Notificare în timp real

Cătălina Racolța-Maria

Ianuarie 2026

Contents

1 Introducere

Am ales această temă deoarece ritmul de viață actual a transformat comanda online de mâncare într-o necesitate zilnică. M-am gândit să implementez o aplicație de "Food Delivery" robustă, care să gestioneze fluxul comenzilor rapid și sigur, eliminând frustrarea cauzată de platformele care se blochează în momentele de trafic intens.

Aplicația propusă permite utilizatorilor să vizualizeze meniuri și să plaseze comenzi, garantând, prin arhitectura sa distribuită, că nicio cerere nu se pierde între client și bucătărie. Utilizarea cozilor de mesaje asigură o procesare asincronă eficientă, decuplând primirea comenzii de prepararea ei efectivă.

1.1 Obiective

M-am gândit ca aplicația să fie ușor de utilizat, astfel utilizatorii accesează site-ul și își aleg rolul pe care aceștia îl au, adică pot fi "client" sau poate fi "restaurant". După ce și-au ales rolul, clientul o să vadă o pagină cu meniuri, iar restaurantul vede comenzile plasate și le preia.

Prin dezvoltarea acestei aplicații se urmărește atingerea următoarelor obiective:

1. Utilizatorul să poată să vizualizeze meniul și să plaseze comanda.
2. Restaurantul să preia comenzile și să pună la fiecare comandă stagiul în care aceasta se află.

1.2 Schema MoSCoW

Schema sau metoda MoSCoW are rolul de prioritizare a cerințelor, asigurând implementarea funcționalităților de bază, de care este obligatoriu nevoie, mai apoi în cazul în care este nevoie, având posibilitatea de a adăuga unele cerințe opționale.

Must have	Should have	Could have	Won't have
Fluxul Core Distribuit	Persistența Datelor	Autentificare	Tracking GPS Real
Separarea Rolurilor	Gestionarea Erorilor	Meniu Dinamic	Aplicație Mobilă Nativă
Dockerizare		Simulare Plată	Gateway de Plată Real
Status Flow			Microservicii separate fizic

Figure 1: MoSCoW

1.3 Cazuri de utilizare

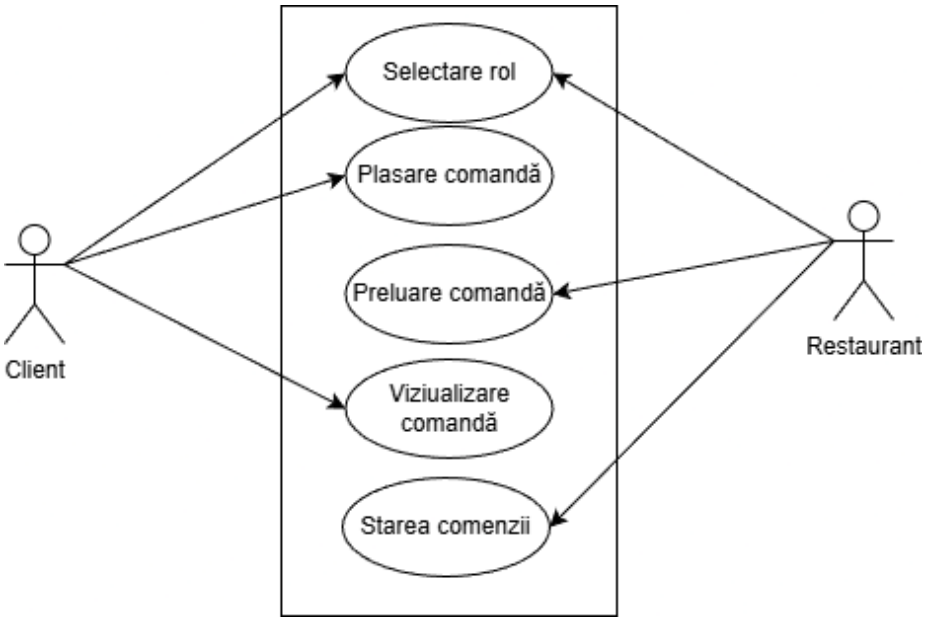

Figure 2: Cazuri de utilizare

2 Arhitectura

Arhitectura aleasă este una de tip Event-Driven (Bazată pe Evenimente). Am eliminat dependența de o bază de date sincronă pentru a crește performanța. Fluxul este următorul: Clientul trimite comanda către Controller, care acționează doar ca un punct de intrare și trimite imediat sarcina către RabbitMQ. Aceasta asigură decuplarea componentelor. Un Listener separat preia mesajul din coadă și, prin intermediul protocolului WebSocket (STOMP), notifică instantaneu toate interfețele conectate (atât Restaurantul cât și Clientul), realizând astfel un sistem reactiv în timp real.

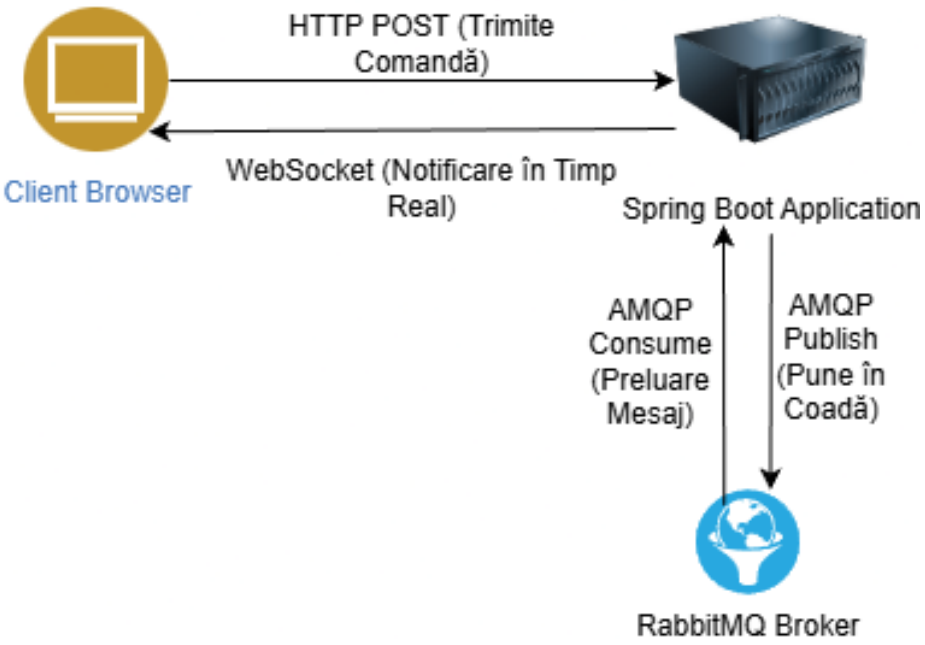

Figure 3: Arhitectura

2.0.1 Frontend-ul (Client Side)

Interfața grafică este realizată ca o aplicație SPA (Single Page Application) simulată.

S-a utilizat Bootstrap 5 pentru a asigura un design responsive (adaptabil pe mobil/desktop) și o experiență de utilizare fluidă (UX). Logica din browser este scrisă în JavaScript, folosind biblioteca SockJS și Stomp.js. Acestea sunt esențiale pentru a menține o conexiune persistentă cu serverul, permițând actualizarea DOM-ului (Document Object Model) în timp real, fără reîncărcarea paginii

2.0.2 Backend-ul (Server Side)

Serverul este construit pe framework-ul Spring Boot, care gestionează logica aplicației.

Controller REST: Clasa DeliveryController expune endpoint-ul /send care acceptă cereri HTTP POST. Acesta acționează ca punct de intrare pentru comenzile noi. WebSocket Broker: Prin clasa WebSocketConfig, serverul configurează un broker de mesaje în memorie (SimpleBroker), la adresa /topic, permițând trimiterea de date către abonați (clienți).

2.0.3 Comunicare între Frontend și Backend

Interacțiunea dintre Frontend și Backend se realizează prin două canale paralele:

1. Canalul de Comandă (Synchronous HTTP):
   (a) Direcție: Frontend - Backend
   (b) Când utilizatorul apasă "Trimite Comanda", browserul face un apel fetch() către endpoint-ul /send. Acest apel trimite datele comenzii (JSON) și primește o confirmare că serverul a preluat sarcina.
2. Canalul de Notificare (Asynchronous WebSocket):
   (a) Direcție: Backend - Frontend
   (b) La pornirea paginii, Frontend-ul deschide un socket către /ws. Când Backend-ul primește un eveniment de la sistemul distribuit (RabbitMQ), acesta face "push" la mesaj prin WebSocket. Funcția JS stompClient.subscribe() capturează acest mesaj și actualizează statusul comenzii pe ecran instantaneu.

2.0.4 Beneficii ale acestei arhitecturii

1. Arhitectura asigură decuplarea completă a componentelor, ceea ce înseamnă că modulul de preluare a comenzilor funcționează independent de cel de livrare, permițând modificarea sau mentenanța lor separată fără a opri întregul sistem.

2. Utilizarea comunicării asincrone prin RabbitMQ optimizează timpii de răspuns, deoarece serverul confirmă primirea comenzii instantaneu către client, eliberând interfața, în timp ce procesarea complexă are loc în fundal.

3. Sistemul oferă o scalabilitate orizontală nativă, permițând adăugarea de noi instanțe de consumatori pentru a gestiona automat vârfurile de trafic, fără a necesita modificări ale codului sursă.

4. Toleranța la erori este garantată de brokerul de mesaje, care acționează ca un buffer temporar și păstrează comenzile în siguranță chiar dacă serviciul de procesare devine temporar indisponibil.

5. Experiența utilizatorului este îmbunătățită prin tehnologia WebSocket, care elimină necesitatea reîncărcării paginii (polling) prin trimiterea notificărilor de status în timp real (Server-Push) direct către interfața grafică.

6. Arhitectura de tip Event-Driven eficientizează consumul de resurse, deoarece procesarea datelor are loc doar la apariția unui eveniment (comandă nouă), evitând astfel verificările repetitive și inutile asupra bazei de date sau a serverului.

3 Implementare frontend și backend

Implementarea sistemului GlovoDist a fost realizată urmând principiile unei arhitecturi moderne, distribuite, cu o separare clară a responsabilităților între interfața utilizator (Frontend) și logica de business (Backend). Obiectivul principal a fost crearea unei aplicații reactive, capabilă să gestioneze fluxuri de date asincrone fără a bloca experiența utilizatorului.

3.1 Implementarea Frontend (Interfața Utilizator)

Partea de client a fost dezvoltată ca o Single Page Application (SPA) simulată, punând accent pe viteza de reacție și feedback vizual instantaneu.

1. Design și Structură: S-a utilizat HTML5 pentru structura semantică și framework-ul Bootstrap 5 pentru stilizare. Aceasta asigură un design responsive, adaptabil atât pe desktop cât și pe dispozitive mobile, element crucial pentru actorii din sistem (clienți și curieri). Componentele vizuale (carduri de produse, dashboard-ul restaurantului) sunt manipulate dinamic în DOM.

2. Logica Reactivă: Elementul inovator al frontend-ului este integrarea bibliotecilor SockJS și Stomp.js. Acestea permit browserului să deschidă un canal de comunicație persistent (WebSocket) cu serverul. Astfel, interfața nu necesită reîncărcare (refresh) pentru a afișa starea comenzii; scripturile JavaScript ascultă activ evenimentele trimise de backend și actualizează elementele vizuale în timp real.

3.2 Implementarea Backend (Logica de Business)

Serverul a fost construit pe ecosistemul Spring Boot, ales pentru robustețea sa în mediul enterprise și ușurința de integrare cu sisteme de mesagerie.

1. Arhitectura Event-Driven: Backend-ul nu funcționează ca un simplu server web tradițional, ci ca un orchestrator de evenimente. Acesta expune un API REST (/send) pentru primirea datelor, dar procesarea efectivă este delegată.

2. Integrarea cu RabbitMQ: Prin utilizarea Spring AMQP, aplicația Java acționează dual:

   (a) Ca Producer: Preia comanda HTTP și o injectează în sistemul distribuit.

   (b) Ca Consumer: Un serviciu de fundal (RabbitListener) preia mesajele procesate din coadă și le pregătește pentru livrarea către client.

3. WebSocket Broker: Backend-ul configurează un broker de mesaje în memorie, care servește drept punte de legătură între evenimentele interne ale sistemului distribuit și interfața utilizatorului.

Această abordare hibridă (REST pentru comenzi + WebSocket pentru notificări) asigură un echilibru optim între fiabilitatea transmisiei datelor și experiența fluidă a utilizatorului.

3.3 Structura componentelor

Proiectul este organizat conform standardului Maven pentru aplicații Java Spring Boot, separând clar codul sursă (Java) de resursele statice (Frontend) și fișierele de configurare.

3.3.1 Componenete folosite

Prinicipalele componente folosite sunt:

Numele componentei	Rolul componentei	Utilizată la endpoint-ul
DeliveryController (Java)	Primește comanda prin HTTP POST și o trimite spre RabbitMQ.	**/send**
WebSocketConfig (Java)	Configurează punctul de intrare (Handshake) pentru conexiunea WebSocket.	**/ws**
SimpleBroker (Java/Spring)	Canalul intern de "Topic" unde serverul publică notificările pentru clienți.	**/topic/alerts**
sendOrder() (JavaScript)	Funcția din browser care împachetează datele și inițiază cererea de comandă.	**/send**
SockJS Client (JavaScript)	Inițiază conexiunea persistentă cu serverul dacă WebSocket e suportat.	**/ws**
stompClient.subscribe() (JS)	Funcția care ascultă (se abonează) la fluxul de mesaje din server.	**/topic/alerts**
RabbitConfig (Java)	Configurează infrastructura internă (Nu este expus public prin HTTP).	*Intern (AMQP)*

Figure 4: Componente folosite

3.4 Apache Maven

Maven este un instrument standard în industria Java pentru automatizarea procesului de compilare (Build Automation) și gestionarea dependențelor. Eu l-am folosit deoarece:

1. Gestionarea Dependențelor: În loc să descarc manual fișiere .jar pentru Spring, RabbitMQ sau WebSocket și să le adaug în proiect, Maven le descarcă automat din repository-ul central pe baza fișierului pom.xml.

2. Structură Standard: Maven impune o structură clară a proiectului (src/main/java, src/main/resources), ceea ce face codul ușor de înțeles și navigat pentru orice dezvoltator Java.

3. Build Lifecycle: Simplifică procesul de rulare a aplicației prin comenzi simple (ex: mvn spring-boot:run sau mvn clean install).

3.5 Spring Boot

Un framework care simplifică dezvoltarea aplicațiilor Java, eliminând necesitatea configurărilor complexe (boilerplate code). Eu l-am folosit deoarece:

1. Server Embedded: Spring Boot include un server web Tomcat integrat. Astfel, aplicația rulează ca un simplu fișier .jar executabil, fără a fi nevoie să instalez și să configurez un server de aplicații separat.

2. Auto-Configuration: Detectează automat bibliotecile din classpath. De exemplu, pentru că am adăugat dependența spring-boot-starter-amqp, Spring Boot a configurat automat conexiunea către RabbitMQ, scutindumă de scrierea a zeci de linii de cod de infrastructură.

3. Productivitate: Mi-a permis să expun rapid endpoint-uri REST (DeliveryController) și să integrez WebSocket cu adnotări simple (@RestController, @EnableWebSocketMessageBroker).

3.6 RabbitMQ

Un Message Broker (Intermediar de mesaje) open-source, care implementează protocolul AMQP (Advanced Message Queuing Protocol). Eu l-am folosit deoarece:

1. Decuplare și Asincronism: Este componenta critică pentru caracterul "distribuit" al sistemului. Permite aplicației să trimită o comandă și să continue execuția imediat, fără să aștepte procesarea ei.

2. Fiabilitate (Reliability): RabbitMQ garantează că mesajele nu se pierd. Dacă consumatorul (partea de livrare) este oprit temporar, mesajele rămân stocate în coada delivery queue și sunt livrate imediat ce serviciul repornește.

3. Buffer de Trafic: Protejează sistemul în cazul unor vârfuri de sarcină, stocând comenzile în coadă și permițând procesarea lor secvențială.

3.7 WebSocket (cu SockJS și STOMP)

Un protocol de comunicație care oferă un canal full-duplex (bidirecțional) între client și server printr-o singură conexiune TCP. Eu l-am folosit deoarece:

1. Feedback în Timp Real: Spre deosebire de HTTP, unde clientul trebuie să ceară date (Polling), WebSocket permite serverului să facă Server-Push. Astfel, statusul comenzii ("În preparare", "Livrat") se actualizează pe ecranul clientului instantaneu, fără ca acesta să dea refresh la pagină.

2. Eficiență: Reduce traficul de rețea, eliminând cererile HTTP repetitive și inutile.

3. Compatibilitate: Am folosit SockJS ca fallback (dacă browserul nu suportă WebSocket nativ, trece automat pe HTTP Long Polling) și STOMP pentru a defini un format standard al mesajelor.

3.8 Docker

O platformă de containerizare care permite împachetarea aplicațiilor și a dependențelor lor în unități standardizate numite containere. Eu l-am folosit deoarece:

1. Portabilitate: Asigură că proiectul rulează identic pe orice calculator (Windows, Linux, macOS), eliminând problema ”la mine pe calculator mergea”.

2. Infrastructură Rapidă: Mi-a permis să pornesc instanța de RabbitMQ într-un mediu izolat în câteva secunde, fără a fi nevoie să instalez Erlang și RabbitMQ server direct în sistemul de operare gazdă.

3.9 Bootstrap 5

Cel mai popular framework CSS frontend pentru dezvoltarea de interfețe responsive și mobile-first. Eu l-am folosit deoarece:

1. Viteza de Dezvoltare: Oferă componente pre-construite (Carduri pentru produse, Modale pentru checkout, Bare de progres) care arată profesional ”out-of-the-box”.

2. Responsive Design: Sistemul de Grid (col-md-3, row) asigură că interfața se adaptează automat, arătând bine atât pe laptopul managerului de restaurant, cât și pe telefonul clientului.

3.10 Folosirea aplicației

Se începe prin a selecta rolul de client sau restaurant.

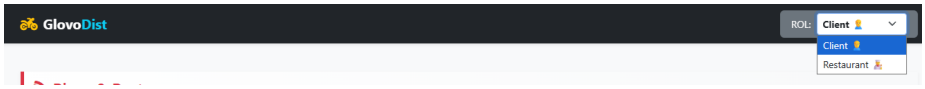

Figure 5: selectare client/restaurant

După ce a fost selectat clientul, pe pagina acestuia se va vedea meniurile și diferitele feluri de mâncare.

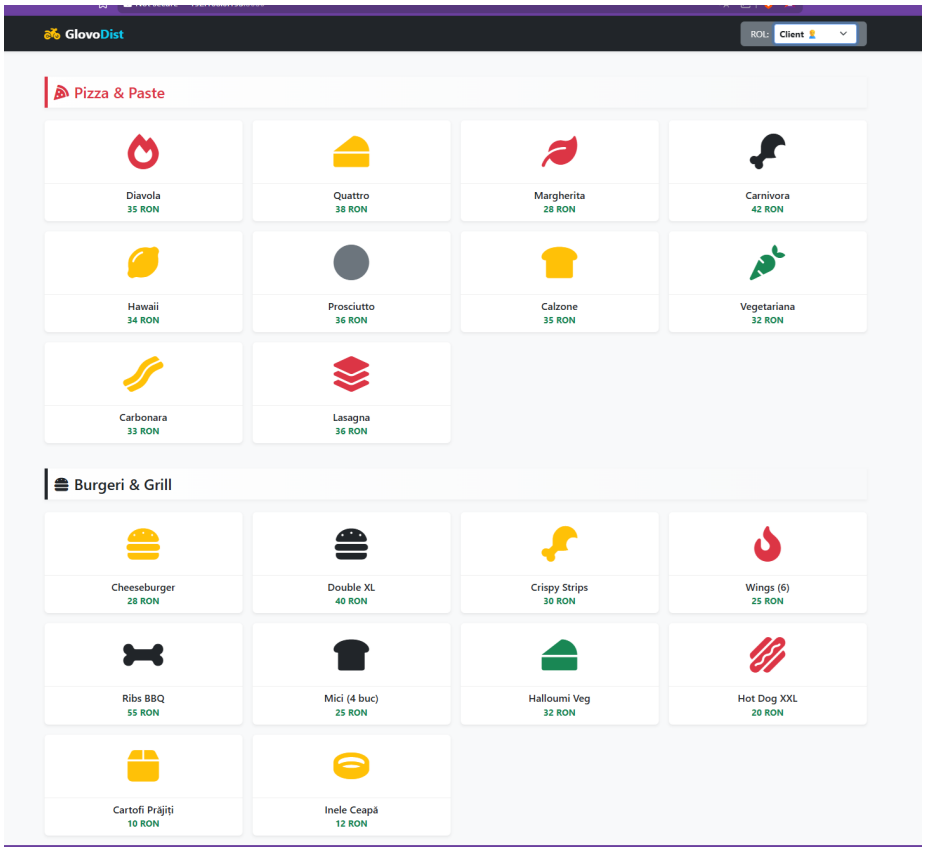

Figure 6: meniu client

Clientul o să își poată selecta ce doreste să comande, după care, în partea dreapta o să îi apară ce are în coș pus și o să poată finaliza comanda. Finzalizarea comenzii se face prin adăugarea numelui, adresei și eventual alergeni, după care se apasă pe "Trimite comanda", iar dacă clientul doreste să mai selecteze și să mai comande ceva, are buton de "Mai comand" care îi oferă posibilitatea de a mai pune în coș.

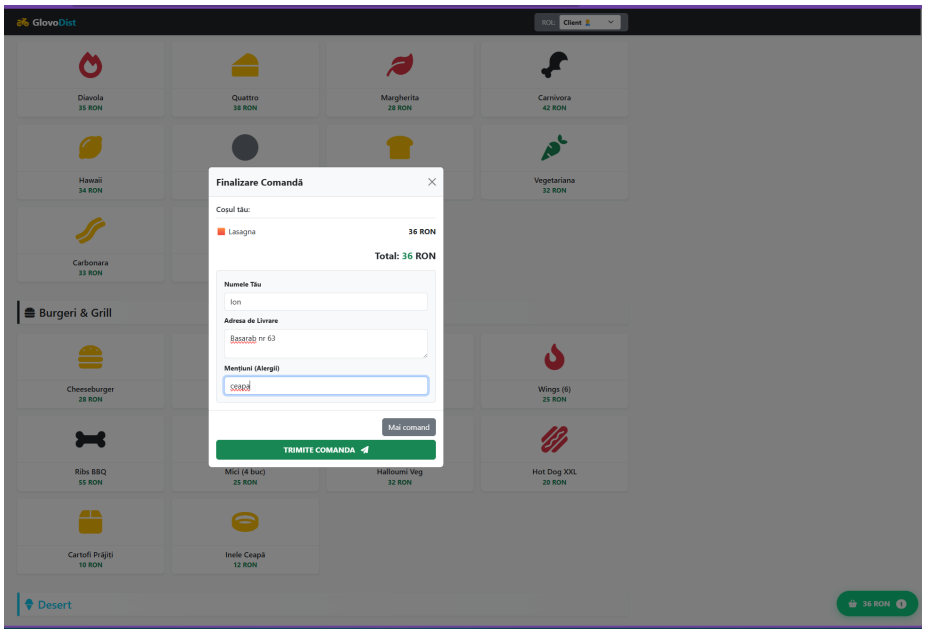

Figure 7: comandă

După ce clientul a finalizat comanda, aceasta o să apară pe pagina restaurantului. Acesta o să poată să accepte comanda și să o prepare și să puna un timp de asteptare, iar la client o să îi apară că i se gătește comanda. După care restaurantul pune comanda spre curier, iar la client o să îi apară, curierul vine, iar dupa ce comanda a ajuns, restaurantul pune finalizarea comenzi, aceasta dispare din restaurant, iar la client apare ca livrat.

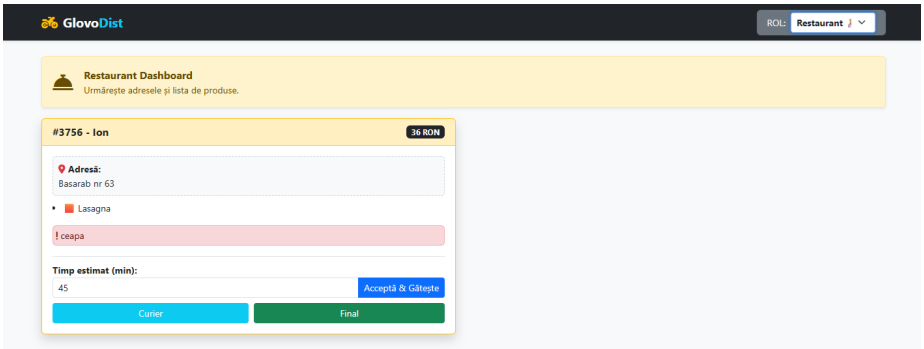

Figure 8: comandă restaurant

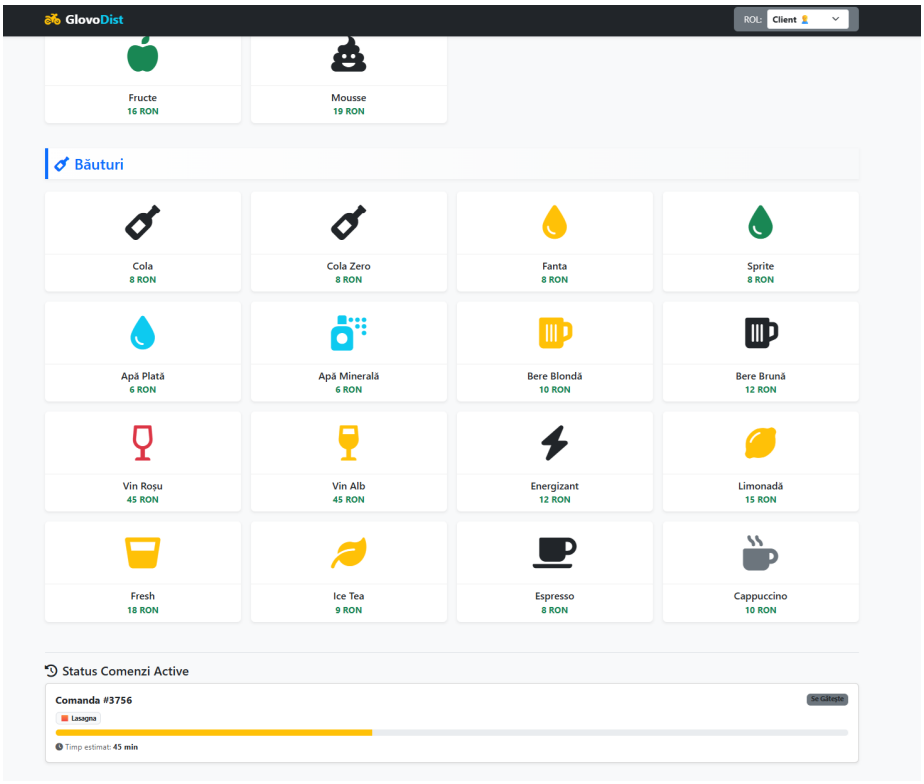

Figure 9: gatire

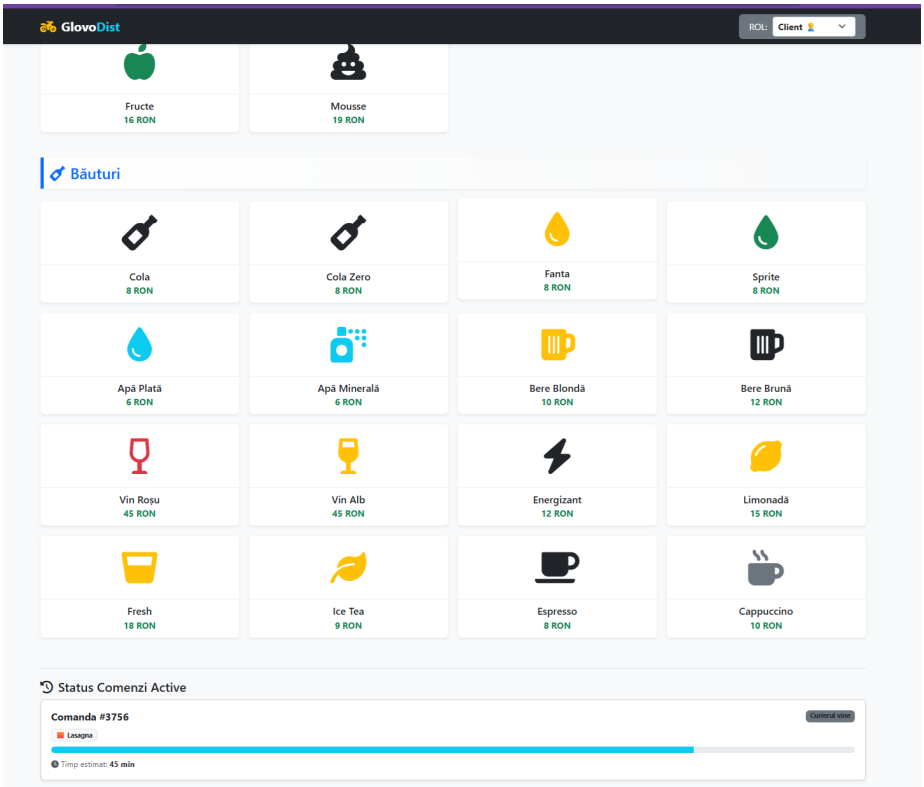

Figure 10: curierul vine

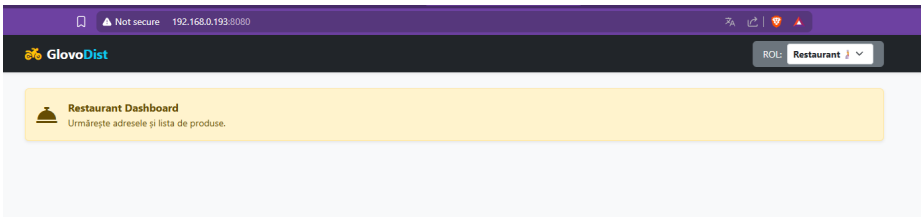

Figure 11: restaurant final

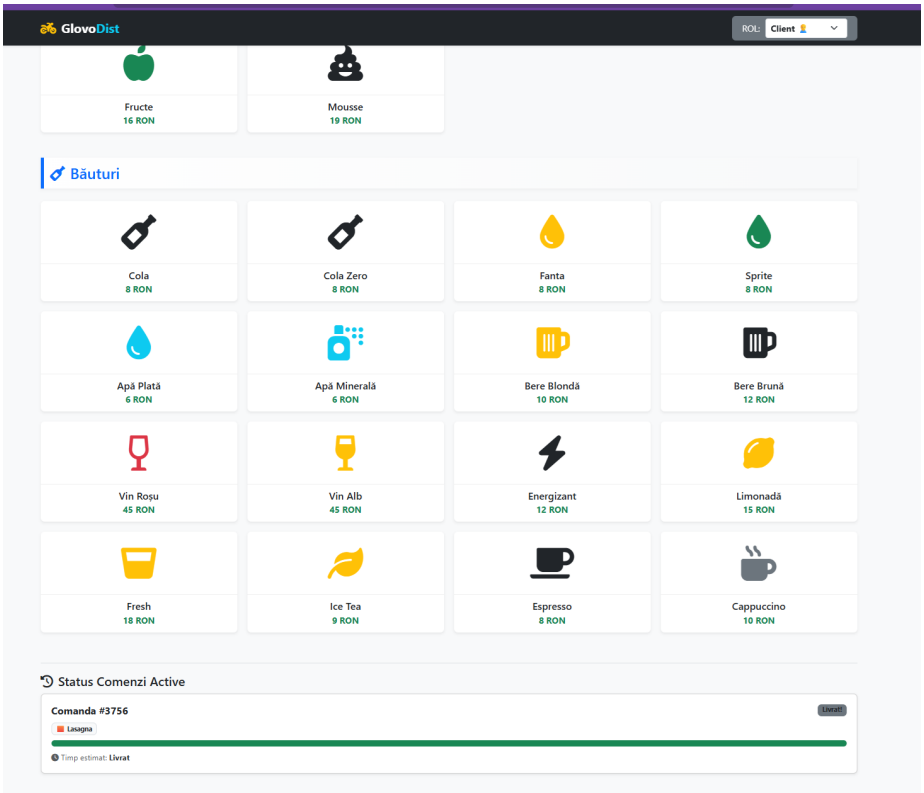

Figure 12: comandă livrată

4 Concluzii

Prin realizarea aplicației GlovoDist, am reușit să implementez un sistem distribuit funcțional care simulează fluxul complet al unei comenzi de livrare în timp real. Proiectul a demonstrat validitatea unei arhitecturi Event-Driven (bazată pe evenimente) pentru aplicațiile moderne care necesită timpi de răspuns rapizi și interactivitate ridicată.

Principala realizare a fost decuplarea logică a componentelor: modulul de preluare a comenzilor nu depinde direct de modulul de procesare, comunicarea realizându-se asincron prin brokerul de mesaje RabbitMQ. Această abordare a eliminat blocajele specifice sistemelor monolitice și a asigurat o toleranță sporită la erori.

Integrarea tehnologiei WebSocket a transformat experiența utilizatorului dintr-una statică într-una reactivă, eliminând nevoia de reîncărcare a paginii pentru verificarea statusului comenzii. Astfel, am demonstrat că protocoalele standard (HTTP) pot coexista eficient cu protocoale de streaming (STOMP) pentru a crea aplicații web performante.

5 Bibliografie

1. VMware, Inc. Spring Boot Reference Documentation.
   Disponibil la: https://docs.spring.io/spring-boot/docs/current/reference/html/
2. VMware, Inc. Messaging with RabbitMQ in Spring.
   Disponibil la: https://spring.io/guides/gs/messaging-rabbitmq/
3. RabbitMQ. AMQP 0-9-1 Model Explained.
   Disponibil la: https://www.rabbitmq.com/tutorials/amqp-concepts.html
4. Docker, Inc. Docker Compose Documentation.
   Disponibil la: https://docs.docker.com/compose/
5. Bootstrap Team. Bootstrap 5.3 Documentation.
   Disponibil la: https://getbootstrap.com/docs/5.3/
6. Baeldung. Intro to WebSockets with Spring.
   Disponibil la: https://www.baeldung.com/websockets-spring
7. Note de curs
   Disponibil la: https://cti.ubm.ro/sdi/