UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Energy Management System

Distributed Systems

Isac Sergiu Ionut
Grupa 30643

Facultatea de Automatica si Calculatoare

31 Octombrie 2024

Cuprins

1 Introducere

Sistemul de Management al Energiei este o aplicație software concepută pentru a ajuta la gestionarea datelor utilizatorilor și a dispozitivelor pentru monitorizarea consumului de energie.

Sistemul permite administratorilor să gestioneze utilizatori și dispozitive prin operațiuni CRUD, iar utilizatorii pot, la rândul lor, să vizualizeze.

Aplicația este dezvoltată folosind o arhitectură bazată pe microservicii, având două back-end-uri Spring Boot – unul pentru gestionarea utilizatorilor și altul pentru dispozitive. Fiecare serviciu este conectat la propria bază de date PostgreSQL și este implementat prin Docker.

Frontend-ul, dezvoltat în Angular, oferă o interfață pentru interacțiunea cu aceste servicii.

2 Tehnologii folosite

Aplicația utilizează un set de tehnologii moderne pentru dezvoltare, administrare și implementare:

- *Backend*: Fiecare microserviciu este dezvoltat cu Spring Boot, o platformă Java puternică pentru dezvoltarea de aplicații web și API REST. Acesta permite o configurare ușoară și o integrare rapidă cu alte servicii.
- *Bază de date*: PostgreSQL este baza de date utilizată pentru ambele microservicii, oferind capabilități robuste de gestionare a datelor și o compatibilitate ridicată cu Spring Boot.
- *Frontend*: Angular este folosit pentru a crea interfața web. Oferă funcționalități avansate de gestionare a stării și posibilitatea de a crea o aplicație cu o experiență de utilizare modernă și responsivă.
- *Deployment*: Docker este utilizat pentru a containeriza aplicația și pentru a gestiona toate componentele într-un mediu izolat și replicabil. Docker Compose permite orchestrarea serviciilor și a bazelor de date necesare.

3 Arhitectura

Microservicile folosesc arhitectura bazata pe layere.

Arhitectura pe straturi oferă separarea responsabilităților, asigurând o gestionare și testare mai facilă a logicii aplicației. Fiecare layer funcționează independent, conferind astfel flexibilitate și permițând modificări locale fără a afecta alte componente. De asemenea, testabilitatea este îmbunătățită, deoarece fiecare layer poate fi verificat separat, garantând o calitate ridicată a codului. În plus, scalabilitatea sistemului este asigurată prin arhitectura de microservicii, care permite scalarea independentă a fiecărui serviciu pentru a gestiona eficient un volum mare de utilizatori și dispozitive.

3.1 Controller Layer (API)

Layer-ul Controller este responsabil de expunerea API-ului REST și gestionează toate cererile HTTP venite. Acesta primește cererile de la client (frontend-ul Angular sau alte microservicii), validează datele de intrare și apoi invocă metodele corespunzătoare din Service Layer pentru a executa logica de business.

Depinde de Service Layer pentru a obține rezultatele și a efectua operațiunile necesare.

3.2 Service Layer

Service Layer implementează logica de business a aplicației. Acesta este locul în care se definesc operațiunile specifice fiecărui domeniu de activitate. Service-ul realizează calcule, procesează datele și se ocupă de logica ce stă la baza fiecărei operațiuni, asigurând astfel consistența și integritatea datelor între diferite operațiuni.

Depinde de Repository Layer pentru a interacționa cu baza de date și a manipula entitățile persistente.

Uneori, poate comunica cu alte servicii folosind un client REST (RestTemplate sau WebClient), de exemplu, pentru a sincroniza datele utilizatorilor și dispozitivelor între microserviciile User și Device.

3.3 Repository Layer

Repository Layer este responsabil de gestionarea accesului la baza de date și oferă metode CRUD pentru interacțiunea cu entitățile persistente (User și Device). Acest layer folosește JPA (Java Persistence API) pentru a efectua operațiuni de bază pe baza de date, cum ar fi salvarea, găsirea, actualizarea și ștergerea înregistrărilor.

Este dependent de Model Layer (Entity) pentru a cunoaște structura datelor pe care trebuie să le persiste în baza de date.

Este independent de Service și Controller, oferind astfel un nivel de abstractizare care asigură separarea între logica de business și persistența datelor.

3.4 Entity Layer

Model Layer (sau Entity Layer) definește structura datelor utilizate de microservicii. Aici sunt definite entitățile care corespund tabelelor din baza de date. În cazul de față, entitățile principale sunt User și Device, fiecare reprezentând o înregistrare din baza de date.

Folosit de Repository Layer pentru a persista și recupera date.

Utilizat de Service Layer și Controller Layer pentru a reprezenta structurile de date din cererile și răspunsurile API.

3.5

4 Functionalitati principale

4.1 Microserviciul User

- *Managementul utilizatorilor*: Gestionează crearea, ștergerea și actualizarea conturilor de utilizator.
- Autentificare: Include logica de autentificare, precum validarea credențialelor de login.
- *Rol de Administrator*: Metoda initAdmin inițializează un cont de administrator implicit dacă acesta nu există deja.
- *Sincronizare User-Device*: La crearea sau ștergerea unui utilizator, microserviciul Utilizator comunică cu microserviciul Dispozitiv pentru a adăuga sau elimina referințele către utilizatori. Aceasta asigură o consistență a datelor între servicii.
- *Handler pentru API REST*: Clasa UserHandler folosește un RestTemplate pentru a trimite cereri HTTP către microserviciul Dispozitiv, permițând comunicarea între servicii pentru gestionarea datelor utilizatorilor.

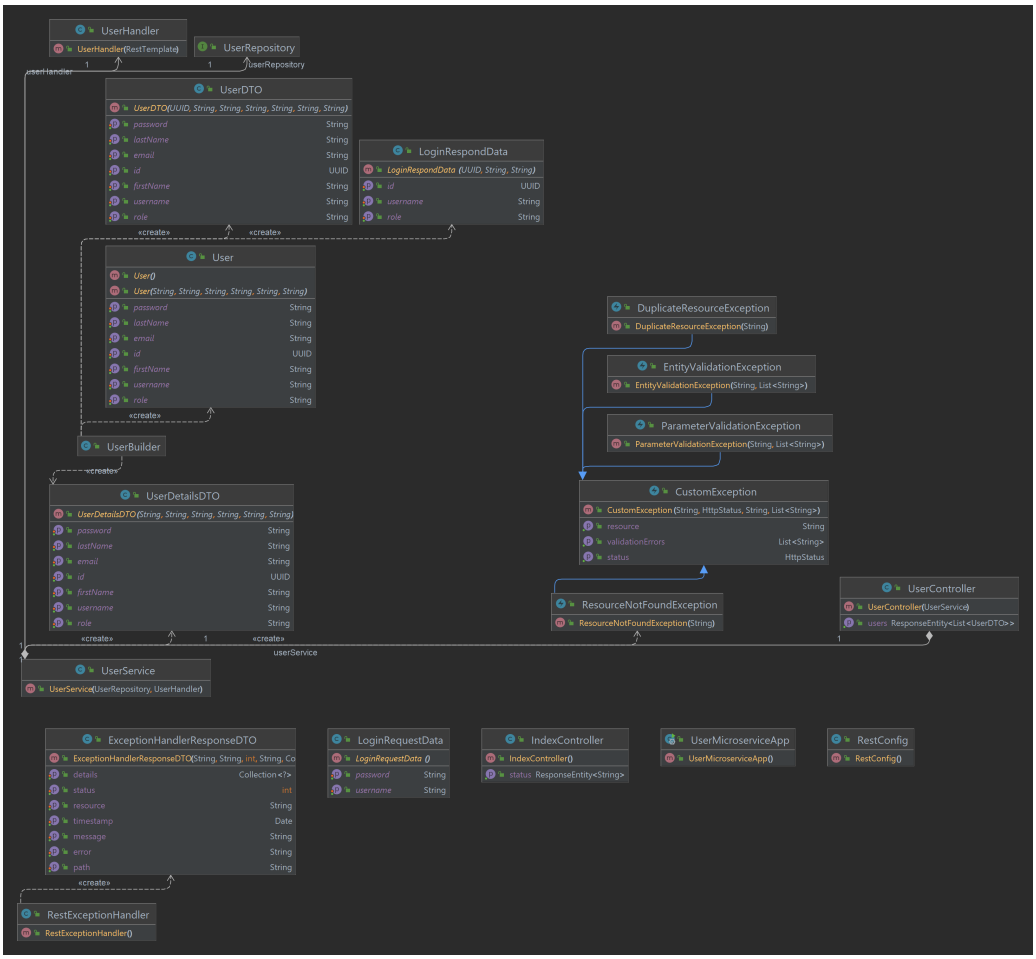

Figura 1: User UML Diagram

4.2 Microserviciul Device

- *Managementul dispozitivelor*: Oferă operațiuni CRUD pentru dispozitive.
- *Asociere cu Utilizatori*: Dispozitivele pot fi asociate cu utilizatori. Dacă un utilizator este șters, dispozitivele devin neatribuite și pot fi realocate altor utilizatori.
- *Sincronizare Dispozitiv-Utilizator*: Dacă un utilizator este șters, microserviciul Dispozitiv elimină asocierea cu acel utilizator pentru a evita referințe orfane.
- *Gestionare Erori*: Utilizează excepții personalizate (ex. ResourceNotFoundException) pentru cazurile în care resursa cerută (utilizator sau dispozitiv) nu este găsită.

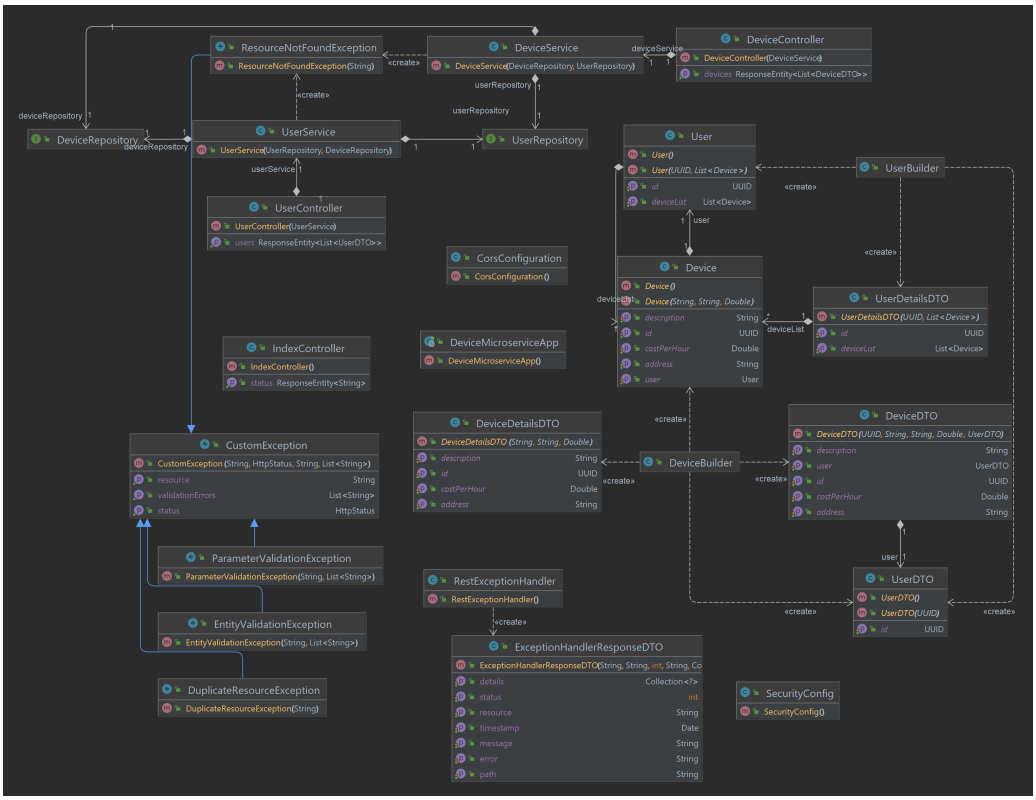

Figura 2: Device UML Diargam

4.3 Microserviciul Monitoring

- *Colectarea Metricilor*: Monitorizează și colectează date despre performanța dispozitivelor și starea utilizatorilor.
- *Alertare*: Trimite notificări sau alerte în cazul în care anumite praguri de performanță sau erori sunt depășite.
- *Interogare și Raportare*: Permite interogarea datelor istorice pentru generarea de rapoarte despre starea dispozitivelor și utilizatorilor.
- *Gestionare Erori*: Utilizează excepții personalizate (ex. DataNotAvailableException) pentru a trata cazurile în care datele nu sunt disponibile sau nu pot fi accesate.

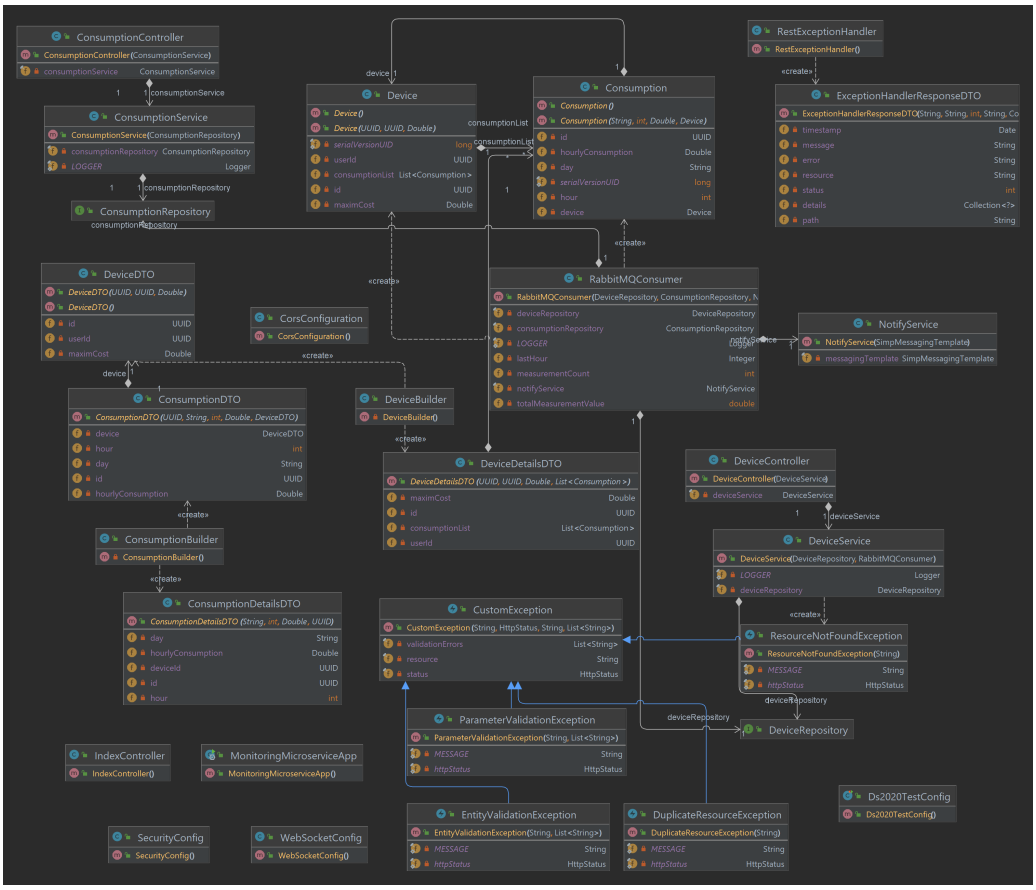

Figura 3: Monitoring UML Diagram

4.4 Microserviciul Chat

- *Trimiterea mesajelor într-o cameră de chat*: Prin metoda sendMessage, mesajele sunt trimise de la un client la un topic specific camerei și sunt retransmise tuturor clienților conectați.
- *Notificarea utilizatorilor care tastează*: Prin metoda typingMessage, atunci când un utilizator începe să tasteze, numele acestuia este trimis către toți ceilalți utilizatori din camera respectivă, notificându-i că un alt utilizator este activ.

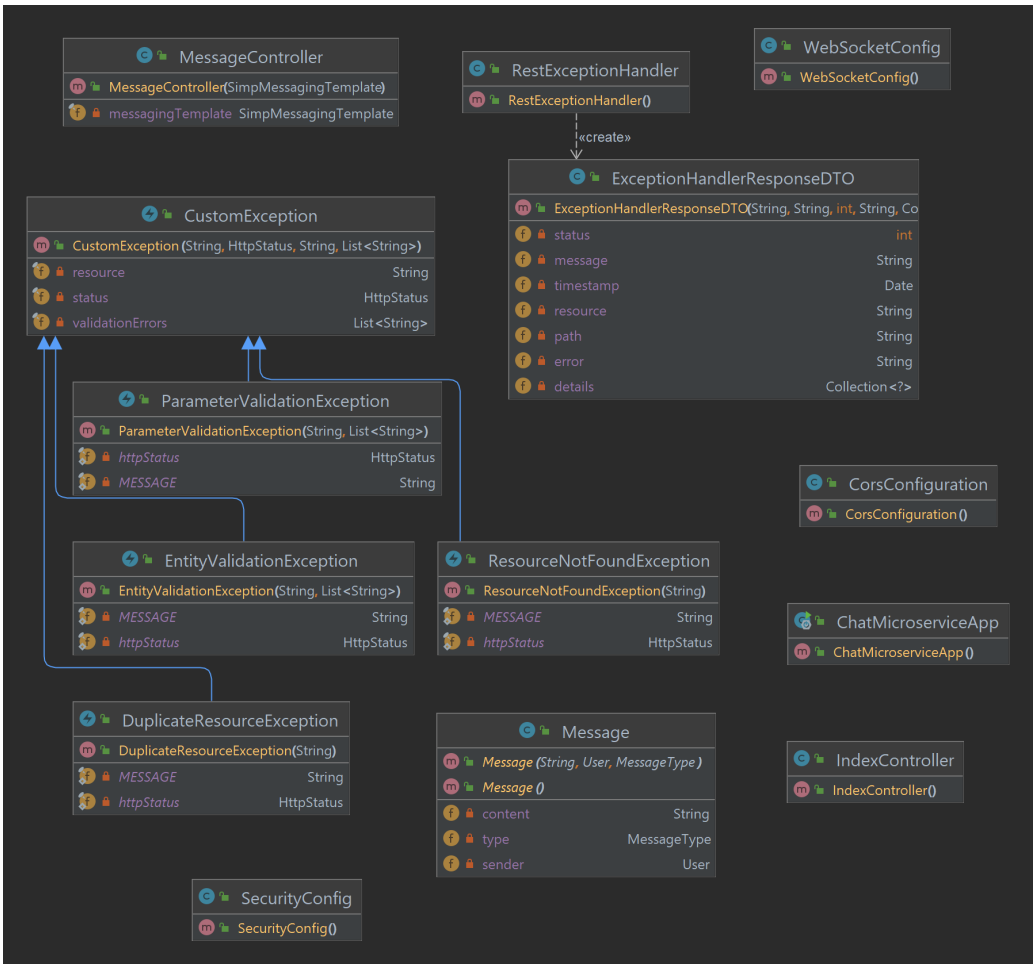

Figura 4: Chat UML Diagram

4.5 Aplicatia Frontend

- *Dashboard Utilizator*: Utilizatorii se pot autentifica și pot vizualiza dispozitivele asociate.
- *Dashboard Administrator*: Administratorii pot gestiona atât utilizatorii, cât și dispozitivele. Pot asocia dispozitive neatribuite utilizatorilor și pot efectua operațiuni CRUD pe înregistrările de utilizatori și dispozitive.
- *Interfață de Autentificare*: Oferă formulare de autentificare și înregistrare pentru gestionarea accesului.
- *Vizualizare Status Dispozitive*: Utilizatorii pot vedea statusurile dispozitivelor asociate în timp real. Datele includ consumul curent și istoric, alături de eventuale alerte legate de depășirea limitelor configurate.

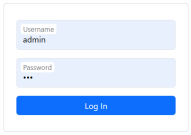

Figura 5: Interfața Login

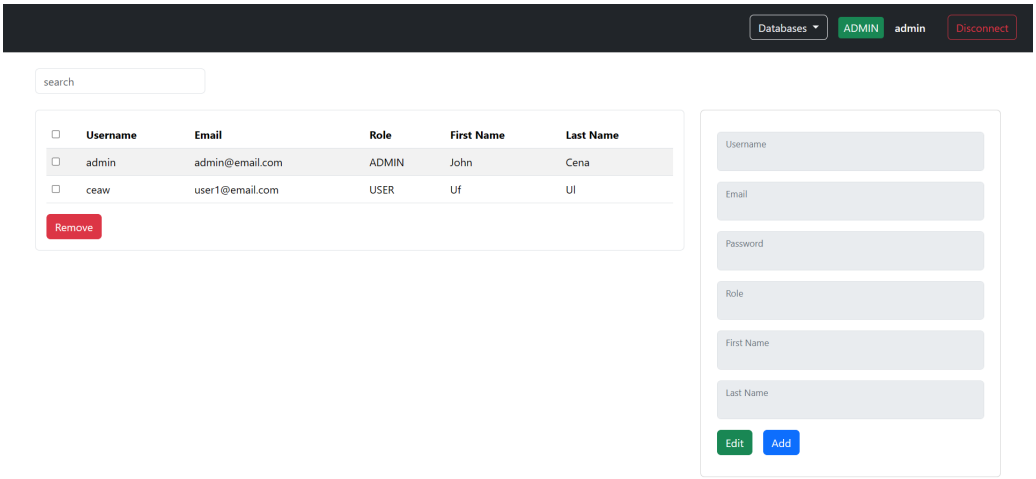

Figura 6: Interfața Utilizator

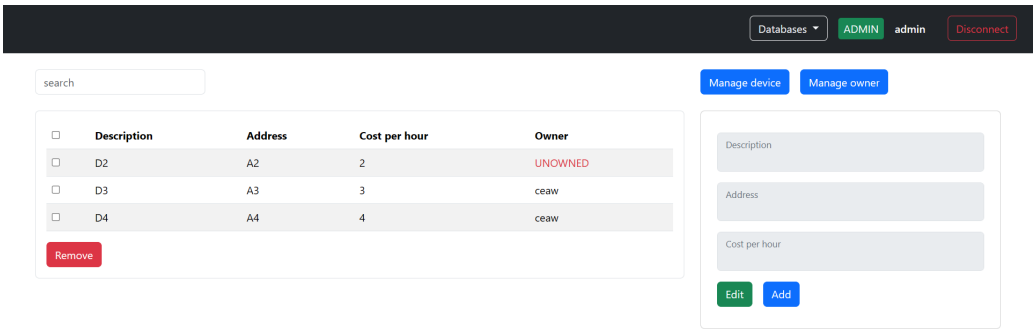

Figura 7: Interfața Dispozitive

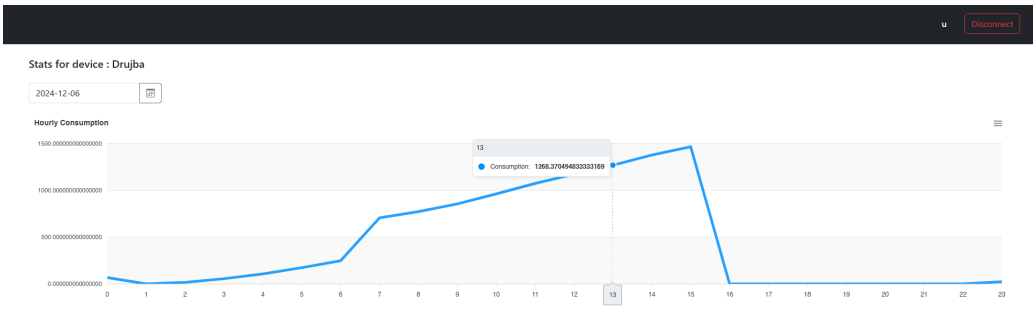

Figura 8: Interfața Vizualizare Status Dispozitive

4.6 Workflow

În arhitectura microserviciului User, interacțiunea dintre componentele principale — controller și servicii — este esențială pentru gestionarea eficientă a operațiunilor legate de utilizatori, cum ar fi adăugarea sau ștergerea acestora. Când o cerere HTTP pentru adăugarea unui utilizator este primită de controller, acesta o trimite către layer-ul de servicii, care implementează logica de afaceri. UserService creează un nou utilizator, folosind UserRepository pentru a salva datele în baza de date și, ulterior, folosește un UserHandler pentru a comunica cu Device Microservice printr-o cerere REST. Această cerere sincronizează datele, adăugând o referință la utilizator în sistemul de dispozitive.

Ștergerea unui utilizator implică un proces similar. Controllerul primește cererea și o transmite către UserService, care, înainte de a elimina utilizatorul, asigură că toate dispozitivele

asociate sunt deconectate. Ulterior, UserService folosește din nou UserHandler pentru a notifica Device Microservice și a elimina referințele utilizatorului din baza de date a dispozitivelor. Această sincronizare bidirecțională între microservicii garantează consistența datelor.

Această separare clară a responsabilităților între controller, servicii și repository permite o gestionare eficientă a operațiunilor, iar comunicarea între microservicii asigură că sistemul rămâne sincronizat. Logica de afaceri din layer-ul de servicii este bine izolată, facilitând testarea și adaptarea componentelor fără a afecta alte părți ale aplicației.

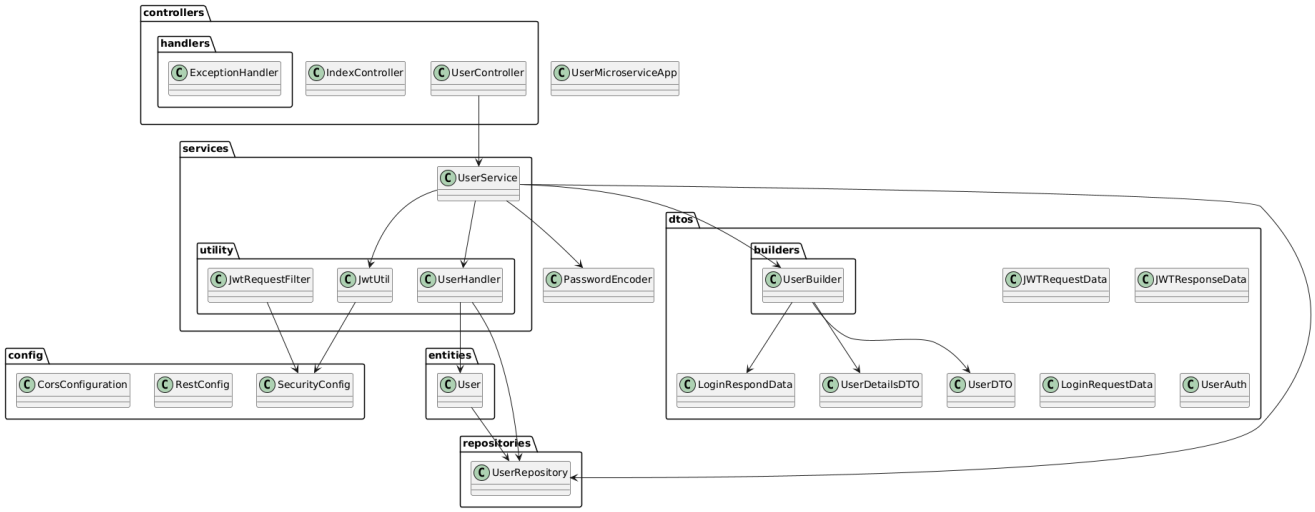

Figura 9: Conceptual User

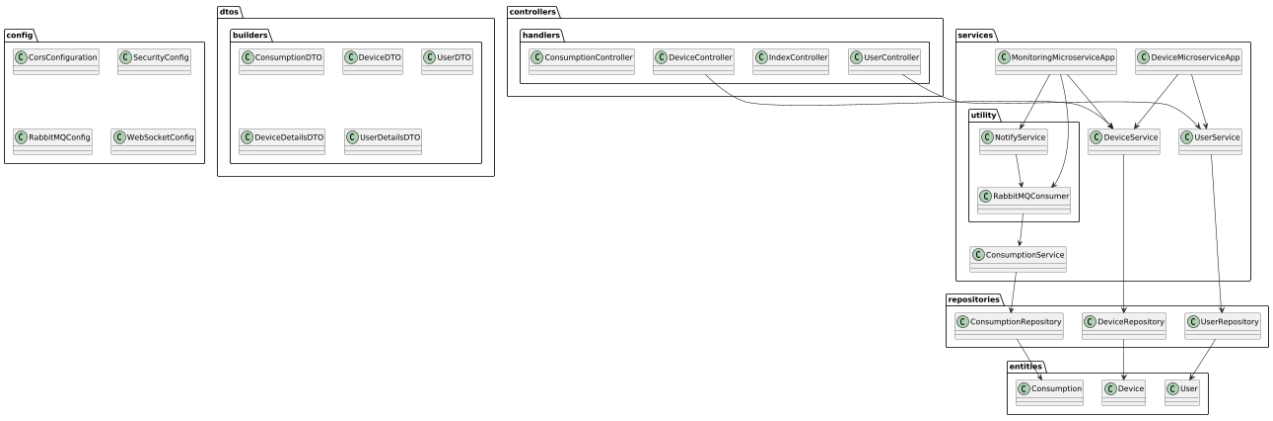

Figura 10: Conceptual Monitoring

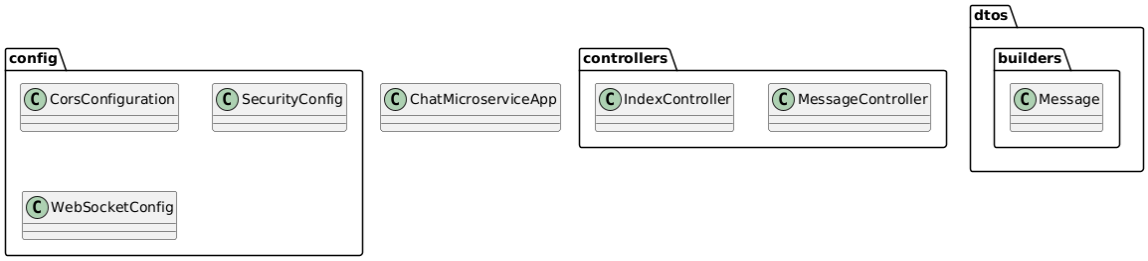

Figura 11: Conceptual Chat

5 Docker

Arhitectura bazată pe microservicii permite ca fiecare componentă să fie containerizată și gestionată independent. Docker Compose este utilizat pentru a defini și configura toate serviciile și pentru a permite o comunicare ușoară între acestea.

Configurare Servicii Docker-Compose:

Fișierul *`docker-compose.yaml`* definește configurația pentru implementarea întregului sistem, incluzând cele două microservicii, bazele de date PostgreSQL și aplicația frontend Angular.

- *Baze de Date PostgreSQL*: Fiecare microserviciu are un container dedicat PostgreSQL (`postgres-user` pentru microserviciul Utilizator și `postgres-device` pentru microserviciul Dispozitiv). Configurația bazei de date este gestionată prin variabile de mediu.
- *Microserviciul Utilizator* (`user-ms`): Este construit din directorul UserMicroservice, depinde de baza de date postgres-user și comunică cu microserviciul Dispozitiv pentru a gestiona relațiile utilizator-dispozitiv.
- *Microserviciul Dispozitiv* (`device-ms`): Este construit din directorul DeviceMicroservice, depinde de baza de date postgres-device.
- *Microserviciul Chat* (`chat-ms`): Este construit din directorul ChatMicroservice, depinde de baza de date user-ms.
- *Aplicația Frontend* (`web-app`): Este construită din directorul `ds-frontend` și asigură interfața pentru utilizatori. Aplicația frontend depinde de ambele microservicii pentru a se asigura că toate back-end-urile sunt funcționale înainte de a începe frontend-ul.

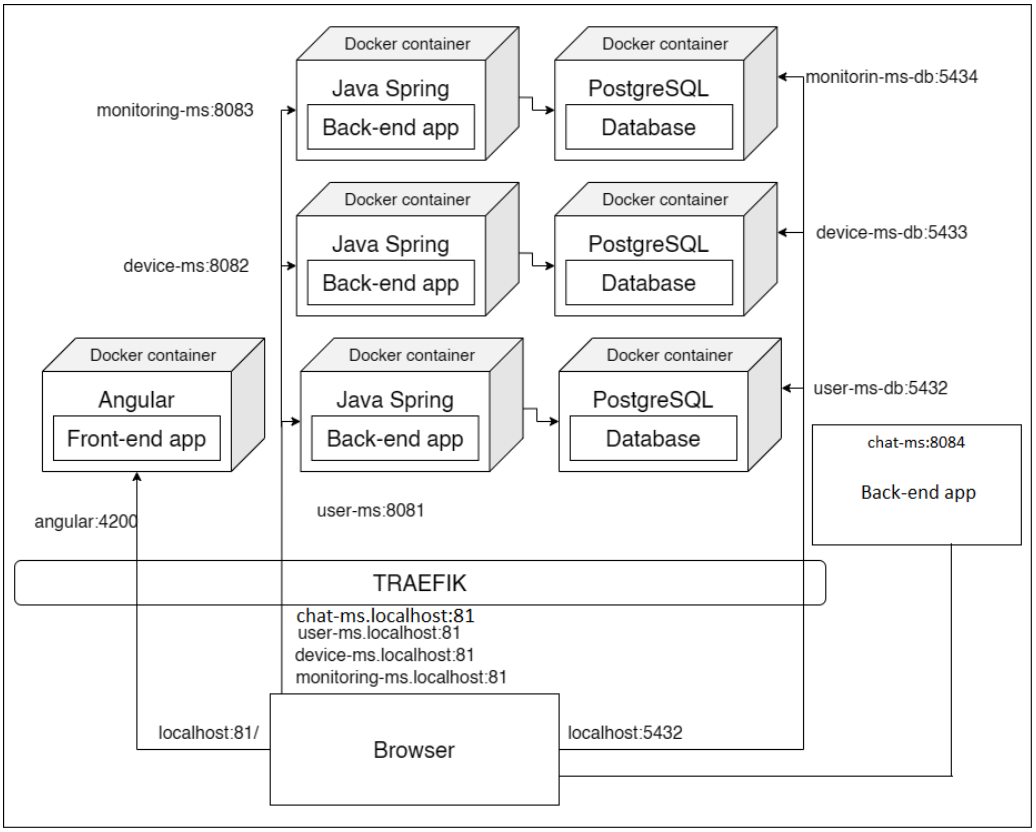

Figura 12: Docker

5.1 Rețea și Volume

- *Rețea*: Toate serviciile sunt parte dintr-o rețea Docker comună denumită net, permițându-le să comunice între ele prin numele serviciilor (de exemplu, postgres-user sau device-ms).
- *Volume*: Se definesc volume persistente pentru bazele de date (`postgres_data_user` și `postgres_data_device`), asigurând persistența datelor chiar dacă containerele sunt reconstruite.

6 Reverse Proxy

- *Traefik Reverse Proxy*: Traefik este utilizat pentru a gestiona rutarea traficului între serviciile containerizate. Configurația definește rutele pentru microservicii pe baza regulilor Host (de exemplu, `user-ms.localhost` pentru Microserviciul Utilizator).
- *Dashboard Traefik*: Un dashboard de monitorizare este accesibil prin `http://traefik.localhost:8081`, permițând vizualizarea metricilor și a statusului serviciilor.
- *Integrare cu Docker*: Traefik utilizează Docker ca provider, ceea ce permite configurarea automată a rutelor pe baza etichetelor (*labels*) adăugate fiecărui container.
- *EntryPoints și Load Balancing*: EntryPoints sunt definite pentru trafic web (`80`) și acces dashboard (`8081`). Traefik gestionează și balansarea de încărcare între replici multiple ale microserviciilor.

7 Biografie

- Learn Spring Boot
- Docker Tutorial 101
- Angular Docs