Sensor Monitoring System and Real-Time Notification

Asynchronous Communication

Assignment 2

Pop Ruxandra Maria

Grupa 30643

Curpins

1. Arhitectura conceptuală a platformei online 3
2. Arhitectura conceptuală a sistemului distribuit 5
3. Design-ul bazei de date 7
4. Diagrama UML de Deploy 9
5. Fisierul READme - considerări de build și execuție 11

1. Arhitectura conceptuală a platformei online

Aplicația de față respectă arhitectura client-server după cum se poate ovserva.
Deci există 2 aplicații, una ce se ocupă de partea de server (backend) și una care se ocupă de partea de client (frontend).

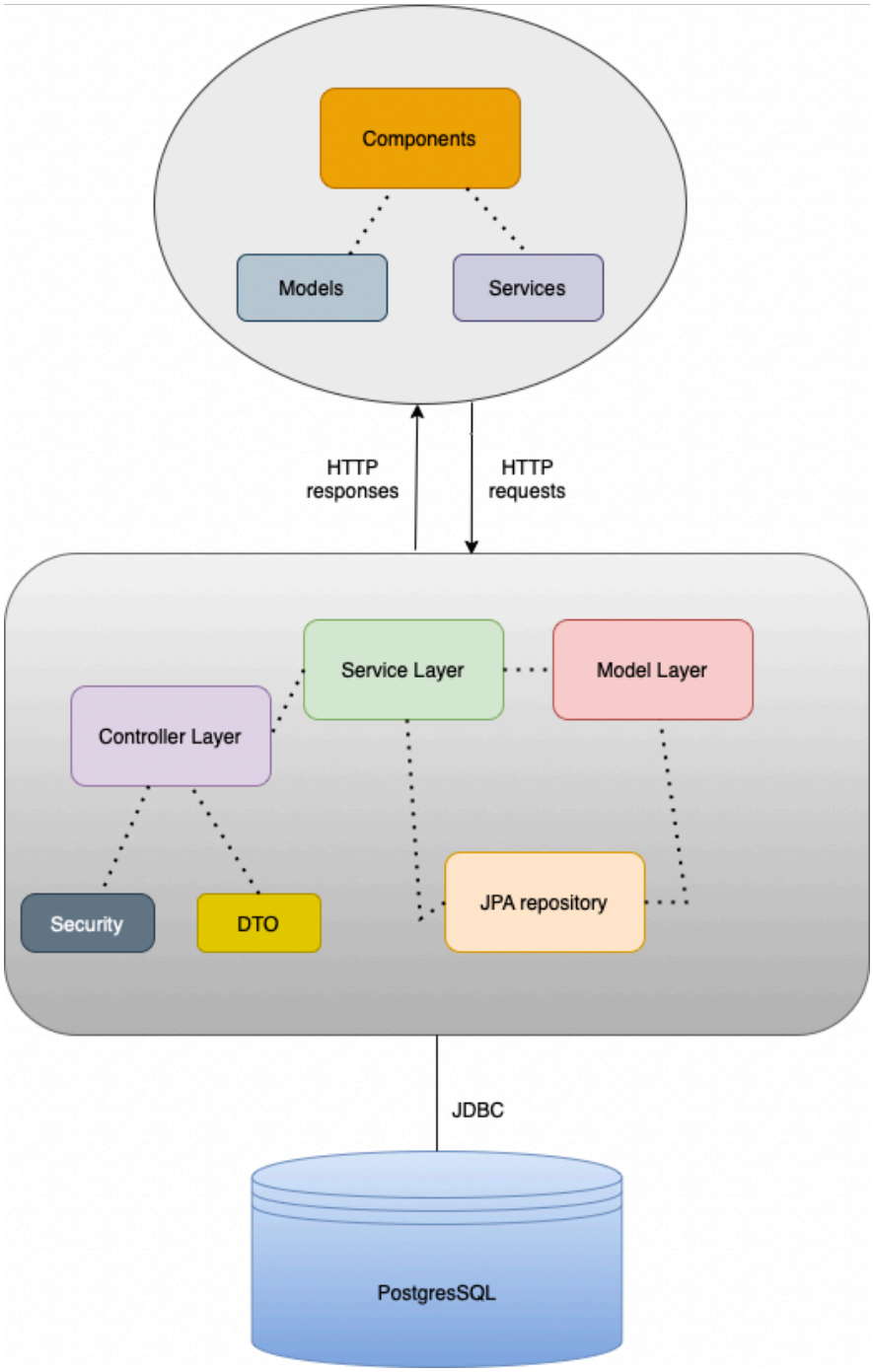

Fig 1.1 Arhitectura conceptuală a platformei online

Comunicarea între client și server se face prin intermediul protocolului HTTP, prin request-uri și responses, design-ul aplicației respectând principiul de transmitere a reqest-urilor http de către client și primirea acestora de către server.

- Înterfața cu utilizatorul a fost creată în Angular (care este un framework structurall TypeScript pentru aplicațiile web prezente pe o singură pagina dinamică) și prezintă următoarele componente:

  1. **Components**: sunt colecți logice de template-uri HTML, siluri asociate și modele care ar putea să opereze intr-un tempate folosind imbinarile /legaturile Angular.

  2. **Models**: care reprezintă mappari-le obiectelor din backend

  3. **Services**: sunt atașate componentelor și au ca rol comunicarea cu server-ul (backend-ul). Aici se crează request-uri HTTP cu headere și parametri corespunzători pentru fiecare metodă și se așteapă un răspuns de la server.

- Pentru partea de backend am folosit Java Spring impreună cu JWT Token pentru securitate

  1. **Security**: a fost realizată cu principiul JWT Token, astfel este nevoie de autentificare pentru a accesa endpoint-urile. După ce te loghezi se generează un JWT token care este stocat într-un Cookie pe masină internă a utilizatorului și este folosit pentru a apela metodele corespunzătoare rolului utilizatorului. Când se primeste un request acesta este filtrat, iar un răspuns la acest request se va da doar după ce se va verifica existența token-ului în header-ul requestului, și apoi se validează, și anume se verifică dacă utilizatorul are dreptul de a accesa aceea metodă cerută.

  2. **Controller**: de obicei se realizează un controller pentru fiecare model prezent în baza de date. Acestea conțin service care vor preluca datele primite de pe frontend, prin apelarea de metode de tip Repositories. Aici întâlnim toate endpoint-urile pentru operațiile CRUD la nivelul informațiilor despre modele. Pe scurt rolul sau este de a primi request-uri și de a trimite mai departe un răspuns la client în funcție de request-ul primit, făcând operațiile necesare.

  3. **Service**: prelucrează datele primite de la controler și transmite la client, noile obiecte pe care s-au făcut operațiile cerute.

  4.**Repositories**: reprezintă înterfețele unde sunt definite metodele care au acces la baza de date. Aici am creat diferite query pentru a prelucra obiectele din baza de date.

  5. **Models**: sunt clase care reprezintă obiectele cu care aplicația va lucra. Fiecare tabel din baza de date, va avea asociată o clasă de timp model cu atributele care reprezintă de fapt coloanele din tabel. Avem 3 modele : User, Device, Measurement.

- Pentru baza de date am folosit **PostgreSQL**, comunicarea cu aceasta realizânduse prin interfețele Repositories, care cum am specificat mai sus conțin metodele CRUD necesare tabelelor.

2. Arhitectura conceptuală a sistemului distribuit

Arhitectura este formată din: **Message Producer** și **Message Consumer, Message Broker,** respectiv **WebSocket**

1. **Message Producer -** simulează un senzor care citeste consumul de energie,a unui dispozitiv a cărui id este dat ca și argument, dintr-un fișier **sensor.csv.** Valorile din fișier sunt transmise din 10 în 10 minute către Message Broker, acesta reprezentând coada, **RabbitMQ.**
2. **Message Broker** - este reprezentat de coada **Rabbit Mq,** unde sunt stocate pentru o perioadă determinată de timp valorile citite din fisier. Aceste valori sunt stocate sub forma unui obiect JSON de tipul MeasurementDTO și anume:

{ “deviceID”: 1,
“consumation”:33.44,
“timestamp“: “2022-12-02 12:25:54
}

   - **deviceID:** este unic și corespunde dispotivului din baza de date pentru care se doreste a citi informații de pe senzor.
   - **consumation:** reprezinta valoarea de energie kWh inregistrata de sensor la un moment de timp, fiind valoarea citita din fisierul **sensor.csv .**
   - **timestamp:** este timpul la care a fost inregistrată valoarea din fisier, timpul local.

Message Producer și Broker sunt cuprinse în aplicația desktop care va fi rulată din terminal de catre utilizator atunci când doreste să pornească senzorul.
Aplicația Desktop este formată din 3 clase plus clasa standard Main.

- **MeasurementDTO** - care reprezintă tipul obiectului care va fi inregistrat în coadă și pe urmă trimis spre backend.
- **ScannerSensor -** în această clasă sunt citite pe rând datele din fișier, după care se crează obiecte de tip measurementDTO.
- **Send** - aici se realizează conexiunea la **RabbitMQ** și respectiv se trimit datele citite din fisier prin intermediul clasei **ScannerSensor.**

3. Message Consumer - este cuprins în aplicația de backend și are rolul de a prelua și preprocesa datele primite de la **Message Broker**. Datele sunt preluate și stocate în baza de date, în tabelul Measurement. Totodată tot aici se calculează consumul total de energie pe oră, iar dacă acesta depăseste valoarea maximă a dispozitivului înregistrat în baza de date se anunță utilizorul in mod asincron pe interfața sa web. Trimiterea de notificari spre interfața web se face prin intermediul Web Socket-urilor, astfel dacă utilizatorul este conectat el va primi un warning cum că valoarea maxima totală pe ora a fost depasită. Această transmisie este una asincronă altfel notificarea va fi vizibilă în interfață doar daca utilizatorul este logat. Chiar dacă sa inregistrat o notificare utilizatorul nu a fost logat acesta va putea vedea ulterior notificarea doarece am ales să salvez toate notificarile în baza de date.

Message Consumer-ul cuprinde următoarele clase:

- **Receiver -** unde sunt recepționate datele de la coadă prin intermediul lui RabbitListener. Aceste data sunt transformate din GSON object in obiecte de tip MeasurementDTO,

după care sunt salvate în baza de date, iar în cazul unui consum de enegie se creaza o notificare care este transmisă spre frontend.

- **WebSocketConfig** - se fac configurarile necesare pentru a transmite mesaje spre frontend.

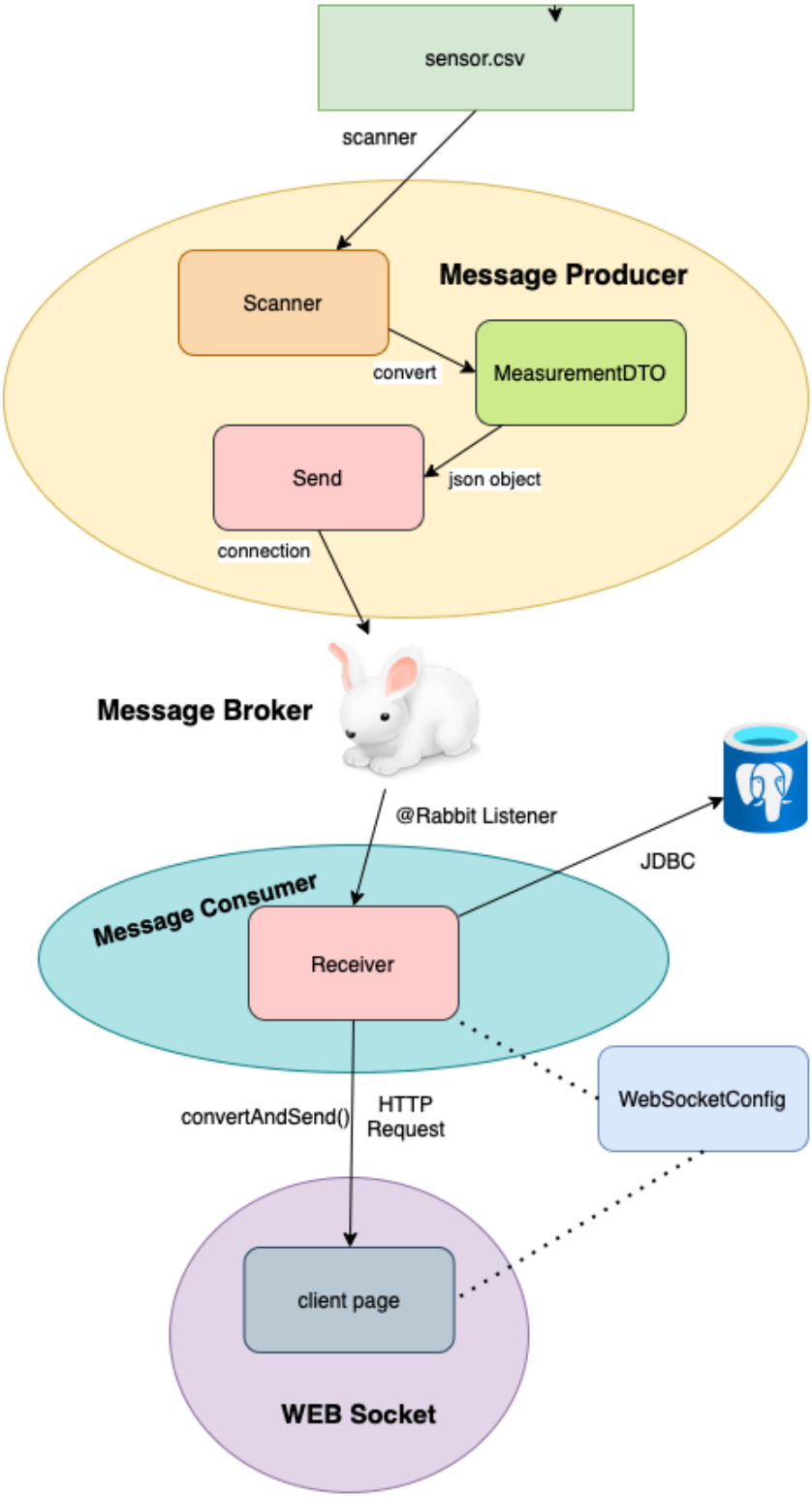

Fig 1.2 Arhitectura conceptuală a sistemului distribuit

3. Design-ul bazei de date

Baza de date folosită prezintă 3 tabele după cum se poate observa și in figura 1.2. Aceasta reprezintă persistența modelelor folosite în aplicația server.

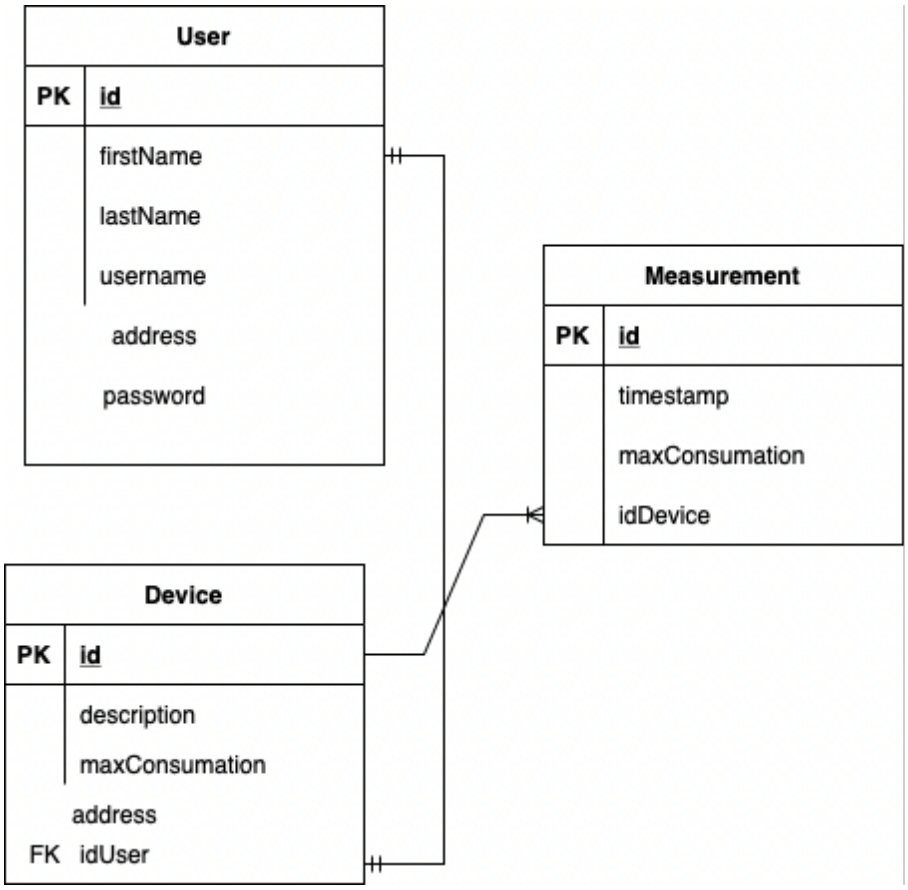

Figura 1.3 . Diagrama bazei de date

- Tabele **User**: conține detaliile despre utilizatorii aplicației și anume: prenume, nume, username, adresă, parolă și rolul. Prezintă o relație de **OneToOne** cu tabela **Device** astfel un user poate să aibe un singur device pe care îl foloseste la un moment dat.

- Tabela **Measurement**: conține informați despre măsuratorile efectuate de un device la un anumit timp. Prezintă următoarele coloane: timestamp, consumul de energie maxim per ora și este în relație de **ManyToOne** cu tabela **Device**.

- Tabele **Device**: conține informații despre dispozitivele aplicației, prezentând următoarele coloane: descriere, adressa, consumul de energie maxim per ora și idUser-ului care îl deține. Prezintă o relație de **OneToMany** cu tabela **Measurement** deoarece un device poate avea mai multe măsuratori ai consumului, la ore diferite, în funcție de orele la care utlizatorul îl foloseste.

Rolul utilizatorului reprezintă capacitatea acestuia de a accesa anumite pagini web ale aplicației sau nu. Utilizatori sunt de 2 tipuri: **admin** și **user**, fiecare dispunând de funcționalități diferite deci de metode diferite în controller care vor fi restricționate de partea de securitate în funcție de tip.
Când se mapează coloana device cu coloana user, se inițializeazp pentru device un measurement, iar la fiecare ora se înițializeazp un nou measuremet în funcție de consumul folosit la acea ora.
Consumul este generat random în funcție de valoarea maximă de consum a device-ului.

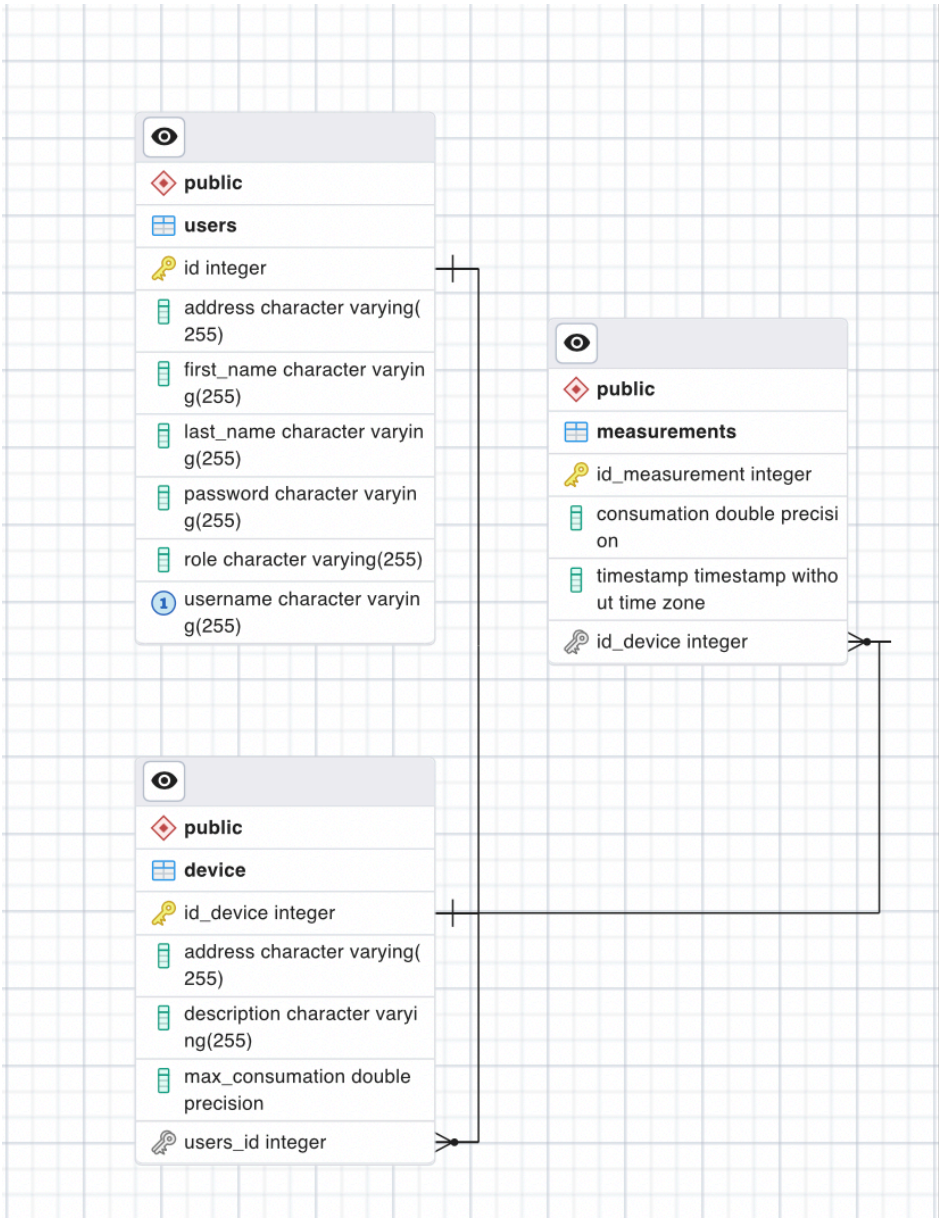

Figura 1.4. Diagrama bazei de date generate de PostgreSQL

4. Diagrama UML de Deploy

Cele 2 aplicații (backend și frontend) au fost deployate local ca și containere docker. Pentru partea de **client** sa folosit portul **4040**, pentru partea de **server 8080** iar pentru **baza de date** portul **5432.** Diagrama UML de deploy prezintă 4 noduri, care sunt containerele în docker.

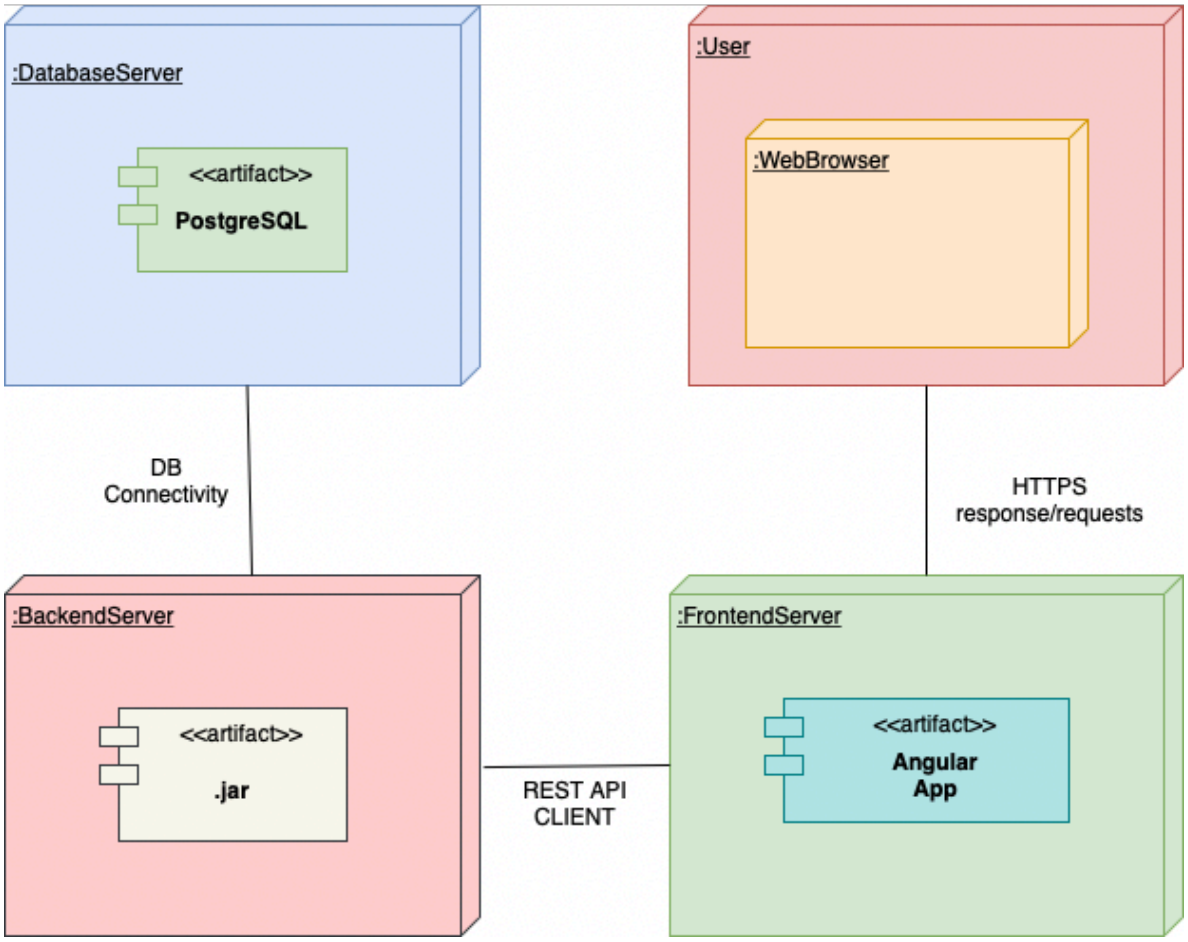

Figura 1.5 Diagrama UML de deploy

- **DatabaseServer**: este baza de date a aplicației implementată în PostgreSQL.
- **BackendServer** : este patea de server a aplicației. Este un API ce conține endpont-urile securizate în funcție de rolul utilizatorului. Conexiunea cu baza de date s-a realizat prin intermediul aplicației JAVA prin includerea detaliilor bazei de date în fisierul .properties al aplicației.

- **FrontendServer** : reprezintă partea de client, realizată în Angular, și anume reprezintă ce va vedea utilizatorul atunci când deschide browser-ul WEB . Aici am implementat auth guad pentru a restricționa accesul neautorizat a unui user pe o anumită pagina, aftel când un utilizatorul încearcă să acceseze o pagină pentru care nu are autorizație va fi redirecționat pe pagina de home al aplicației. Conexiunea cu serverul se face prin crearea unui request și trimiterea acestuia pe partea de server prin api pentru a primi un response.

- **Web Browser**: reprezintă browser-ul utilizatorului de unde se accesează aplicația Angular.

  Conexiunea cu celelalte containere se face prin intermediul TCP/IP

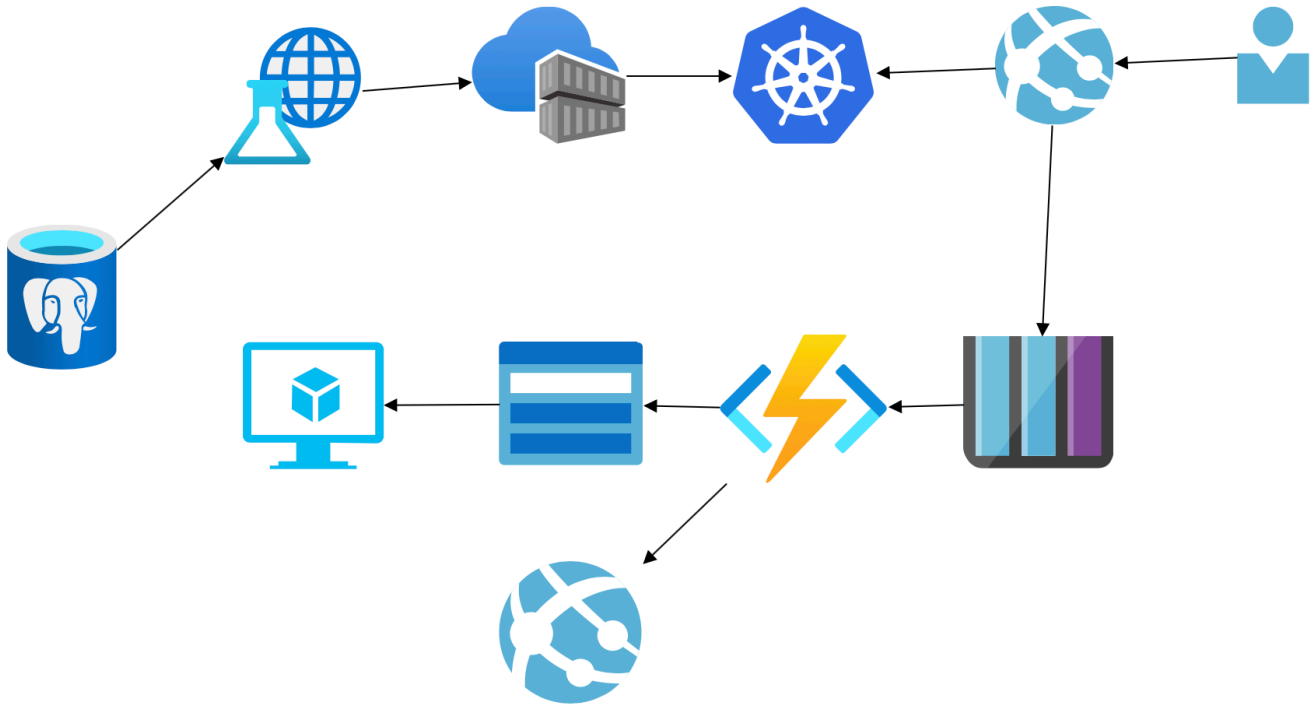

Figura 1.6 Diagrama UML de deploy azure

5. Fisierul READme - considerări de build și execuție

1. Frontend - Aplicația client Angular

- **Versiuni utilizate**

  - Node Js version: **v18.12.0**
  - Npm version: **v8.19.2**
  - NGINX

- **Pași**

  - Se clonează aplicația (repository-ul) de pe git.
  - Se deschide un terminal și se execută următoarea comandă pentru a instala pachetul node_modules: **npm install** sau dacă nu funcționează îl forțam **npm install —force .**
  - După care pornim aplicația utilizând comanda: **npm start.**
  - Iar pe urmă o accesam din broswer: **localhost:4200.**
  - În clasa websocket.service.ts redenumești topic-ul în funcție de topicul scris în backend.
  - WebSocketEndPoint trebuie sa corespundă cu endPoint-ul din backend.

2. Bakcend - Aplicația Java Spring Boot

- **Versiuni utilizate**

  - Java version**: 18.0.2.1**

- **Pași**

  - Se clonează aplicația de pe git.
  - Se clonează aplicația (repository-ul) de pe git
  - Se deschide un terminal și se execută următoarea comandă pentru a instala pachetul
  - Setarea variabilelor de conexiune la baza de date în fisierul application.properties, în cazul meu baza de date este denumită ca fiind **db**
  - **URL**= jdbc:postgresql://localhost:5432/db **Username**=popruxi
    Password=ruxi
    Port=5432
  - Setarea proprietăți hibernate: **spring.jpa.hibernate.ddl-auto=update**

- În fisierul application.properties adăugi configurarile pentru RabbitMQ în funcție de datele voastre. În cazul de față trebuie să introduceți:
  - **spring.rabbitmq.host**=beaver.rmq.cloudamqp.com
  - **spring.rabbitmq.port**=5672
  - **spring.rabbitmq.username**=sxuozhsx
  - **spring.rabbitmq.password**=wM5Um7W2ww5VnK6uvXTVV0l4YsHjuzMT
  - **spring.rabbitmq.virtual-host**=sxuozhsx
  - **queue.name**=sd
- În clasa Receiver, la metoda convertAndSend introduceți topicul dorit care trebuie să fie la fel ca cel din frontend pentru a putea fi transmise datele prin webSocket.
- Asigurați-vă din nou ca endpoint-ul setat in clasa WebSocketConfig este acelasi cu cel din frontend.
- De asemenea, trebuie să setați permisiunea de a accesa aceste api-uri de către tipul de utilizator dorit, în fișierul WebSecurityConfig, în metoda filterChain.
- Rulează aplicația

3. Docker

- **Pași**

- Se instalează aplicația docker desktop

- Atât pentru client cât și pentru server se crează un fisier **Dockerfile** și unul **docker-compose.yml**

- În interiorul acestor fisiere se întroduc valorile din fisierele prezente pe git + se face configurarea la baza de date în fisierul compose în funcție de baza de date folosită de tine (password , username, port, etc.)

- După care atât pentru server cât și pentru client se rulează în terminal comanda **docker-compose up** , care va crea containere pentru fiecare componente in docker desktop.

- Containerele se află acuma în modul running, iar backend-ul poate fi accesat la portul **localhost:8080,** frontendul prin **localhost:4040**

- Baza de date nou creată va fi goală, astfel trebuie să adăugăm prin intermediul pgadmin- ului un utilizatorul de tip admin, iar apoi prin intermediul interfației ne logăm pe contul de admin și începem să adăugam date în bază. PgAdmin poate fi accesat la portul **localhost:8180** pentru a adăuga adminul. Atenție la parola, aceasta trebuie să fie una criptată pentru a corespunde cu securitatea implementată în java spring.