UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Energy Utility Platform

Proiect la disciplina Sisteme Informatice Distribuite
An universitar 2022-2023

Student:
Bîrluțiu Claudiu-Andrei
Tehnologia Informației
Grupa 30643

An universitar 2022/2023

Cuprins

1. Enunțul problemei

Problema se referă la dezvoltarea unei aplicații web (client-server) ce are ca temă monitorizarea consumului de energie cu ajutorul unor aparate de măsurat. Aplicația este role based, permițând în mod restricționat și filtrat execuția funcționalităților pe baza rolurilor asignate userilor ce se conectează. Există 2 tipuri de utilizatori: cei cu rol de admin și clienții obișnuți (în contextul aplicației de față, consumatorii).

Criza energetică devine din ce în ce mai accentuată, iar principalul factor îl prezintă consumul mare de energie electrică făcut în mod excesiv, fie din neglijența oamenilor, fie de folosirea unor aparate electrice vechi și consumatoare. Existența unei aplicații care să monitorizeze consumul de energie electrică prin intermediul unor device-uri specializate și care oferă rapoarte zilnice cu consumul de energie pe oră, ar oferi consumatorilor o nouă perspectivă prin care să-și poată eficientiza consumul de energie.

Scopul principal al aplicației este cel de proiectare și implementare a unei aplicații web care să permită unor utilizatori de tip admin să insereze noi metering devices și utilizatori, precum și partea de asignare a dispozitivelor userilor. Utilizatorii de tip client vor avea posibilitatea de vizualizare a dispozitivelor asignate și vor putea accesa rapoarte zilnice ale consumului pe bază de grafice.

2. Instrumente utilizate

În faza de proiectare a aplicației s-a folosit instrumentul ***StarUML***, disponibil la adresa https://staruml.io/. Acesta oferă posibilitatea proiectării aplicației prin sintetizarea cazurilor de utilizare prin diagrame *Use Case* și prin definirea arhitecturii acesteia cu ajutorul unor *Diagrame UML de clase sau pachete și* crearera de diagrame ERD în care se observă relaționarea entităților aplicației server.

Pentru faza de implementare a backend-ului s-a folosit mediul de dezvoltare *IntelliJ IDEA 2022.2.3 Ultimate Edition (JetBrains)* care oferă o serie de funcționalități cum ar fi versionarea aplicației cu GIT, formatarea codului și de asemnea un sistem de sugestii și autofill de acuratețe ridicată. Proiectul creat pentru aplicație în Intelliji este de tip Spring Boot, având dependințele definite în fișierul *pom.xml.*

Pentru partea de aplicație client s-a folosit *ReactJs*, iar mediul de dezvoltare VSCode. Pentru versionarea aplicației la nivel local s-a folosit GIT, iar repository-ul remote a fost creat pe Github în organizația *DS202230643BirlutiuClaudiuAndrei*.

De asemenea, pentru crearea bazei de date s-a folosit **postgresSql** cu environment-ul **Pgadmin4**. Pentru migrarea bazei de date s-a folosit **Flyway.** A fost creat un script de introducere a unui user de tip admin, un user de tip client și 2 device-uri. De asemenea, s-au adăugat măsurători pentru un device al clientului pentru a forma datele becesare generării de raport orar pe grafic.

Pentru containerizarea celor două aplicații, backend și frontend, s-a folosit Docker. Containerele docker create au fost urcate într-un registry Azure. Cu azure devops s-a definit

pipeline-ul pentru continous integration, iar apoi s-a configurat pipeline-urile pentru continous development. Pentru rularea job-urilor, respectiv task-urilor s-a folosit un agent local care se ocupă de crearea noilor imagini docker și crearea printr-un script bash a docker context-ului legat de azure care sa facă push imaginii noi create pe azure și să pornească un container din imagine.

3. Limbajele de programare

Pentru dezvoltarea aplicației backend s-a optat pentru limbajul de programare JAVA (11). Cu acest limbaj se poate lucra foarte eficient cu conceptul de obiect. Pentru dezvoltatorul aplicației a reprezentat o noua posibilitate de familiarizare cu acest limbaj și principille SOLID și de a exersa tehnicile de clean coding. Un alt motiv pentru care s-a ales acest limbaj este faptul că împrumută o mare parte din sintaxa C și C++, dar are un model al obiectelor mai simplu. Framework-ul pentru server-ul de backend este Spring Boot, iar dependențele au fost definite în fișierul pom.xml. S-au definit mai multe fișire de tipul application.properties pentru profilele diferite ale aplicației. Pentru production se ia fișierul implicit, pe când pentru modul development există fișierul application-dev.properties.

Pentru aplicația de frontend s-a ales *ReactJs*, o bibliotecă JavaScript care permite dezvoltarea interfețelor client, pemițând definrea de componente care să fie randate pe paginile utilizatorilor. S-a folosit protocolul de comunicare HTTP între cele cele 2 aplicații client-server, aplicația de backend jucând rolul unui Rest Service. Stilizarea componentelor HTML s-a realizat prin script-uri css.

4. Arhitectura conceptuală a aplicației

Aplicația de față a fost structurată după modelul arhitectural **client-server.** Astfel, vor exista două aplicații, una ce se ocupă de partea de server, iar cealaltă reprezintă partea de client. Arhitectura client / server este un model de calcul în care serverul găzduiește, livrează și gestionează majoritatea resurselor și serviciilor care urmează să fie consumate de client.

Comunicarea între client și server se va face prin intermediul protocolului **http**, iar designul structural al celor două aplicații este construit ținând cont de transmiterea request-urilor http și formarea răspunsurilor de către server. Astfel, ambele aplicații sunt structurate pe mai multe layere ce au rolul de abstractizare și grupare a funcționalităților comune. Acest model structural bazat pe mai multe layere oferă posibilitatea reutilizării codului și de asemnea crește ușurința de scalabilitate. Print-o astfel de structură, cum e prezentată în figura 1, se pot respecta principille SOLID. Fiecare layer are o responsabilitate clară, iar comunicarea între layere se face prin dependcy injection.

Aplicația Client- React Js:

- **pages**: în componenta **app.js** sunt mapate link-urile la componenetele react ce trebuie randate pentru paginile respective; toate componentele js definite vor fi folosite pentru creare unor pagini html ce vor fi accesibile la un anumit URL
- **componente**: deoarece s-a lucrat cu react hooks, componentele definite sunt funcții ce au rolul de a randa bucăți de cod html în vederea realizării paginilor utilizator.

Aceste componente au responsabilităti diferite, fie de la form-uri de editare şi adăugare utilizatori sau device-uri, fie tabele sau carduri în care e inclusă informaţia primită de la server

- **services**: acestea sunt ataşate componentelor descrise mai sus şi se ocupă de comunicarea cu partea de server. În cadrul acestora se creează request-urile http cuc headerele şi parametri coresunzători şi se aşteaptă un răspuns de la server

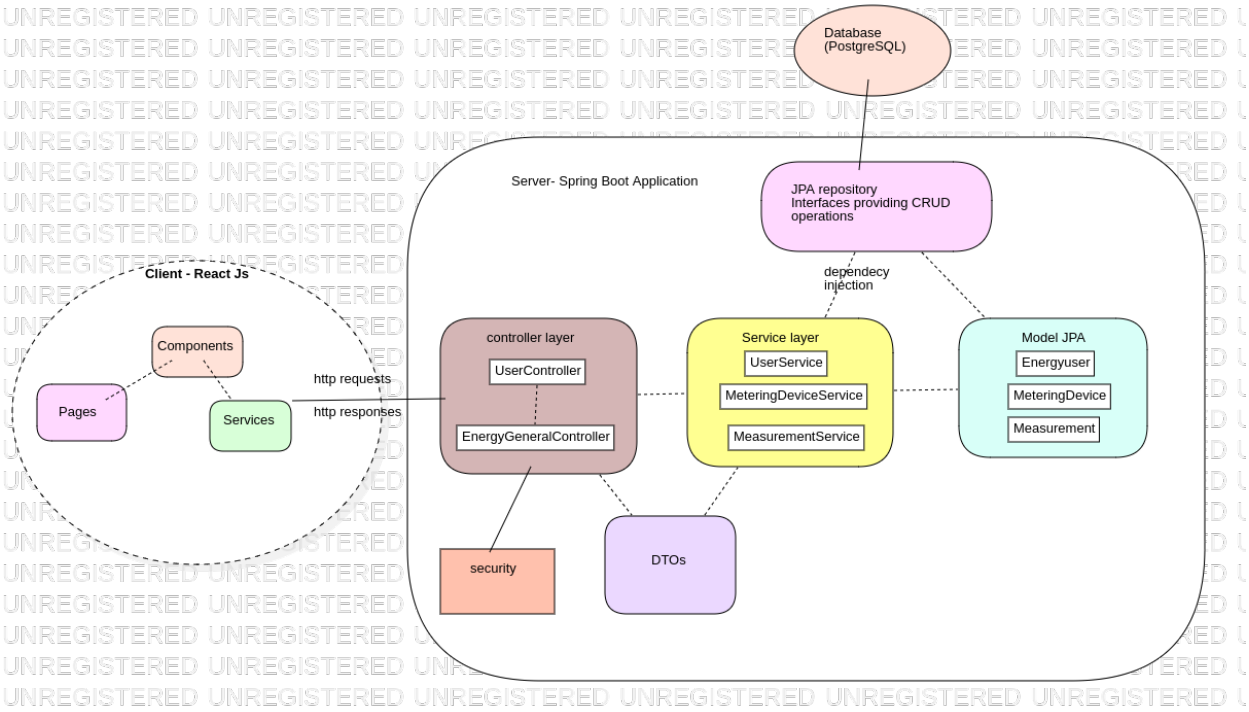

Figure 1: Arhitectura Conceptuală

Aplicaţia Server – Spring Boot

- **security**: în momentul în care se primeşte un request http, acesta este filtrat de partea de securitate a aplicaţiei; Se va verifica de obicei existenţa token-ului în header-ul request-ului, iar apoi validitatea acestuia.
- **controller**: vor exista mai multe controllere în care se vor defini endpoint-urile în funcţie de ce date vor fi prelucrate. De exemplu, în aplicaţie este prezent un controller denumit *UserController* care conţine toate endpoint-urile pentru operaţiile CRUD la nivelul informaţiilor despre user. Endpoint-urile sunt securizate în funcţie de rol. Rolul acestor controllerer este de a primi request-urile de la client şi de a oferi un răspuns client-ului după ce se vor executa operaţiile necesare la nivelul layere-lor de mai jos (services, repositories etc.)
- **service**: clasele de tip service vor prelucra datele obţinute din baza de date şi vor forma obiecte de tip DTO ce vor fi transmise controlerelor şi mai departe client-ului

- **jpa repository**: sunt interfețe unde sunt definite metode ce au acces la baza de date. Acestor metode sunt atașate query-uri ce vor fi executate pentru a obtine modelele din baza de date
- **jpa model:** sunt clase ce definesc modelele persistente ale aplicației cum ar fi useri, device-urile și măsuratorile. Aceste clase reprezintă modelul abstract al tabelelor din baza de date cu relațiile dintre acestea

Database:

- baza de date folosită este PostgreSQL; Aceasta a fost containerizată într-un container docker

5. DB Design

În continuare este prezentat designul bazei de date împreună cu diagrama ERD. Se observă că în baza de date sunt prezente 3 tabele în care se face persistența modelelor folosite în aplicație. De remarcat în cele 2 figuri sunt relațiile dintre tabele. Un user poate sa aibă mai multe device-uri, dar un device aparține unui singur user la un moment dat. Pentru măsurători, s-a ales să se rețină și user-ul pentru care s-a făcut măsurătoarea la un moment de timp, având în vedere faptul că pentru un device se poate modifica proprietarul. Astfel, pentru un user și un device putem avea mai multe măsurători.

Deoarece folosesc ca tool de migrare a bazei de date **Flyway**, acesta își va crea o tabelă în baza de ate unde va reține istoria script-utilor sql executate.

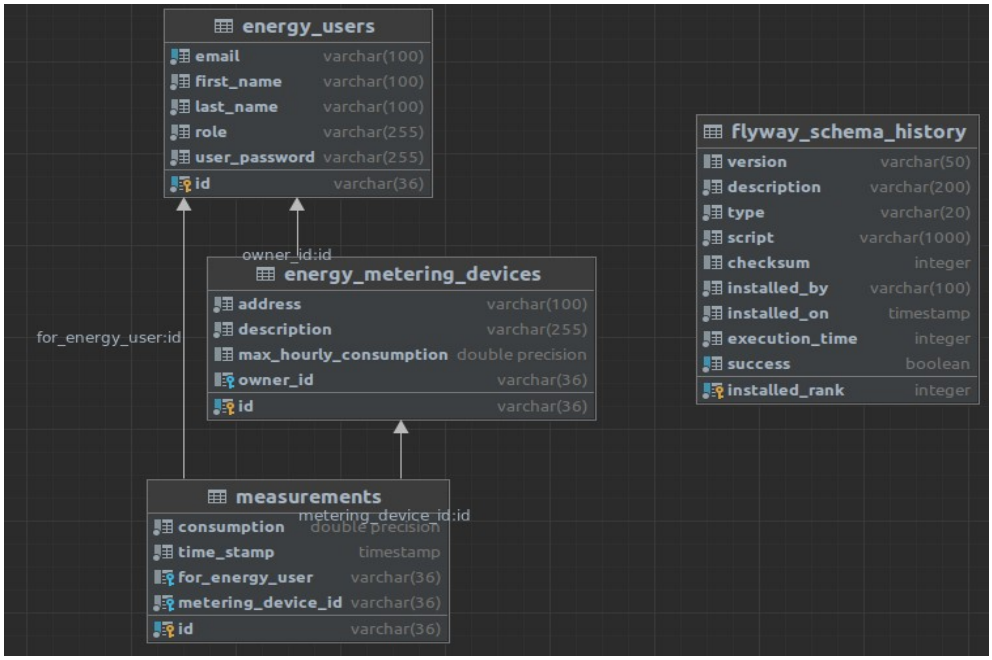

Figure 2: Database Design

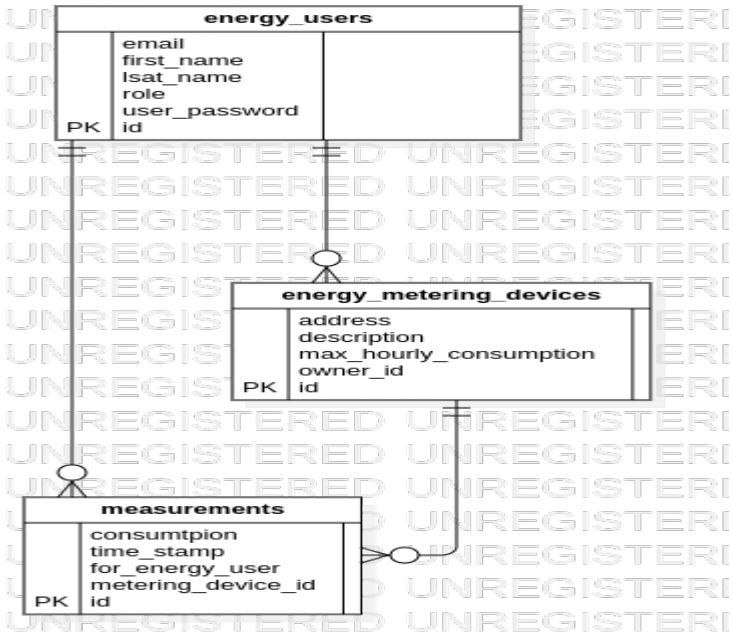

Figure 3: Diagrama ERD

6. UML Deployment Diagram

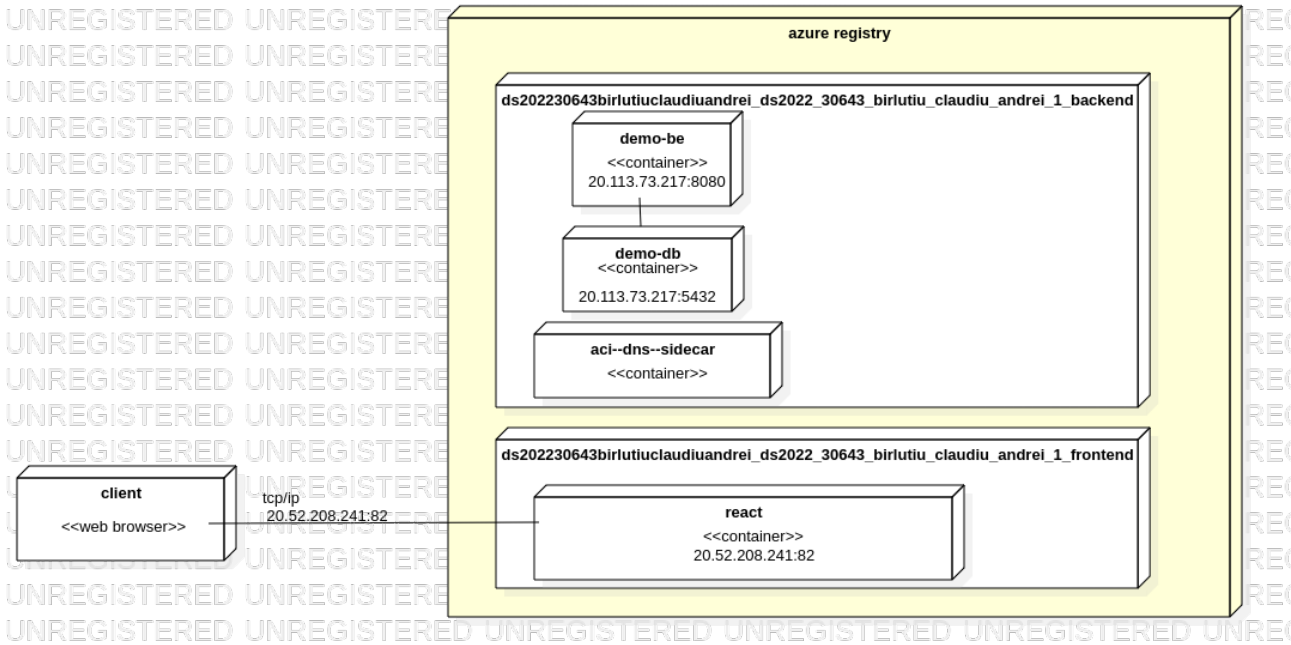

Figure 4: UML Deployment Diagram

În figura de mai sus este prezentată diagrama de deployment. Atât aplicația de backend, cât și cea de frontend au fost deployate în azure registry ca și containere docker. S-au integrat în azure

devops pipeline-urile pentru continous integration și continous deployment. Pentru continous integration, branch-urile *main* ale proiectelor de pe github sunt folosite ca trigger pentru declanșarea pipeline-ului de verificare a codului și formarea unei imagini noi de docker pentru codul sursă. S-a folosit un agent local care s-ă execute job-urile, respectiv task-urile descrise în pipeline-ul de CI.

Pentru partea de realease s-a ataşat tot agentul menitonat mai sus pentru execuția pașilor descriși în pipeline-ul de deployment. În urma execuției scripturilor, se va porni în contextul **aci** (cel legat de registry-ul remote de pe azure) noul container din ultima imagine creată din codul sursă.

Aplicația poate fi accesata din brower la link-ul http://20.52.208.241:82/login. Containerle de baza de date și backend au fost incluse în aceeași reatea 20.113.73.217 (portul 5432 pentru db, iar pentru serverul de backend s-a asociat portul 8080).

7. Build and execution considerations

Mediu local:

- **Aplicația client ReactJs**
  - **Prerequisite**: Node Js (14 or newer versions), Git
  - **Steps**:
    - clonarea repository-ului local
    - deschiderea unui terminal și executia următoarei comenzi pentru a instala pachetele node_modules: **npm install**
    - start the application using : **npm start**
    - acesare din browser : **localhost:3000**
- **Aplicația server Spring Boot**:
  - **Prerequsite**: Maven 3.8.6, Java 11, existența unei baze de date PostgreSQL cu database-ul **energey_monitoring_db**
  - Steps:
    - clonarea repository-ului de pe git local
    - setarea unei variabile de mediu care activează profilul de development al aplicației, pentru a putea fi rulată din intelliji SPRING_PROFILES_ACTIVE=dev
    - setarea variabilelor de conexiune la baza de date (ip, username, password, etc.)
    - rulează aplicația ori maven lifecycle pentru împachetarea într-un fișier jar

8. Concluzii

Pentru dezvoltatorul aplicației a fost o provocare partea de deployment și încă sunt anumite aspecte ce trebuie lămurite și îmbunătățite.