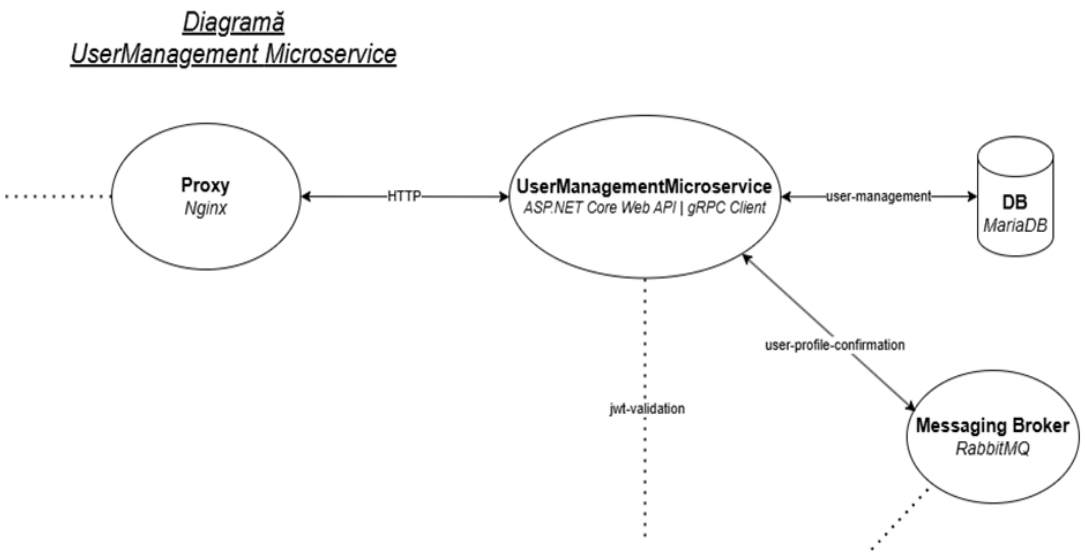


Figura 8: Diagrama UserManagement Microservice

3.4.4.1. Funcționalități principale

Microserviciul *UserManagement* expune un *REST API* prin care permite accesarea și gestionarea datelor extinse de profil pentru clienți și furnizori de servicii. Operațiile disponibile includ interogarea informațiilor curente și actualizarea acestora, iar accesul este strict autorizat prin validarea *token-ului JWT*. Totodată, acest microserviciu acționează ca un consumator de mesaje *RabbitMQ* pentru a procesa, în mod asincron, solicitările de creare profil, primite în urma unei înregistrări noi.

Actualizarea profilului de utilizator este realizată prin metoda *HTTP PATCH*, utilizând *DTO-uri* dedicate precum *UpdateClientDto* și *UpdateServiceProviderDto*. Sistemul aplică o strategie flexibilă de validare, concentrându-se exclusiv pe câmpurile efectiv furnizate în *request*, prin folosirea atributului *ValidateIfPresent*. Astfel, utilizatorii pot modifica doar datele dorite, fără a fi obligați să trimită întregul profil. În cazul câmpurilor sensibile precum email sau număr de telefon, sistemul impune unicitatea valorilor. De asemenea, se asigură că *request-ul* include cel puțin un câmp valid de actualizat, printr-o extensie custom numită *HasAtLeastOnePopulatedField*.

3.4.4.2. Consumator de mesaje RabbitMQ și Integrare cu MassTransit

Microserviciul *UserManagement* este configurat pentru a acționa ca un consumator activ de mesaje *RabbitMQ*, folosind biblioteca *MassTransit* pentru a simplifica modelul de mesagerie și a asigura o orchestrare robustă a evenimentelor. Scopul principal al acestui consumator este să prelucreze mesajele *CreateUserProfileMessage*, care semnalează necesitatea creării unui profil nou în urma unei înregistrări efectuate în microserviciul *UserAuth*.

La nivel de infrastructură, *MassTransit* configurează automat un *ReceiveEndpoint* numit *create_user_profile_queue*, care ascultă mesajele publicate în exchange-ul *create_user_profile_event*. Acest *endpoint* este mapat la o clasă dedicată, *CreateUserProfileConsumer*, responsabilă cu logica de procesare a mesajului primit. Pe baza datelor incluse, consumatorul determină tipul de utilizator (*Client* sau *Service Provider*) și creează entitatea corespunzătoare în baza de date. După încheierea procesului de creare profil, sistemul publică un răspuns într-un exchange de tip *fanout*, corespunzător rezultatului: *user_profile_created_event* sau *user_profile_failed_event*. Aceste evenimente de tip *UserProfileCreated* sau *UserProfileCreationFailed* includ același *CorrelationId* primit inițial, permițând astfel microserviciului *UserAuth* să coreleze starea operațiunii cu sesiunea corespunzătoare a utilizatorului (vezi Anexa 6). Această abordare asigură o decuplare completă între producătorul și consumatorul de mesaj, o trasabilitate clară a operațiunilor și posibilitatea de compensare în caz de eșec, respectând astfel principiile unui *pattern* asincron de tip *Request-Reply*, cu suport complet pentru consistență eventuală.

3.4.4.3. Persistența datelor

Baza de date a microserviciului *UserManagement* este organizată în jurul entității centrale *user_profile*, care stochează informațiile comune tuturor utilizatorilor: nume, prenume, email, număr de telefon, țară, oraș și adresă. Această entitate este proiectată pentru a permite o extindere elegantă și scalabilă, fiind relaționată prin asocieri de tip unu-la-unu cu entități specializate, în funcție de rolul utilizatorului. Pentru utilizatorii de tip *ServiceProvider*, profilul este completat cu atribute specifice precum nivelul de educație, certificările obținute, descrierea experienței, programul de lucru, aria geografică acoperită și lista de categorii de servicii oferite (*category_ids*). Entitatea client, asociată utilizatorilor de tip *Client*, este în prezent minimalistă, dar este prevăzută pentru a susține extensii ulterioare.

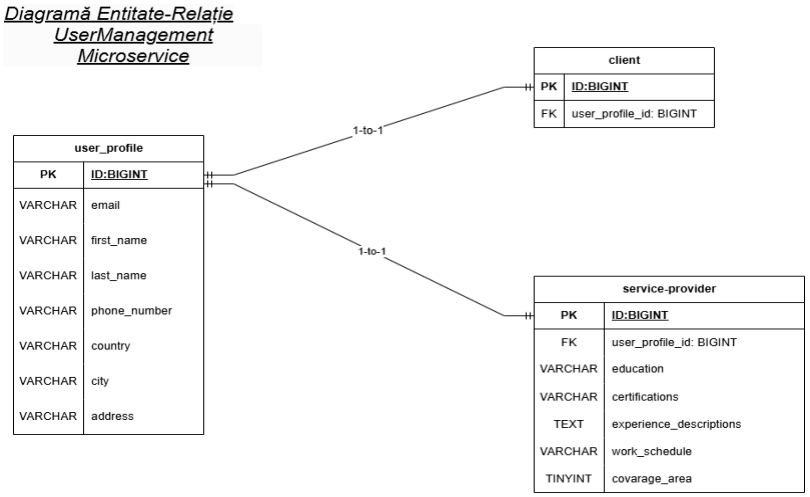


Figura 10: Diagrama Entitate-Relație UserManagement Microservice

3.4.4.4. Mecanisme de Securitate

În cadrul microserviciului *UserManagement*, securitatea este asigurată printr-un mecanism coerent de validare a autenticității și autorizării fiecărei cereri către *REST API-ul* expus. *Endpoint-urile* sunt protejate printr-un *middleware* personalizat (*ExternalJwtValidationMiddleware*), care interceptează fiecare cerere și transmite *token-ul JWT* extras din antet către microserviciul *UserAuth* printr-un apel *gRPC* la metoda *ValidateJwt*. Doar dacă *token-ul* este valid, sistemul injectează în context un obiect *ClaimsPrincipal*, ce conține identitatea și rolul utilizatorului. Pe baza acestor *claims* validate, se aplică politici de autorizare granulară, precum *SameClientOnly* sau *SameServiceProviderOnly*, care verifică dacă identificatorul utilizatorului din *token* corespunde cu cel din resursa accesată. Astfel, se previn accesările neautorizate ale profilurilor altor utilizatori.

Mai mult decât atât, pentru *endpoint-urile* de tip *PATCH*, care permit actualizarea parțială a profilului, se aplică un set de validări stricte, implementate prin atribute personalizate. Acestea se asigură că sunt validate doar câmpurile trimise efectiv în *request* și că este furnizat cel puțin un câmp valid de actualizat. În cazul unor erori de validare, aplicația returnează un răspuns *422 Unprocessable Entity* cu detalii clare despre câmpurile invalide. De asemenea, pentru a preveni duplicarea datelor sensibile (ex: email, telefon), sunt tratate explicit și conflictele de integritate (*409 Conflict*), detectate prin extensia *IsUniqueConstraintViolation*.

3.4.5. Componenta ServiceCatalog Microservice

Componenta *ServiceCatalog* joacă un rol esențial în cadrul platformei *UrHomie*, fiind responsabilă pentru gestionarea serviciilor oferite de furnizorii înregistrați. Această componentă este dezvoltată folosind *Spring Boot* și interacționează cu o bază de date *MongoDB*, aleasă pentru flexibilitatea în stocarea datelor semi-structurate și adaptabilitatea la structuri dinamice (ex: câmpuri personalizate pe categorie). Microserviciul expune un set de *endpoint-uri REST* și se integrează cu restul platformei prin validarea *token-urilor JWT* folosind un client *gRPC* către *UserAuth*, asigurând astfel o autorizare riguroasă pentru fiecare operațiune efectuată.

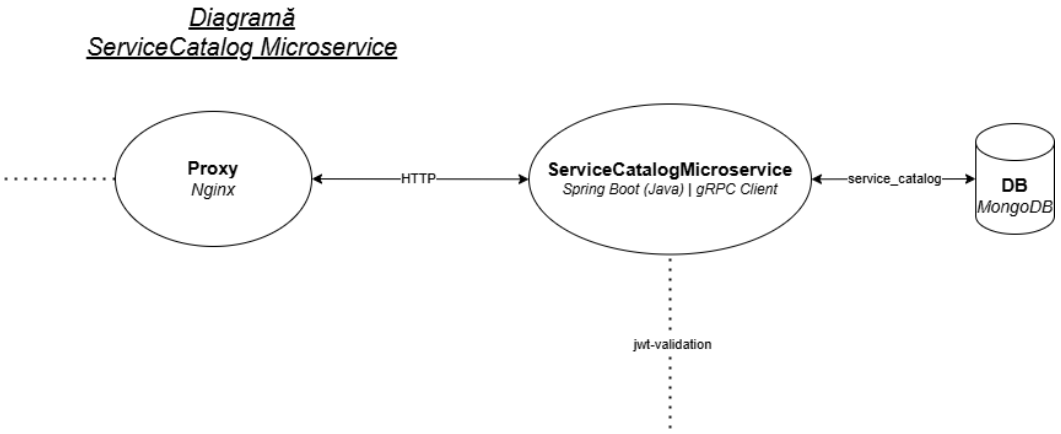


Figura 11: Diagramă ServiceCatalog Microservice

3.4.5.1. Funcționalități principale

Componentele funcționale ale microserviciului *ServiceCatalog* sunt organizate în jurul a trei funcționalități esențiale: gestionarea categoriilor de servicii, a șabloanelor de câmpuri asociate acestora și a serviciilor efectiv publicate de furnizori. Sistemul permite expunerea publică a unei liste de categorii predefinite, fiecare definită printr-un identificator unic, o denumire sugestivă, o descriere concisă și o imagine reprezentativă. Aceste categorii constituie baza organizării informațiilor prezentate în platformă și contribuie la o structurare clară a serviciilor oferite. Pe lângă alegerea unei categorii, aplicația oferă posibilitatea de configurare dinamică a formularului de publicare a serviciilor, prin definirea unor șabloane de câmpuri specifice fiecărei categorii. Aceste

șabloane permit personalizarea conținutului completat de furnizori în funcție de natura serviciului, oferind o flexibilitate ridicată în colectarea datelor.

Serviciile publicate de către furnizori pot fi adăugate, modificate sau eliminate prin intermediul unor operații expuse prin *endpoint-uri REST*. Platforma susține atât crearea completă a unui serviciu, cât și actualizarea sa parțială, fiind implementate mecanisme care aplică modificările doar asupra câmpurilor furnizate explicit. În plus, validările de consistență și integritate sunt asigurate printr-un sistem de validari generale, dar și custom, iar toate erorile apărute sunt gestionate uniform de către un *handler* global, care returnează răspunsuri semnificative pentru coduri precum *400, 409, 422* sau *404*.

3.4.5.2. Căutarea avansată a serviciilor

Un alt element definitoriu al microserviciului este mecanismul de căutare avansată, care oferă utilizatorilor posibilitatea de a identifica rapid servicii relevante, pe baza unor potriviri contextuale aplicate simultan pe câmpuri precum titlu, descriere, numele categoriei, oraș sau adresă. Această funcționalitate este integrată direct în interfața principală a aplicației, oferind o experiență de căutare flexibilă și tolerantă la variații de ordine sau formulare a cuvintelor-cheie. La nivel de *backend*, interogarea este construită dinamic și aplicată asupra documentelor stocate în *MongoDB*, folosind expresii regulate și operatori logici care permit evaluarea simultană a mai multor câmpuri. Pentru un plus de expresivitate și control, se utilizează *pipeline-uri* de agregare, care permit filtrarea rezultatelor și structurarea acestora într-un format adecvat pentru *frontend* (paginare, sortare, limitare). Această abordare, bazată pe facilitățile native ale *MongoDB*, oferă o combinație eficientă între flexibilitate și performanță, permițând platformei să scaleze odată cu creșterea volumului de date și a gradului de personalizare a serviciilor.

3.4.5.3. Persistența datelor

Modelul de stocare al microserviciului *ServiceCatalog* este organizat în jurul a trei colecții principale: categorii de servicii, șabloane de câmpuri asociate și servicii publicate de furnizori. Baza de date utilizată este *MongoDB*, aleasă pentru capacitatea sa de a gestiona documente semi-structurate și pentru flexibilitatea oferită în scenarii cu structură de date variabilă, cum este cazul formularelor dinamice. Fiecare serviciu salvat conține informații precum titlu, descriere, preț, durată estimativă, locație și un câmp de detalii configurabil, a cărui structură depinde de categoria din care face parte serviciul respectiv.

Relațiile dintre colecții sunt modelate simplu: fiecare serviciu aparține unei categorii, iar fiecare categorie este asociată unui șablon care definește câmpurile disponibile în secțiunea de detalii. Această organizare de tip unu-la-mai-mulți (*categorie–servicii*) și unu-la-unu (*categorie–șablon*) permite extinderea ușoară a platformei și susține o diversitate largă de servicii fără a compromite coerența datelor. Modelul rămâne scalabil și eficient, fiind susținut de capabilitățile native ale *MongoDB* în gestionarea interogărilor complexe și actualizărilor selective.

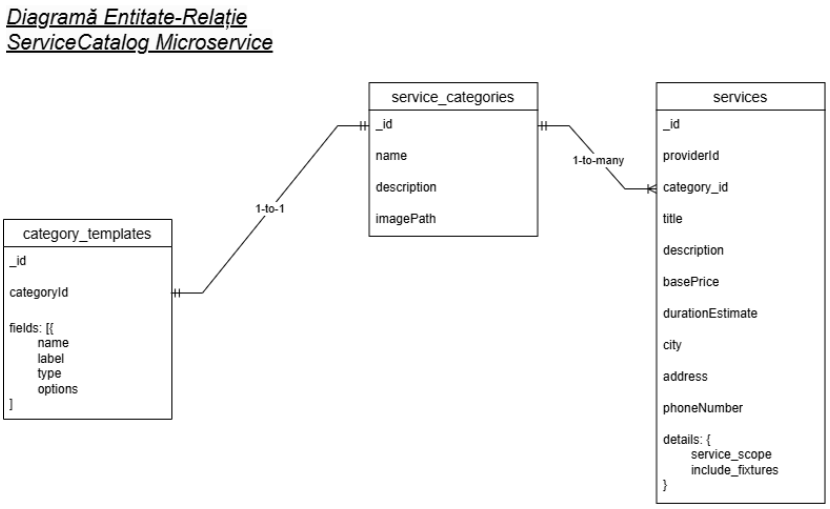


Figura 12: Diagrama Entitate-Relație ServiceCatalog Microservice

3.4.5.4. Mecanisme de securitate

În cadrul microserviciului *ServiceCatalog*, securitatea este implementată printr-un sistem riguros de validare a autenticității și autorizării cererilor transmise către *REST API*. Fiecare *request* este interceptat de un filtru de autentificare care extrage *token-ul JWT* din antetul cererii și îl transmite, printr-un apel *gRPC*, către microserviciul *UserAuth*, pentru validare prin metoda *ValidateJwt*. În urma acestei validări, în contextul aplicației este injectată identitatea utilizatorului și rolul asociat, ceea ce permite aplicarea unor politici de acces bine definite asupra resurselor expuse. Pe baza acestor informații, sunt aplicate restricții clare de autorizare, în special în cazul operațiunilor critice precum modificarea sau ștergerea serviciilor. Pentru a preveni accesul neautorizat, sunt efectuate verificări explicite care asigură că utilizatorul autentificat este, într-adevăr, furnizorul serviciului vizat. Astfel, doar autorul unui serviciu poate opera asupra acestuia, iar orice încercare de acces neautorizat este blocată printr-un răspuns corespunzător. În plus, sistemul de validare tratează separat și cazurile în care *token-ul* este invalid, expirat sau lipsă, printr-un mecanism global de gestionare a erorilor care returnează statusuri *HTTP* clare, precum *401 Unauthorized* sau *403 Forbidden*.

De asemenea, pentru operațiile de actualizare parțială, aplicația aplică reguli stricte de validare per câmp. Se verifică atât prezența unor câmpuri valide în *request*, cât și conformitatea acestora cu structura definită de categoria serviciului. În cazul în care apar erori de validare, răspunsul furnizat este de tip *422 Unprocessable Entity*, însoțit de detalii clare despre câmpurile afectate.

3.4.6. Componenta Booking Microservice

Microserviciul *Booking* are rolul de a centraliza și administra procesul de rezervare al serviciilor disponibile pe platforma *UrHomie*, facilitând interacțiunea structurată dintre utilizatori și furnizorii înregistrați. Implementat în *Java*, folosind *Spring Boot*, acesta operează pe o bază de date relațională *MariaDB*, potrivită pentru modelarea clară a entităților și pentru menținerea integrității tranzacțiilor. Expunerea funcționalităților se realizează printr-un *REST API*, iar accesul este securizat printr-un mecanism de autorizare distribuit, în care validarea *token-urilor JWT* este externalizată către microserviciul dedicat autentificării, prin intermediul unei interfețe *gRPC*.

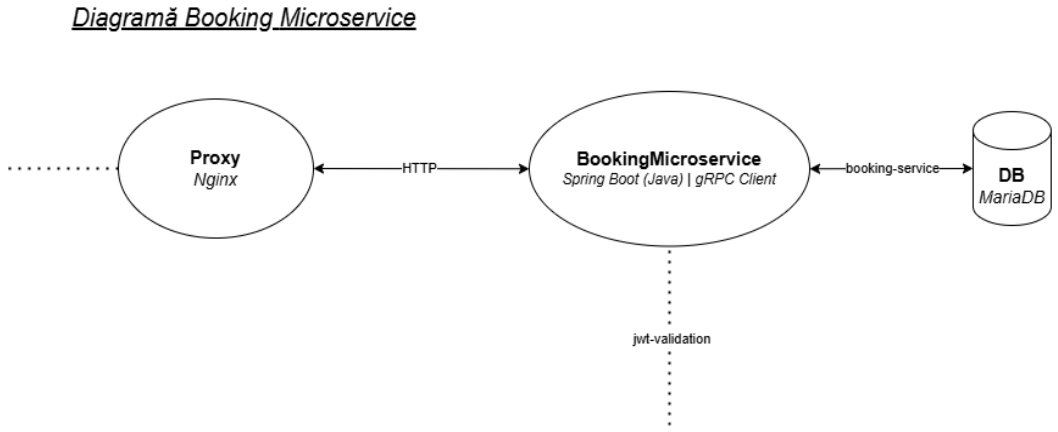


Figura 13: Diagramă Booking Microservice

3.4.6.1. Funcționalități principale

Funcționalitățile principale ale microserviciului *Booking* sunt centrate pe administrarea completă a rezervărilor efectuate în platformă, de la inițierea unei cereri de programare până la finalizarea sau anularea acesteia. Sistemul gestionează atât rezervările inițiate de clienți, cât și acțiunile furnizorilor asupra acestor cereri, printr-o suită de operații accesibile prin intermediul unui *REST API*. Crearea unei rezervări presupune completarea unui formular ce include date precum serviciul selectat, momentul dorit pentru programare și detalii suplimentare relevante. După înregistrare, rezervarea este marcată implicit cu statusul *PENDING*, urmând a fi procesată de furnizorul asociat. Odată primită, rezervarea poate fi fie acceptată (*CONFIRMED*), fie respinsă (*CANCELLED*) de către furnizor, iar în cazul în care serviciul este finalizat, acesta poate marca rezervarea ca *COMPLETED*. În paralel, sistemul permite și anularea unei rezervări de către client, cu transmiterea unui mesaj justificativ. Toate modificările de status sunt înregistrate în mod transparent, fiind acompaniate de detalii privind autorul acțiunii și momentul efectuării. Această trasabilitate este asigurată printr-o componentă dedicată de tip jurnal, *BookingLog*, care permite atât consultarea statusului curent, cât și obținerea unui istoric complet al unei rezervări.

3.4.6.2. Persistența datelor

Modelul de stocare al microserviciului *Booking* este organizat în jurul a trei entități relaționale: rezervarea propriu-zisă, detaliile asociate acesteia și istoricul modificărilor. Datele sunt persistate într-o bază de date relațională *MariaDB*, aleasă pentru consistența ridicată a tranzacțiilor și capacitatea de a susține integritatea referențială între entități. Entitatea principală, *booking*, stochează informațiile esențiale legate de o programare: clientul și furnizorul implicați, serviciul rezervat, statusul actual al rezervării și legătura cu detaliile suplimentare.

Detaliile aferente fiecărei rezervări , precum numele clientului, locația, ora programării și eventuale observații, sunt păstrate într-o entitate separată, *booking_details*, cu o relație de tip unu-la-unu față de înregistrarea principală. Această separare permite o gestionare mai clară a datelor contextuale și susține o eventuală extindere a structurii fără a afecta schema rezervării. În plus, orice schimbare de stare a unei rezervări este înregistrată în entitatea *booking_log*, care păstrează informații despre statusul anterior și cel nou, identificatorul utilizatorului care a efectuat modificarea și un eventual mesaj de respingere. Această relație de tip unu-la-mai-mulți între rezervare și loguri asigură trasabilitatea completă a fiecărei acțiuni, fiind esențială pentru audit și transparență în interacțiunile client–furnizor.

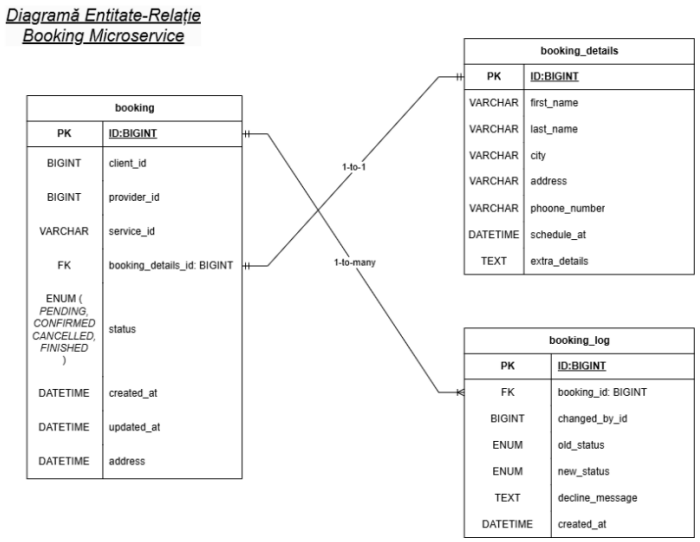


Figura 14: Diagrama Entitate-Relație Booking Microservice

3.4.6.3. Mecanisme de securitate

În cadrul microserviciului *Booking*, securitatea este implementată printr-un mecanism clar de validare a autenticității și autorizării cererilor adresate către *REST API*. Fiecare cerere este interceptată de un filtru de autentificare care extrage *token-ul JWT* din antet și îl transmite către microserviciul *UserAuth* printr-un apel *gRPC*, apelând metoda dedicată de validare. În urma acestui proces, identitatea utilizatorului și rolul său sunt injectate în contextul aplicației, permițând aplicarea unor politici de acces stricte în funcție de acțiunea solicitată. Pe baza acestor informații validate, se aplică restricții clare de autorizare, în special în cazul operațiunilor sensibile precum vizualizarea detaliilor unei rezervări, modificarea statusului sau inițierea unei acțiuni în numele altui utilizator.

Pentru a preveni accesările neautorizate, sunt implementate verificări explicite care asigură că utilizatorul curent este, de exemplu, clientul care a inițiat rezervarea sau furnizorul căruia îi aparține serviciul rezervat. În cazul în care aceste condiții nu sunt îndeplinite, aplicația returnează un răspuns adecvat (*401* sau *403*), printr-un sistem global de tratare a excepțiilor care gestionează toate erorile de acces și autentificare într-un mod unitar și predictibil. De asemenea, pentru acțiunile care implică actualizarea statusului unei rezervări sau completarea unui mesaj de anulare, sistemul validează riguros prezența și consistența datelor furnizate în *request*. În cazurile în care cererea este incompletă sau conține câmpuri invalide, serverul returnează un răspuns de tip *422 – Unprocessable Entity*, însoțit de informații detaliate despre câmpurile afectate, microserviciul asigurându-se astfel de protecția resurselor și de o interacțiune robustă și clară din perspectiva utilizatorului.

3.5. Alegerea tehnologiilor

Pentru dezvoltarea platformei distribuite *UrHomie*, alegerea tehnologiilor utilizate a fost realizată pe baza unor criterii precum modernitatea și popularitatea în industrie, suportul comunității, compatibilitatea cu scenariile proiectului și, nu în ultimul rând, dorința de a înțelege și experimenta integrarea practică între ecosisteme diferite. Astfel, aplicația reunește componente dezvoltate în trei limbaje și *framework-uri* majore, *Java* (*Spring Boot*), *C#* (*ASP.NET Core Web API*), și *Python* (*FastAPI*), comunicând între ele prin protocoale moderne precum *REST* și *gRPC*, toate integrate și orchestrate prin *proxy-uri* avansate precum *NGINX* și *Envoy*.

Pentru *frontend*, am ales *React*, o bibliotecă *JavaScript* open-source menținută de *Meta*, cunoscută pentru performanța ridicată în aplicații de tip *single-page* (*SPA*) și pentru ecosistemul matur din jurul său. Această alegere a permis construirea unei interfețe dinamice, modularizate pe componente, ușor de integrat cu mecanisme moderne de reactivitate și gestionare a stării. *React* este în prezent una dintre cele mai utilizate tehnologii *frontend* în industrie, fiind adoptată de companii

precum *Netflix*, *Airbnb* și *Uber* datorită flexibilității și eficienței sale [14].

Pentru comunicarea între *frontend* și *backend*, în special cu microserviciul de autentificare scris în *Python*, am utilizat *gRPC* împreună cu *gRPC-Web*. Această tehnologie, dezvoltată de *Google*, oferă performanțe superioare *REST-ului* tradițional datorită utilizării protocolului *HTTP/2* și a serializării binare (*Protocol Buffers*), facilitând apeluri rapide și eficiente. Integrarea cu *frontend-ul* a fost posibilă prin introducerea *proxy-ului Envoy*, care convertește cererile *HTTP/1.1* compatibile cu browserul în apeluri *HTTP/2* necesare serverului *gRPC*. Acest setup este validat de practica curentă în companii mari precum *Google*, *Lyft* sau *Square*, care adoptă *Envoy* ca *proxy* universal în arhitecturi distribuite moderne [15].

Pentru componenta de autentificare și autorizare, am optat pentru implementarea unui microserviciu în *Python*, folosind *framework-ul FastAPI*, cunoscut pentru simplitatea sintaxei și performanțele excelente în aplicații asincrone. *FastAPI* se bazează pe standardele moderne *OpenAPI* și este recomandat în multe scenarii de microservicii pentru viteza de dezvoltare și suportul nativ pentru validarea datelor. Astfel, această alegere a permis implementarea rapidă a unui server *gRPC* eficient, responsabil de generarea și validarea *token-urilor JWT*, cu integrare ușoară cu *Redis* și *RabbitMQ*.

Microserviciile orientate pe modelul utilizatorilor și al serviciilor oferite sunt implementate folosind *ASP.NET Core Web API* și *Spring Boot*, două *framework-uri* mature și robuste, fiecare în ecosistemul său. *ASP.NET Core* a fost ales pentru componenta *UserManagement* datorită suportului excelent pentru *REST API-uri*, a instrumentelor moderne de validare și a performanței în mediile enterprise *.NET*. În paralel, *Spring Boot* a fost utilizat pentru microserviciile *Booking* și *ServiceCatalog*, fiind considerat standardul de facto în lumea *Java* pentru dezvoltarea de aplicații enterprise rapide și scalabile. Ambele tehnologii permit o configurare ușoară a validărilor, autentificării și gestionării erorilor, integrându-se fără dificultăți în arhitectura generală [16][17].

Pentru orchestrarea și expunerea *endpoint-urilor* publice ale microserviciilor am folosit *NGINX* ca *reverse proxy* principal, iar *Envoy* pentru conversia apelurilor *gRPC-Web*. Această combinație este frecvent întâlnită în sisteme distribuite moderne și este recomandată în arhitecturi bazate pe servicii, pentru separarea clară între client și *backend* și pentru facilitarea securității, rutării și compatibilității *cross-platform*.

3.6. Identificarea avantajelor și a dezavantajelor

Alegerea unei arhitecturi distribuite bazate pe microservicii aduce o serie de avantaje semnificative în contextul platformei *UrHomie*. În primul rând, separarea clară a responsabilităților permite dezvoltarea, testarea și scalarea independentă a fiecărei componente, un aspect esențial în proiecte complexe și în continuă evoluție. Folosirea unor protocoale moderne precum *gRPC*, alături de *REST*, optimizează performanța comunicației inter-servicii și oferă flexibilitate în funcție de nevoile fiecărui flux. Tehnologiile moderne adoptate (*React, Spring Boot, FastAPI, ASP.NET Core*) sunt susținute de comunități vaste și documentații solide, ceea ce a facilitat dezvoltarea și învățarea pe parcurs. În plus, utilizarea *Envoy* și *NGINX* asigură o rutare eficientă și sigură a cererilor, iar validarea distribuită a autentificării prin *token-uri JWT* a contribuit la consolidarea unui mecanism de securitate scalabil și coerent.

Totuși, arhitectura aleasă implică și anumite dezavantaje, în special din perspectiva complexității de ansamblu. Coordonarea între microservicii dezvoltate în limbaje și *framework-uri* diferite presupune un efort suplimentar de integrare, versionare și monitorizare, iar procesul de depanare poate deveni mai dificil în lipsa unor mecanisme robuste de observabilitate. Utilizarea *gRPC-Web* necesită o configurație specifică și adaugă un strat suplimentar de infrastructură (ex: *Envoy*), care poate complica *deploy-ul* și *debugging-ul* în mediile locale. În plus, multiplicarea tehnologiilor implică o curbă de învățare mai abruptă și o nevoie crescută de sincronizare între echipe sau între componente, ceea ce poate afecta inițial viteza de dezvoltare și întreținere. Cu toate acestea, aceste dezavantaje sunt compensate de valoarea didactică ridicată și de apropierea față de scenariile reale din industria software actuală.

Capitolul 4. Testarea soluției și rezultate experimentale

4.1. Descrierea modalității de punere în funcțiune/lansare a aplicației

Pentru a facilita dezvoltarea, testarea și rularea distribuită a aplicației *UrHomie*, întregul ecosistem *backend* a fost containerizat folosind tehnologia *Docker* și orchestrat cu ajutorul unui fișier *docker-compose* (vezi Anexa 7). Această abordare permite pornirea rapidă a tuturor microserviciilor și a componentelor auxiliare (precum bazele de date, *proxy-urile* etc.), eliminând nevoia unor configurări manuale complexe și reducând riscul discrepanțelor între mediile de dezvoltare și producție.

Fișierul *docker-compose.yml* definește serviciile principale ale aplicației, împărțite în module funcționale distincte: *user-auth, user-management, service-catalog, booking*, precum și componentele de infrastructură necesare pentru funcționarea acestora, precum *nginx, envoy, redis, mariadb, mongodb și rabbitmq*. Pentru fiecare serviciu sunt configurate volume persistente, variabile de mediu, porturi expuse și comenzi de inițializare, în funcție de nevoile specifice.

4.1.1. Configurarea și containerizarea UserAuth Microservice

Microserviciul *user-auth-ms* este definit în fișierul *docker-compose.yaml* și rulează împreună cu serviciile auxiliare *user-auth-db, user-auth-redis* și *rabbitmq*, în cadrul rețelelor *Docker backend* și *user-auth-db*. Imaginea este construită local pe baza unui *Dockerfile* dedicat, care pornește de la o imagine oficială *Python* și instalează toate dependențele definite în fișierul *requirements.txt*. Acestea includ pachete precum *fastapi, uvicorn[standard], grpcio, peewee, redis, PyJWT, aio-pika* și *mysqlclient*, necesare pentru funcționarea serverului *gRPC* și *WebSocket*, gestionarea sesiunilor și conexiunile la baza de date. Containerul *user-auth-ms* are configurat un *healthcheck* care interoghează periodic *endpoint-ul (healthcheck)* pentru a verifica starea serviciului. Porturile *50051 (gRPC)* și *8000 (HTTP/WebSocket)* sunt expuse și mapate local pentru a permite accesul altor servicii și testarea externă. Variabilele de mediu sunt încărcate din fișierul *.env.production*, care include configurări precum hostul bazei de date, portul, utilizatorul și parola, *secret-key-ul* pentru *JWT* sau adresele serviciilor *Redis* și *RabbitMQ*.

Containerul *user-auth-db* rulează o instanță *MariaDB* și utilizează un volum persistent (*user-auth-db-data*) pentru stocarea datelor. La inițializare, sunt montate și executate trei scripturi *SQL*: *1-grant.sql*, *2-create.sql* și *3-insert.sql*, care creează baza de date *user-auth*, tabela *user_account*, și inserează câteva conturi de test. Aceste fișiere asigură o stare minimă validă a bazei de date pentru testare imediată. Containerul este inclus în aceeași rețea cu serviciul *user-auth-ms*, fiind accesibil prin *hostname-ul user-auth-db*. *Redis* este containerizat sub denumirea *user-auth-redis* și utilizat pentru stocarea temporară a statusurilor asincrone (ex: înregistrare în curs), în timp ce *rabbitmq* este partajat cu restul ecosistemului pentru publicarea și consumul de mesaje în contextul înregistrării distribuite. Ambele sunt configurate cu volume persistente și sunt conectate la rețeaua *backend*, fiind accesibile direct de către *user-auth-ms*. Această structură modulară permite replicarea consistentă a mediului de dezvoltare și testare, cu izolarea clară a componentelor și configurarea declarativă a tuturor serviciilor implicate.

4.1.2. Configurarea și containerizarea UserManagement Microservice

Microserviciul *user-management* este containerizat în *Docker* și rulează în cadrul rețelei *backend*, împreună cu o instanță *MariaDB* sub denumirea *user-management-mariadb* și cu serviciul de mesagerie *rabbitmq*. Configurările aplicației sunt centralizate în fișierul *appsettings.Production.json*, care definește parametrii de conectare la baza de date (host, port, utilizator, parolă) și configurația pentru *RabbitMQ*. Datele sunt persistate într-un volum dedicat (*user-management-db-data*), iar la inițializare sunt rulate scripturi *SQL* (1-*grant.sql*, 2-*create.sql*, 3-*insert.sql*) ce configurează schema bazei de date, creează tabelele pentru *user_profile*, *client*,

service_provider și inserează câteva înregistrări de test, utile pentru validarea funcțională a serviciului.

Containerul este construit pe baza unui *Dockerfile* care pornește de la o imagine *.NET SDK* și publică aplicația folosind *dotnet publish*. La rulare, aplicația expune un *endpoint REST* accesibil pe portul *8081*, care este mapat în *docker-compose* pentru a permite testarea locală sau integrarea cu alte componente. De asemenea, microserviciul consumă evenimente distribuite prin *RabbitMQ*. Pentru accesul *gRPC* la microserviciul user-auth, proiectul este configurat ca *gRPC client*: fișierul *.csproj* include directive pentru compilarea automată a fișierului *.proto* la *build*, specificând sursa și *namespace-ul*, astfel încât serviciile *gRPC* să fie disponibile ca interfețe direct în codul *C#*. Acest mecanism permite apelul metodei *ValidateJwt* pentru autorizarea utilizatorilor. În ceea ce privește dependințele, proiectul utilizează pachete precum *Grpc.Net.Client* pentru integrarea *gRPC*, *MassTransit* pentru mesagerie asincronă cu suport pentru *RabbitMQ*, și *Pomelo.EntityFrameworkCore.MySql* pentru conectivitate și *ORM* cu baza de date *MariaDB*.

4.1.3. Configurarea și containerizarea ServiceCatalog Microservice

Microserviciul *service-catalog* este implementat în *Java* folosind *framework-ul Spring Boot* și este containerizat cu ajutorul unui *Dockerfile* dedicat. Configurațiile necesare pentru rularea în mediu de producție sunt definite în fișierul *application-prod.properties*, care include detalii despre conexiunea la baza de date *MongoDB*, locația serverului *gRPC* pentru autentificare, portul aplicației și alte setări esențiale. Aplicația este configurată să ruleze pe portul *8083*, care este mapat corespunzător în fișierul *docker-compose.yaml* pentru a permite accesul altor servicii și testarea locală. Microserviciul este conectat la containerul *service-catalog-mongodb*, care rulează *MongoDB* pe portul standard și este accesat cu autentificare prin *authSource=admin*. Datele sunt persistate într-un volum *Docker* montat ca *service-catalog-db-data*.

Pentru funcționarea ca *gRPC* client, proiectul este configurat să compileze automat fișierul *.proto* al serviciului *user-auth* la momentul *build-ului*. Acest lucru este posibil datorită configurării în *pom.xml* a *plugin-ului protobuf-maven-plugin*, care utilizează *protoc* și *protoc-gen-grpc-java* pentru a genera interfețele și clasele necesare comunicării. După compilare, fișierele rezultate sunt plasate în directorul target/generated-sources, care este setat ca *resource root* în proiect, permițând accesul direct la clasele generate în codul *Java*. Această integrare *gRPC* permite microserviciului să valideze *token-urile JWT* prin apeluri către metoda *ValidateJwt*, expusă de *user-auth-ms*.

În ceea ce privește dependințele, fișierul *pom.xml* include toate bibliotecile necesare pentru rularea microserviciului: *spring-boot-starter-data-mongodb* pentru accesul la baza de date *NoSQL, spring-boot-starter-security* pentru gestionarea autentificării și autorizării, și *spring-boot-starter-web* pentru expunerea *endpoint-urilor REST*. Pentru partea de *gRPC* sunt incluse bibliotecile *grpc-netty-shaded, grpc-protobuf, grpc-stub, protobuf-java*, iar pentru procesarea adnotărilor este adăugat și *lombok*. Compilarea aplicației se face prin comanda *Maven* standard, iar imaginea *Docker* rezultată este construită local și orchestrată în cadrul rețelei *backend*.

4.1.4. Configurarea și containerizarea Booking Microservice

Microserviciul *booking-microservice* este containerizat folosind un *Dockerfile* dedicat, care pornește de la o imagine oficială *Java*, compilează aplicația folosind *Maven* și publică un jar executabil *Spring Boot*. Configurațiile specifice mediului de producție sunt definite în fișierul *application-prod.properties*, care conține datele de conectare la baza de date *MariaDB* (*booking-mariadb*), setările pentru serverul *gRPC* de autentificare (*user-auth-ms*) și portul de expunere al aplicației. Serviciul este configurat să ruleze pe portul *8082*, acesta fiind mapat corespunzător în fișierul *docker-compose.yaml*, pentru a permite comunicarea cu alte servicii și testarea locală. Persistența este asigurată printr-un volum *Docker* montat în containerul bazei de date, care reține toate datele despre rezervări și loguri. Pentru persistență relațională, aplicația folosește *Spring Data JPA* împreună cu driverul oficial *MariaDB* (*mariadb-java-client*). Conexiunea la baza de date este

configurată cu *ddl-auto=none*, deoarece schema este creată separat, în containerul de inițializare, prin rularea scripturilor *SQL*. Acestea definesc tabelele pentru rezervări, detalii asociate și istoricul modificărilor (*booking log*), asigurând o structură robustă pentru tranzacțiile legate de programări.

Proiectul este configurat și ca *gRPC* client, fiind capabil să comunice cu microserviciul de autentificare pentru validarea *token-urilor JWT*. Această integrare este realizată prin includerea în *pom.xml* a dependențelor necesare (*grpc-netty-shaded, grpc-stub, grpc-protobuf, protobuf-java*) și a *pluginului protobuf-maven-plugin*, care compilează fișierele *.proto* la *build*. Fișierele generate sunt plasate în *target/generated-sources* și sunt automat incluse în *classpath-ul* proiectului, facilitând utilizarea serviciului *gRPC* direct din codul *Java*. Pe lângă dependințele *gRPC* și *Spring Boot* de bază (*starter-data-jpa, starter-security, starter-web*), proiectul utilizează *lombok* pentru reducerea *boilerplate-ului* și *springdoc-openapi* pentru generarea documentației *API*. Astfel, microserviciul este complet containerizat, conectat la infrastructura existentă și configurat pentru a gestiona rezervările în mod sigur, scalabil și integrabil în restul arhitecturii distribuite.

4.1.5. Containerizarea componentelor proxy: NGINX și Envoy

Componentele *NGINX* și *Envoy* sunt containerizate și orchestrate în cadrul aceleiași rețele *Docker* denumite *backend*, alături de microserviciile aplicației. Acestea joacă rolul de puncte de intrare pentru cererile *frontend-ului* și pentru traducerea protocoalelor, fiind esențiale în comunicarea între clientul web și serviciile *gRPC* din *backend*. În cadrul configurației *docker-compose*, fiecare dintre aceste *proxy-uri* este definit ca un serviciu separat, cu un volum montat local care conține fișierele de configurare corespunzătoare. Containerul *NGINX* este utilizat ca *reverse proxy* principal pentru *frontend*. La *build*, fișierul de configurare personalizat (*nginx.conf*) este copiat în containerul *NGINX* și montat în directorul standard */etc/nginx/nginx.conf*. Acest fișier direcționează cererile *HTTP* provenite din browser către *frontend* și, în funcție de rută, le redirecționează către *Envoy* pentru apelurile *gRPC-Web*. *NGINX* este configurat să asculte pe portul *80*, acesta fiind mapat în *docker-compose.yaml* pentru a permite accesul din exterior. Containerul este inclus în rețeaua *backend*, ceea ce permite accesul direct la microserviciile disponibile pe această rețea fără configurări suplimentare.

Envoy, pe de altă parte, este responsabil pentru transcodarea (gresit) apelurilor *HTTP/1.1* provenite de la *frontend* în apeluri *gRPC* compatibile cu *HTTP/2*, necesare comunicării cu serverele *gRPC backend*. Fișierul de configurare *Envoy* (*envoy.yaml*) este de asemenea copiat în container și montat ca volum, fiind poziționat în directorul corespunzător pentru rulare automată. Configurația definește rutele pentru *user-auth-ms*, precum și *time-out-uri*, metode permise și setările pentru *CORS*. *Envoy* ascultă cereri pe portul *8000* și funcționează ca un intermediar transparent între *NGINX* și serviciile *gRPC*.

4.1.6. Configurarea aplicației frontend

Aplicația de *frontend* este dezvoltată în *React*, folosind *TypeScript*, și este rulată local prin comanda *npm start*, care pornește serverul de dezvoltare pe portul *3000*. Aceasta nu este containerizată, ci este pornită manual în timpul dezvoltării, fiind conectată la infrastructura *backend* prin *NGINX* și *Envoy*. Configurația de bază este gestionată de sistemul *react-scripts*, fără personalizări majore în *setup-ul Webpack* sau *Babel*. În ceea ce privește dependințele, aplicația include câteva biblioteci suplimentare față de configurația *React* implicită, în special pentru a susține integrarea cu *backendul gRPC*. Se utilizează *grpc-web* și *@improbable-eng/grpc-web* pentru comunicarea cu serviciile *gRPC backend* în combinație cu *google-protobuf*. De asemenea, aplicația folosește *axios* pentru apeluri *REST* și *react-hot-toast* pentru notificări, *framer-motion* pentru animații și *@headlessui/react* împreună cu *@heroicons/react* pentru componente *UI* accesibile și moderne.

Pentru stilizare, este integrat *framework-ul tailwindcss*, împreună cu *postcss* și *autoprefixer*, oferind o experiență de dezvoltare rapidă și un design coerent, adaptabil. Tipurile *TypeScript* pentru biblioteci importante sunt incluse explicit *(@types/react, @types/node*, etc.), asigurând o dezvoltare

sigură din punct de vedere al tipurilor.

4.2. Testarea sistemului

Pentru a asigura fiabilitatea fiecărei componente dezvoltate, sistemul de testare a fost structurat în două etape principale: monitorizarea comportamentului în execuție prin *loguri* și validarea manuală a funcționalităților expuse prin *endpoint-uri REST* și *gRPC*. Fiecare microserviciu din arhitectura propusă este prevăzut cu un mecanism propriu de logare, adaptat limbajului și *framework-ului* utilizat. În cazul microserviciilor *Java* (ex: *booking* și *service-catalog*), sistemul de *loguri* este configurat prin *logback-spring.xml*, care definește atât logarea în consolă, cât și salvarea în fișiere zilnice rotative. Pentru microserviciul *Python user-auth*, este utilizat un *logger* personalizat (*logger.py*) ce scrie mesaje în fișiere datate, cu format standardizat și păstrare timp de șapte zile. În cazul microserviciului *.NET user-management*, a fost integrat *Serilog*, care preia configurațiile din *appsettings.Production.json* și oferă un sistem flexibil de logare în consolă, ideal pentru rularea containerizată. Aceste sisteme sunt esențiale în etapa de testare integrată, deoarece permit observarea în timp real a fluxurilor de cereri și identificarea rapidă a erorilor sau comportamentelor neașteptate, direct în terminalul *Docker*.

După implementarea fiecărei funcționalități, *endpoint-urile* au fost testate individual utilizând *Postman*, unde au fost organizate în foldere separate corespunzătoare fiecărui microserviciu: *booking_microservice, service_catalog_microservice, user_auth_microservice* și *user_management_microservice*. Fiecare *folder* conține cereri detaliate pentru toate operațiile relevante (ex: *POST Create Booking, PATCH Set Cancelled Status, GET By Client Id* etc.), fiind configurate cu *payload-uri* reprezentative și *Bearer Token-uri* valide. Pentru microserviciul *user-auth*, fiind expus prin *gRPC*, *Postman* a fost folosit în modul *gRPC*, iar cererile precum *Login, SignUp, RefreshToken* sau *ValidateJwt* au fost definite folosind fișierele *.proto* corespunzătoare. Răspunsurile au fost verificate atât în format brut, cât și în secțiunea de *metadate* și coduri de stare.

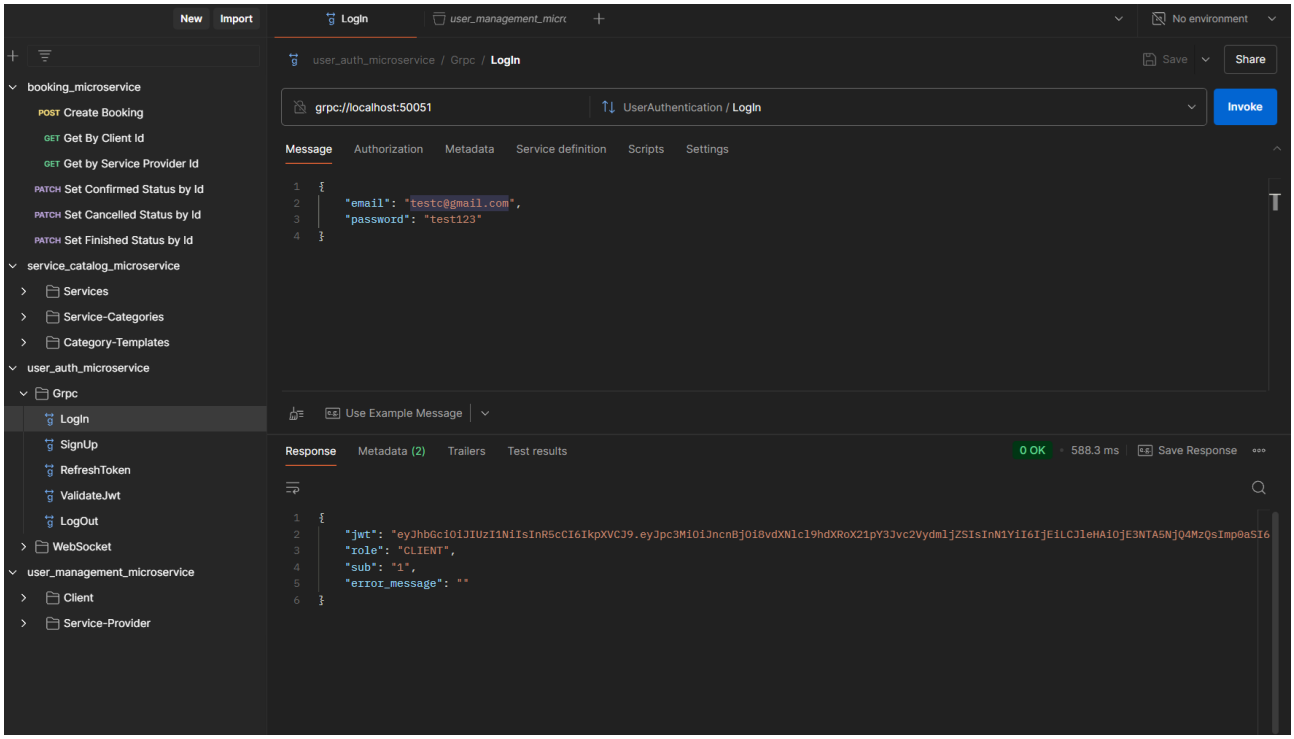


Figura 15: Testare call gRPC LogIn

Capitolul 5. Concluzii

Proiectul realizat își propune să ofere o soluție digitală modernă pentru intermedierea serviciilor de *home maintenance*, într-un context în care digitalizarea acestui sector este încă într-un stadiu incipient. Prin această lucrare am propus un sistem distribuit, containerizat și scalabil, care răspunde concret nevoii de a centraliza într-un singur spațiu virtual interacțiunea dintre clienți și furnizorii de servicii, un demers care contribuie la modernizarea și eficientizarea accesului la servicii specifice. Platforma dezvoltă un ecosistem complet, care digitalizează întregul ciclu de utilizare: de la înregistrare și autentificare, până la publicarea serviciilor, căutarea și rezervarea acestora, oferind în același timp un *UI* personalizat în funcție de rolul utilizatorului.

Contribuția personală se reflectă în arhitectura microserviciilor, integrarea tehnologiilor moderne și modul de abordare al comunicării interservicii. Fiecare componentă, *frontend-ul* în *React*, microserviciile în *.NET, Python* și *Spring Boot*, a fost dezvoltată și orchestrată manual, cu accent pe învățarea și înțelegerea detaliată a protocoalelor implicate (*gRPC*, *REST*, *WebSocket*). Alegerea acestor tehnologii nu a fost întâmplătoare: ele sunt astăzi standarde în industrie și sunt utilizate în companii de top, ceea ce asigură atât relevanța practică, cât și posibilitatea extinderii în contexte reale. *Logging-ul* integrat, autentificarea robustă, gestionarea stării rezervărilor și validările distribuite contribuie la un sistem sigur și coerent, iar rezultatele obținute demonstrează că arhitectura propusă este viabilă și adaptabilă. Testarea modulară și apoi integrată a evidențiat faptul că sistemul funcționează corect și eficient în condiții de orchestrare *Docker*. Interfața intuitivă și designul vizual construit în jurul nevoilor reale ale utilizatorilor, clienți sau furnizori, susțin experiența utilizatorului și răspund principiilor de accesibilitate și claritate în interacțiune. Soluția oferă astfel un cadru clar pentru digitalizarea serviciilor locale, într-un mod accesibil atât tehnic, cât și economic.

În ceea ce privește direcțiile de dezvoltare ulterioare, platforma poate fi extinsă cu funcționalități avansate precum integrarea plăților electronice, gestionarea recenziilor și a ratingurilor, notificări automate sau chiar recomandări personalizate pe baza unui algoritm de învățare. De asemenea, infrastructura poate fi migrată în *cloud* și monitorizată cu sisteme dedicate (ex: *Prometheus, Grafana, ELK stack*). Într-o piață în care digitalizarea devine o necesitate, platforma *UrHomie* propune un exemplu viabil de soluție scalabilă, modulară și aliniată la standardele tehnologice actuale.

Bibliografie

- [1] Issue Monitoring, „Romania's Digital Environment: Navigating the Path to a Tech-DrivenFuture", https://issuemonitoring.eu/en/romanias-digital-environment-navigating-the-path-to-a-tech-driven-future/
- [2] plego, „The Importance of User Interface Design", https://plego.com/blog/importance-of-user-interface-design/
- [3] https://github.com/
- [4] TRAILHEAD, „When to Use a Distributed Architecture—And When Not", https://trailheadtechnology.com/when-to-use-a-distributed-architectureand-when-not/
- [5] saleforce, „6 Fundamental Principles of Microservice Design", https://www.salesforce.com/blog/microservice-design-principles/
- [6] DigitalOcean, „SOLID: The First 5 Principles of Object Oriented Design", https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- [7] komprise, „REST (Representational State Transfer)", https://www.komprise.com/glossary_terms/rest-representational-state-transfer/
- [8] Journal of Student Research, „Home Repairs: Mobile Application for Home Maintenance Services", https://www.jsr.org/index.php/path/article/view/1500/
- [9] https://home-maintenance.ro/
- [10] https://servicii24.ro/
- [11] https://www.taskrabbit.com/
- [12] https://www.handy.com/
- [13] Github, „grpc/grpc-web", https://github.com/grpc/grpc-web
- [14] React, „Use the best from every platform**", https://react.dev/**
- [15] Envoy proxy – home, „Why Envoy", https://www.envoyproxy.io/
- [16] ASP.NET documentation | Microsoft Learn, „ASP.NET documentation", https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0
- [17] Spring Boot :: Spring Boot, „Spring Boot", https://docs.spring.io/spring-boot/

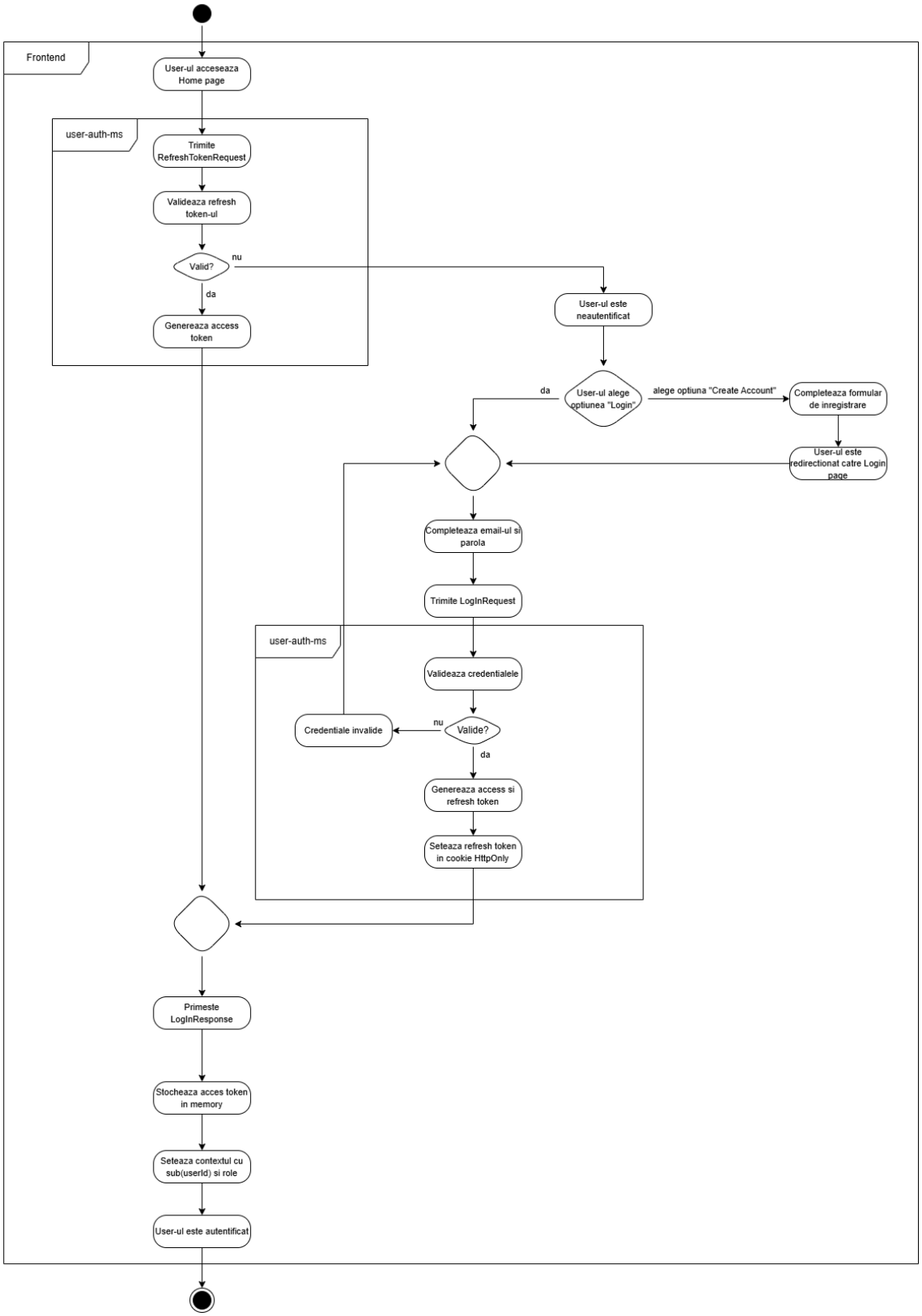
Anexe

Anexa 1: Fișier .proto pentru compilare în React

```
syntax = "proto3";

service UserAuthentication {
  rpc LogIn(LogInRequest) returns (LogInResponse);
  rpc ValidateJwt(ValidateJwtRequest) returns (ValidateJwtResponse);
  rpc SignUp(SignUpRequest) returns (SignUpResponse);
  rpc RefreshToken(Empty) returns (LogInResponse);
  rpc LogOut(Empty) returns (Empty);
}

message LogInRequest {
  string email = 1;
  string password = 2;
}

message LogInResponse {
  string jwt = 1;
  string role = 2;
  string sub = 3;
  string error_message = 4;
}

message ValidateJwtRequest {
  string jwt = 1;
}

message ValidateJwtResponse {
  bool is_valid = 1;
  string sub = 2;
  string role = 3;
  string error_message = 4;
}

message SignUpRequest {
  string email = 1;
  string password = 2;
  string role = 3;

  string first_name = 4;
  string last_name = 5;
  string phone_number = 6;
  string country = 7;
  string city = 8;
  string address = 9;

  oneof profile_data {
    ClientDetails client = 10;
    ServiceProviderDetails provider = 11;
```

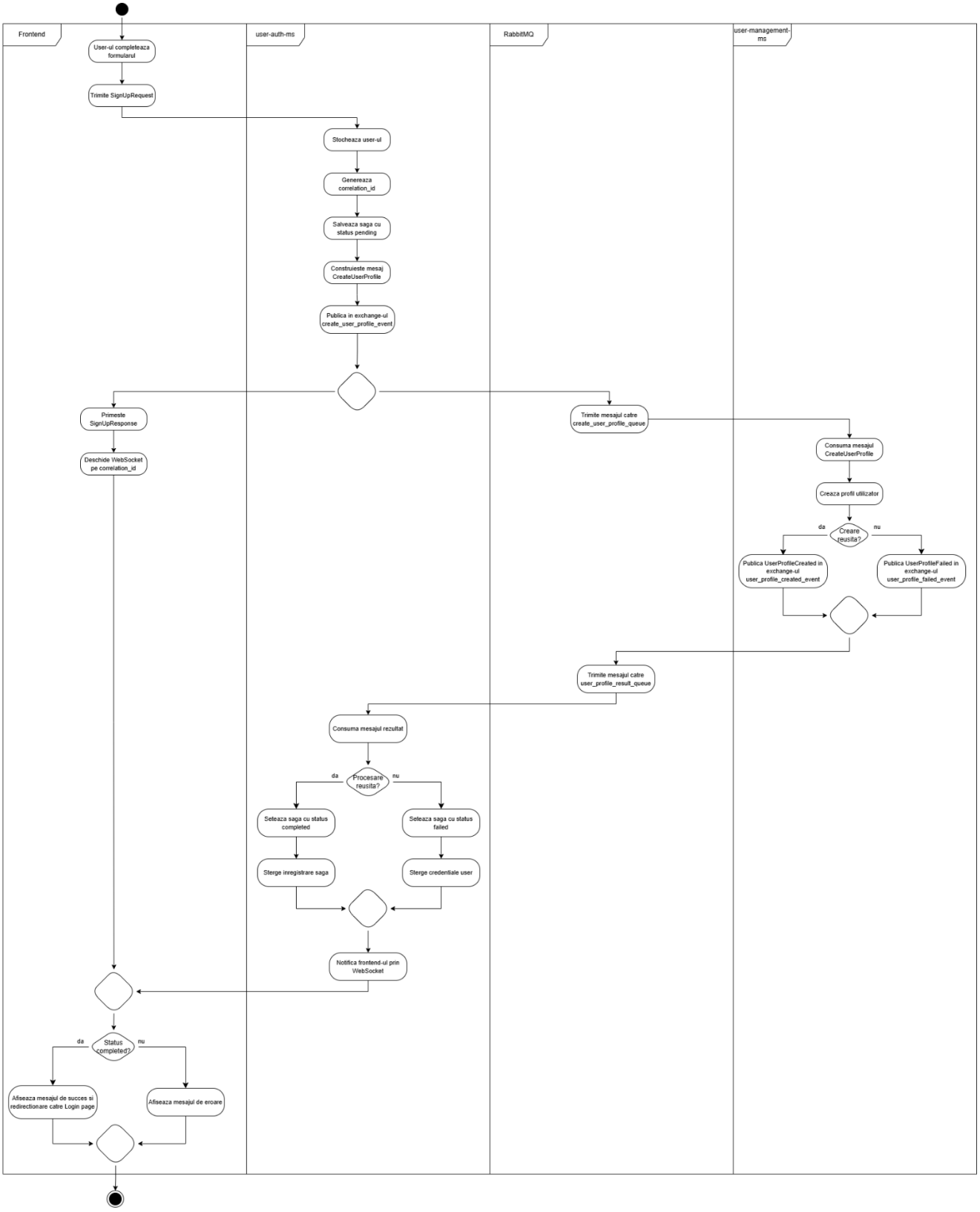
Anexa 2: Diagramă de activitate pentru procesul de autentificare și acces bazat pe token-uri

Diagrama de activitate pentru procesul de autentificare și acces bazat pe tokenuri



Anexa 3: Diagramă de activitate pentru procesul de înregistrare distribuit

Diagrama de activitate pentru procesul de inregistrare distribuit



Anexa 4: Configurație NGINX

```
http {
    include mime.types;
    default_type application/octet-stream;

    map $request_method $cors_preflight {
        default 0;
        OPTIONS 1;
    }

    server {
        listen 80;

        set $cors_origin 'http://localhost:3000';
        set $cors_methods 'GET, POST, OPTIONS, PUT, DELETE, PATCH';
        set $cors_headers 'Origin, Content-Type, Accept, Authorization';
        set $cors_expose_grpc 'grpc-status, grpc-message, grpc-encoding, grpc-accept-encoding, x-user-agent, x-grpc-web';

        location /UserAuthentication/ {
            if ($cors_preflight) {
                add_header 'Access-Control-Allow-Origin' $cors_origin always;
                add_header 'Access-Control-Allow-Methods' $cors_methods always;
                add_header 'Access-Control-Allow-Headers' "$cors_headers, $cors_expose_grpc" always;
                add_header 'Access-Control-Allow-Credentials' 'true' always;
                add_header 'Content-Length' 0;
                add_header 'Content-Type' 'text/plain; charset=UTF-8';
                return 204;
            }

            add_header 'Access-Control-Allow-Origin' $cors_origin always;
            add_header 'Access-Control-Allow-Methods' $cors_methods always;
            add_header 'Access-Control-Allow-Headers' "$cors_headers, $cors_expose_grpc" always;
            add_header 'Access-Control-Allow-Credentials' 'true' always;

            proxy_pass http://user-auth-envoy-proxy:8000;
            proxy_http_version 1.1;
            proxy_set_header Host $host;
            proxy_pass_request_headers on;
        }

        location /user-auth/registration-status {
            add_header 'Access-Control-Allow-Origin' $cors_origin always;
            add_header 'Access-Control-Allow-Credentials' 'true' always;

            proxy_pass http://user-auth-ms:8001;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "Upgrade";
            proxy_set_header Host $host;
```

Anexa 5: Implementare metodă LogIn pe server-ul gRPC

```python
import grpc
import uuid

from src.infrastructure.messaging.message_builder import build_create_user_profile_message
from src.shared.logger import get_logger
from src.business.services.grpc.compiled_protos import user_auth_pb2
from src.business.services.grpc.compiled_protos import user_auth_pb2_grpc

logger = get_logger(__name__)

class UserAuthServer(user_auth_pb2_grpc.UserAuthenticationServicer):
    def __init__(self, user_repo, jwt_service, event_bus, saga_manager):
        self.user_repo = user_repo
        self.jwt_service = jwt_service
        self.event_bus = event_bus
        self.saga_manager = saga_manager

    async def LogIn(self, request, context):
        email, password = request.email, request.password
        logger.info(f"Received LogIn request for email: {email}")

        user = self.user_repo.validate_credentials(email, password)
        if not user:
            logger.warning(f"Authentication failed for email: {email}.")
            return user_auth_pb2.LogInResponse(jwt="", sub="", role="", error_message="Invalid email or password!")

        access_token = self.jwt_service.generate_token(user, expires_in=900)
        refresh_token = self.jwt_service.generate_token(user, expires_in=604800, refresh=True)

        await context.send_initial_metadata((
            ('set-cookie', f'refresh_token={refresh_token}; HttpOnly; Path=/; Max-Age=604800; SameSite=Lax'),
        ))

        logger.info(f"Authentication successful for {email}")
        return user_auth_pb2.LogInResponse(
            jwt=access_token,
            sub=str(user.id),
            role=user.role,
            error_message=""
        )

```

Anexa 6: Implementare consumer de evenimente

```
using MassTransit;
using Microsoft.EntityFrameworkCore;
using MySqlConnector;
using user_management_microservice.Application.Services.Interfaces;
using user_management_microservice.Infrastructure.EventBus.Messages;

namespace user_management_microservice.Infrastructure.EventBus.Consumers;

public class CreateUserProfileConsumer(
    IClientService clientService,
    IServiceProviderService serviceProviderService,
    IPublishEndpoint publishEndpoint,
    ILogger<CreateUserProfileConsumer> logger)
    : IConsumer<CreateUserProfileMessage>
{
    public async Task Consume(ConsumeContext<CreateUserProfileMessage> context)
    {
        var message = context.Message;

        logger.LogInformation("Received CreateUserProfileMessage for UserId={UserId} | CorrelationId={CorrelationId}",
            message.UserId, context.CorrelationId!.Value);

        try
        {
            if (message.Client is not null)
            {
                logger.LogInformation("Detected user as Client. Attempting to create...");

                var client = await clientService.AddClientAsync(message);

                if (client is not null)
                {
                    logger.LogInformation("Client successfully created for UserId={UserId}", message.UserId);

                    await publishEndpoint.Publish(new UserProfileCreated
                    {
                        CorrelationId = context.CorrelationId!.Value
                    });

                    logger.LogInformation("Published UserProfileCreated for UserId={UserId}", message.UserId);
                }
                else
                {
                    logger.LogWarning("Client creation returned null for UserId={UserId}", message.UserId);

                    await publishEndpoint.Publish(new UserProfileCreationFailed
                    {
                        CorrelationId = context.CorrelationId!.Value,
```

Anexa 7: Configurare docker-compose.yaml

```yaml
services:
  Run Service
  user-auth-ms:
    build:
      context: user_auth_microservice
    ports:
      - '50051:50051'
      - '8001:8001'
    networks:
      - user-auth-db
      - backend
    depends_on:
      user-auth-mariadb:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
      user-auth-redis:
        condition: service_healthy

  Run Service
  user-auth-mariadb:
    image: mariadb:latest
    container_name: user-auth-mariadb
    environment:
      MYSQL_DATABASE: user-auth
      MYSQL_ROOT_PASSWORD: root-pass
      MYSQL_USER: user-auth-manager
      MYSQL_PASSWORD: manager-pass
    ports:
      - '3306:3306'
    volumes:
      - ./user_auth_microservice/src/persistence/config/initdb/:/docker-entrypoint-initdb.d
      - user_auth_mariadb_data:/var/lib/mysql
    networks:
      - user-auth-db
    healthcheck:
      test: ["CMD", "mariadb", "-h", "localhost", "-u", "user-auth-manager", "-pmanager-pass", "-e", "SELECT 1"]
      interval: 30s
      retries: 5
      start_period: 30s
      timeout: 20s

  Run Service
  user-auth-redis:
    image: redis:alpine
    ports:
      - "6379:6379"
    networks:
      - user-auth-db
```