Sisteme Distribuite

Assignment 3
Energy Management System

Studenta: VASILACHE Maria

Grupa 30643

- **2023** -

1. Conceptual architecture of the distributed system

1.1. Descriere funcționalități proiect și tehnologii

În tema 2, am pornit de la funcționașitățile de bază implementate în assignment-ul 1, elementul de noutate, îl reprezintă folosirea unui microserviciu suplimentar. Mai precis, microserviciul de **Monitorizare,** unde s-a folosit Server-ul RabbitMQ, pentru a citi consumul pentru un dispozotiv selectat.

Datele sunt citite dint-un fișier CSV, sunt adăugate în coadă, citite și adăugate în baza de date. De asemenea, clientul va fi notificat, folosind WebSockets, cu privire la atingerea limitei maxime a consumului.

Vizualizarea datelor, este o altă funcționalitate nou-implementată în această tema, astfel, în urma citirii datelor senzorilor, am creeat un chart pentru o dată calendaristică selectată de user. Reprezentarea se face pe ore, astfel, utilizatorul va avea acces la consumul total.

Pentru simularea senzorului, am adăugat un field în care utilizatorul va introduce timpul in ms pentru care se dorește citirea.

Comunicarea între aplicații este **Asincronă,** întrucât avem aplicația desktop care citește date și le pune în coadă, iar aplicația de Monitorizare va citi datele și le va introduce în baza de date.

Prin urmare, această lucrare, ne familiarizează cu folosirea unui Server RabbitMQ, care ajută la rularea în paralel a mai multor aplicații acre introduc informații în baza de date.

În momentul pornirii mai multor servicii de simulare a consumului de energie, ele vor trimite simultan date, care vor fi puse în coada, când va fi un loc disponibil, mesajul va fi citit și se vor adăuga datele în baza de date.

De asemenea, s-au folosit WebSokets pentru a notifica clienții atunci când consumul maxim este atins.

Functionalitățile care sunt implementate în tema precedentă, pot fi urmărite în ceea ce urmează. Acest assignment presupune implementarea unei aplicații web, care deservește utilizatotilor, în funcție de rolul pe care il au, la monitorizarea dispozitivelor electronice pe care le dețin în locații diferite.

Menționând cerințele funcționale, aplicația se bucură de parte de login/register cu redirectare pentru user sau admini, fiind permise doat aceste două roluri. Un utilizator logat poate vedea lista sa de dispozitive, pe când, administratorii pot face opetatiile de: create, reade, update și delete asupra user-ilor și dispozitivelor. De asemenea, administratorii pot mapa dispozitive user-ilor, aceștia din urmă, deținând dispozitive la locații diferite.

Din punct de vedere arhitectural, pentru realizarea aplicației, am folosit frontend-ul, prin care userii trimit request-uri către server. Întrucât scalabilitatea este un element important, backend-ul, reprezentat de microserviciile pentru clienti și server, sunt două proiecte separate, cu două baze de date separate, care rulează pe porturi diferite.

Trecând în revistă tehnologiile folosite, am folosit: React (pentru frontend), Spring (pentru backend). Bazele de date relaționalale sunt accesate prin intermediul aplicației MySQL,

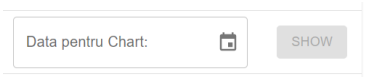
În urma implemenării, testării și verificării componentelor de proiect, atât separat, cât și prin integrarea acestora, am finalizat lucrarea prin deploy-ul în docker.

1.2. Detalii implementare

Administratorul poate vedea device-urile unui utilizator, pentru assignmentul 2, am adăugat un calendar, din care se poate selecta data la care vor să se analizeze consumul pe ore.

Utilizatorul va putea vizualiza chart, doar daca a selectat data din **pick calendar,** altfel butonul va fi disable.

Inainte de selectare data:



Dupa de selectare data:

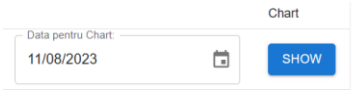
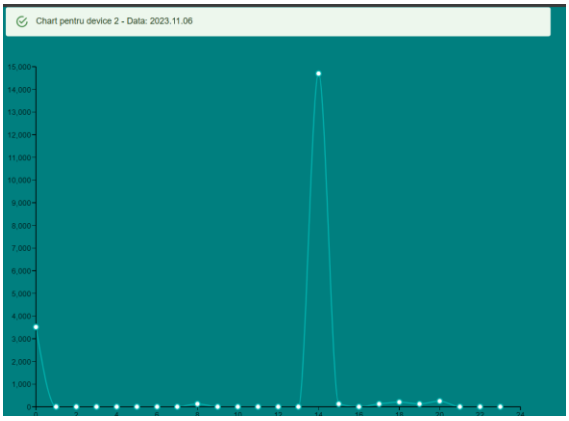


Chart-uri:

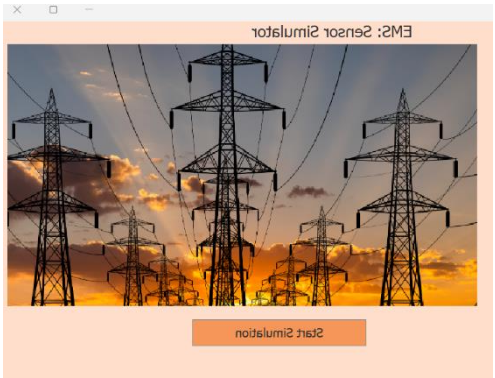
Cu titluri sugestive referitoare la device-ul citit și data pentru care se realizează raportul

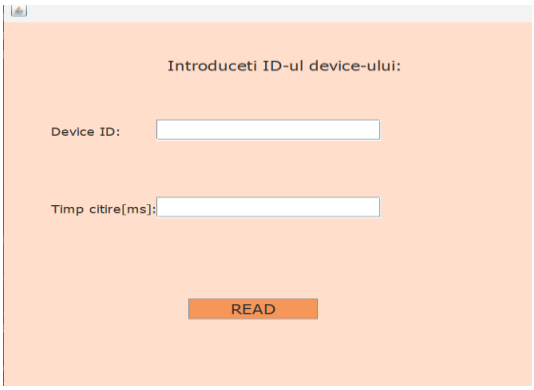


Simulation Senzor:

Administratorul va porni simularea folosind butonul corespunzător device-ului căruia se dorește să i se măsoare datele primite de la sensor. ID-ul device-ului poate fi preluat de la Device-ul selectat.

Pentru a fi mai ușor de testat, am introdus un field îm care user-ul poate introduce timul în ms, pentr





1.3. RabbitMQ

Este un Server care este instalat pe computer, se pot vedea mesajele folosind browser-ul, accesând

http://localhost:15672/

Logare la RabbitMQ

-user: guest

-Parola: guest

RabbitMQ, Broker-ul de mesaje asigură comunicarea între sistemele distribuite, mai precis, între Devices și Senzori.

În implementare am folosit **protocolul Advanced Message Queuing Protocol (amqp).**

În Spring, am folosit dependințete maven:

```
<!-- https://mvnrepository.com/artifact/com.rabbitmq/amqp-client -->
<dependency>
   <groupId>com.rabbitmq</groupId>
   <artifactId>amqp-client</artifactId>
   <version>5.20.0</version>
</dependency>
```

Pentru folosirea Server-ului, propriu-zis, a fost necesară instalarea următoarei dependințe:

```
<!-- https://mvnrepository.com/artifact/com.rabbitmq/amqp-client -->
<dependency>
   <groupId>com.rabbitmq</groupId>
   <artifactId>amqp-client</artifactId>
   <version>5.20.0</version>
</dependency>
```

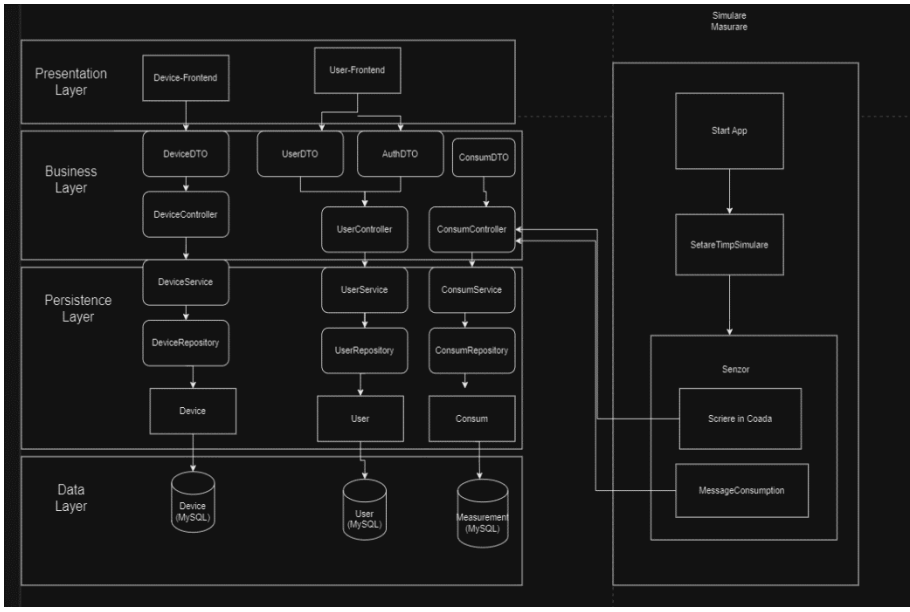
1.4. Arhitectura proiectului

Proiectul este construit pe arhitectura **Layered Architecture.**

Elementele care fundamentează această arhitectură sunt enumerate și prezentate în cele ce urmează:

a. **Presentation Layer:** este întâlnită partea de frontend specifică fiecărui tip de utilizator, aceștia din urmă trimițând request-uri, urmând ca server-ul să trimită răspunsuri, ce sunt vizibile utilizatorilor. Pentru implementarea frontend-ului am folosit **React,** astfel, am organizat acțiunile îm funcție de rolul personaei logate.

b. **Business Layer:** în această parte este întâlnită partea de logică a aplicației. Această parte, pune la o laltă. Entitățile care corespund Tabelelor din Baza de date: user, devices



Comunicarea frontend-backend si transmiterea de request-uri este posibilă prin DTO, care funcționează ca un API.

c. **Persistence/Repository Layer:** Sunt necesare, împreună cu JPA Repository la efectuarea operațiilor Create-Read-Update-Delete asupra bazelor de date.
Instanțele de **Service** ajută la implementarea logicii penru operațiile cu baza de date.

d. **Data Layer** - in acest layer sunt vizate bazele de date, în număr de două, acoperind microserviciile user si device.

Update Assignment 2:

e. **Pentru simularea Citirii senzorilor, avem o aplicație Desktop**.
În cadrul acesteia, în funcție de ID-ul preluat din WebApp și timpul introdus manual, se va porni simularea. La fiecare moment de timp se citește valoare detectată de senzorul simulat dintr-un fișier .csv

Pentru a reține valoarea la care s-a făcut ultima citire, pentru fiecare dispozitiv, se va creea un fișier txt, în care se scrie ultima linie citită, de acolo va continua la umătoarea citire, în cazul în care valoarea maximă a consumului s-a atins, nu se vor mai citi date, și se va trimite o notificare în pagina Web.

Folosirea cozii pentru citirea dispozitivelor – se creează o coadă cu un nume dat, urmând să se trimită în format json mesajul.

```
Connection connection = connectionFactory.newConnection();
Channel channel = connection.createChannel();
channel.queueDeclare("rabbitmq_queue", false, false, false, null);
String json = new ObjectMapper().writeValueAsString(measurement + "status:"+status);
channel.basicPublish("", "rabbitmq_queue", null, json.getBytes());
```

Preluarea datelor din Queue:

```
Channel channel = connection.createChannel();
channel.queueDeclare("rabbitmq_queue", false, false, false, null);
System.out.println(" [Coada rabbitmq_queue creata: se asteapta mesajele transmise]");
channel.basicConsume("rabbitmq_queue", true, deliverCallback, consumerTag ->  });
```

f. Folosirea Socket-urilor pentru timiterea notificărilor:
În Backend, la Dispozitive, se calculează valoarea curentă, cumulând cu ce s-ar adăuga. Dacă se depășește valoarea maximă caracteristică dispozitivului citit, se va deschide un Socket și se va trimite un mesaj caracteristic, în frontrnd, user-ul va vedea în pagină o **alertă**.

Se va configura Socket-ul astfel:

```
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {
   @Override
   public void registerStompEndpoints(StompEndpointRegistry stompEndpointRegistry) {
      stompEndpointRegistry.addEndpoint("/socket")
            .setAllowedOrigins("http://localhost:3000/")
            .withSockJS();
   }
   @Override
   public void configureMessageBroker(MessageBrokerRegistry registry) {
      registry.enableSimpleBroker("/topic");
   }
}
```

Trimiterea mesajului, se face printr-un template:

```
private final MonitorizareService monitorizareService;
String sursa = "/topic/socket/device"+"/"+ String.valueOf(device_ID);
simpMessagingTemplate.convertAndSend(sursa, "Mesaj din Backend Device: " + device_ID + "\nStart simulation");
```

În Frontend, se va deshide Soket-ul, citind ce s-a trimis de la Back:

```
export const subscribe = (DeviceID)=>{
    const websocket  = new SockJS('http://localhost:8082/socket')
   const stompClient = Stomp.over(websocket);
  stompClient.connect({}, (id, headers) => {
     stompClient.subscribe("/topic/socket/device/"+DeviceID, message => {
          alert(message.body)ș
     });  });}
```

g. Transmitere mesaje din Front → Backend

1. Configurare Socket:

```
new *
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {
    no usages  new *
    @Override
    public void registerStompEndpoints(StompEndpointRegistry stompEndpointRegistry) {
        stompEndpointRegistry.addEndpoint( ...paths: "/socket")
                .setAllowedOrigins("http://localhost:3000/")
                .withSockJS();
    }
    no usages  new *
    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker( ...destinationPrefixes: "/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }
}
```

2. Tratarea mesajelor primite

```java
@Controller
public class WebSocketController {

    new *
    @MessageMapping("/sendMessage")
    @SendTo("/topic/receivedMessage")
    public String handleWebSocketMessage(String message) {
        //Mesaj primit
        System.out.println(message);
        return "Server received: " + message;
    }
}
```

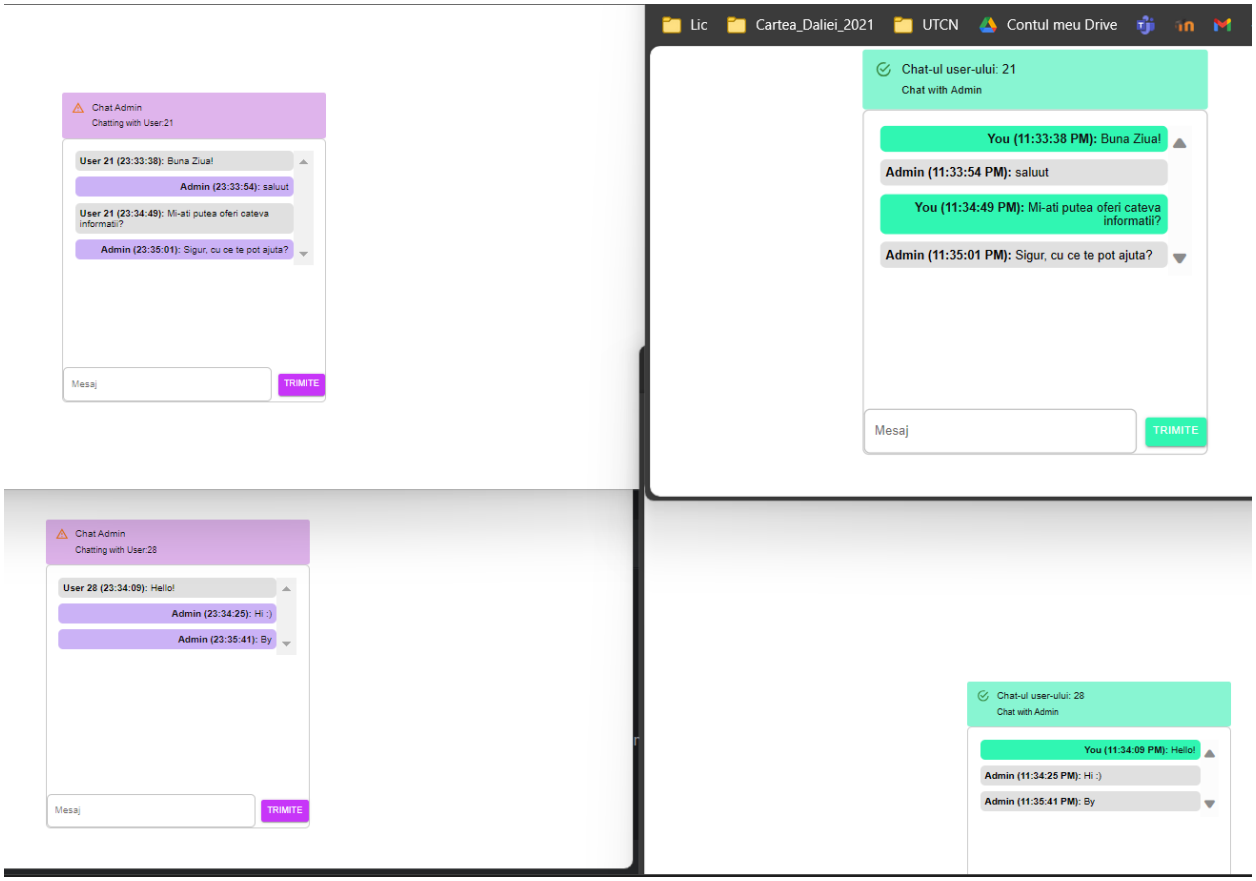
3. Trimiterea mesajelor din Front (JS)

```javascript
const socket : SockJS  = new SockJS('http://localhost:8082/socket');
const stompClient : CompatClient  = Stomp.over(socket);

stompClient.connect( args: {}, function (frame) : void  {
    console.log('Conectat la server WebSocket');

    // Apelează functia pentru a trimite un mesaj către /app/sendMessage
    sendMessage('Acesta este un mesaj din frontend');
});

1 usage  new *
function sendMessage(message) : void  {
    // Trimite mesajul către endpoint-ul /app/sendMessage
    stompClient.send( destination: '/app/sendMessage',  headers: {}, JSON.stringify( value: { content: message }));
}
```

Pentru consultarea diagramei realizate pentru Layered architecture a aplicației Energy Management System, puteți accesa link-ul:
https://drive.google.com/file/d/1lKaPHYkDJcMhSNLH2SztFC6lQiUOfZRh/view?usp=sharing

Update tema 3

1. Chat-ul unui Admin cu mai multi useri folosind WebSockets:



Detalii implementare – WebSockets Tema 3

1. WebSocketController -Primire mesaje:

```
@MessageMapping("/sendMessage")
@SendTo("/topic/receivedMessage")
public String handleWebSocketMessage(String message) {
    String mesajTransmis ="";
    int idDestinatar = 0;
    String sursa = null;
    /*** Mesaj primit din front!!!!*/
    System.out.println(message);
```

2. WebSocketController -Transmiterea mesaje:

```
sursa = "/topic/socket/device" + "/" + String.valueOf(idDestinatar);
```

```
sursa = "/topic/socket/deviceAdmin" + "/" + String.valueOf(idDestinatar);
```

```
simpMessagingTemplate.convertAndSend(sursa, mesajTransmis);
```

3. Frontend- primire mesaje de la Admin:

```
useEffect( effect: () : void => {
    const websocket : SockJS = new SockJS('http://localhost:8084/socket');
    const stompClient : CompatClient = Stomp.over(websocket);

    stompClient.connect( args: {}, ( headers) : void => {
        stompClient.subscribe( destination: "/topic/socket/device/"+id, callback: message : IMessage => {
            const newMessage : {text: string, timestamp: number} = { text: message.body, timestamp: Date.now() };
            setReceivedMessages( value: (prevMessages : any[] ) => [...prevMessages, newMessage]);
        });
    });
}, deps: [id]);
```

4. Frontend- transmitere mesaje catre Admin:

```
const socket : SockJS = new SockJS('http://localhost:8084/socket');
const stompClient : CompatClient = Stomp.over(socket);
1 usage
function sendMessage(message) : void {
    stompClient.send( destination: '/app/sendMessage', headers: {}, JSON.stringify( value: { mesajUser: message, UserID: id }));
}
```

5. Front- primire mesaje de la un User specific

```
useEffect( effect: () : void => {
    const websocket : SockJS = new SockJS('http://localhost:8084/socket');
    const stompClient : CompatClient = Stomp.over(websocket);

    stompClient.connect( args: {}, ( headers) : void => {
        stompClient.subscribe( destination: "/topic/socket/deviceAdmin/"+id, callback: message : IMessage => {
            const newMessage : {text: string, timestamp: number} = { text: message.body, timestamp: Date.now() };
            setReceivedMessages( value: (prevMessages : any[] ) => [...prevMessages, newMessage]);
        });
    });
}, deps: []);
```

6. Trimitere mesaje catre un User Specific

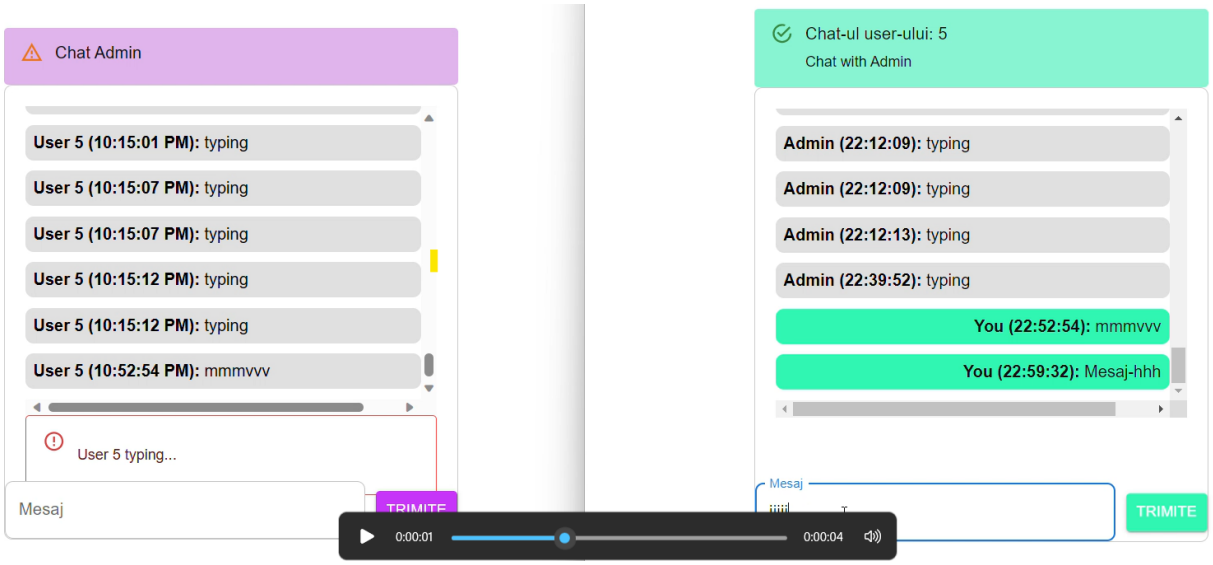
```
const socket : SockJS = new SockJS('http://localhost:8084/socket');
const stompClient : CompatClient = Stomp.over(socket);
1 usage
function sendMessage(message) : void {
    stompClient.send( destination: '/app/sendMessage', headers: {}, JSON.stringify( value: { mesajAdmin: message, sendToUserID: id }));
}
```

Detalii implementare persistare mesaje:

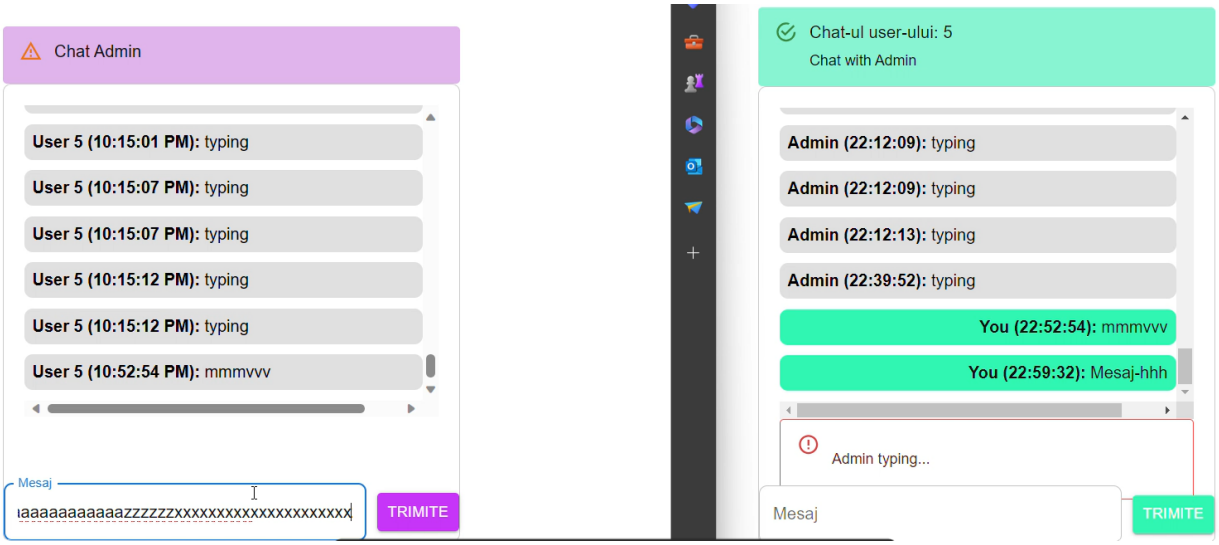
```
useEffect( effect: () : void => {
    localStorage.setItem(`chatMessages_${id}`, JSON.stringify(messages));
}, deps: [id, messages]);
```

Detalii notificare cand user-ul scrie:

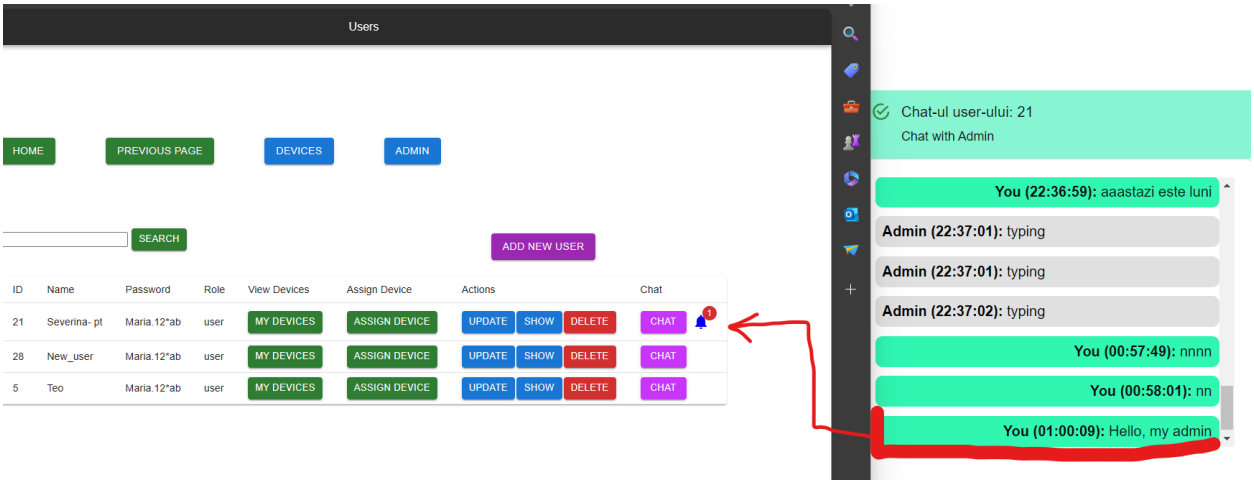
"Clienul X typing" : notificare trimisa Adminului cand ussr-ul scie



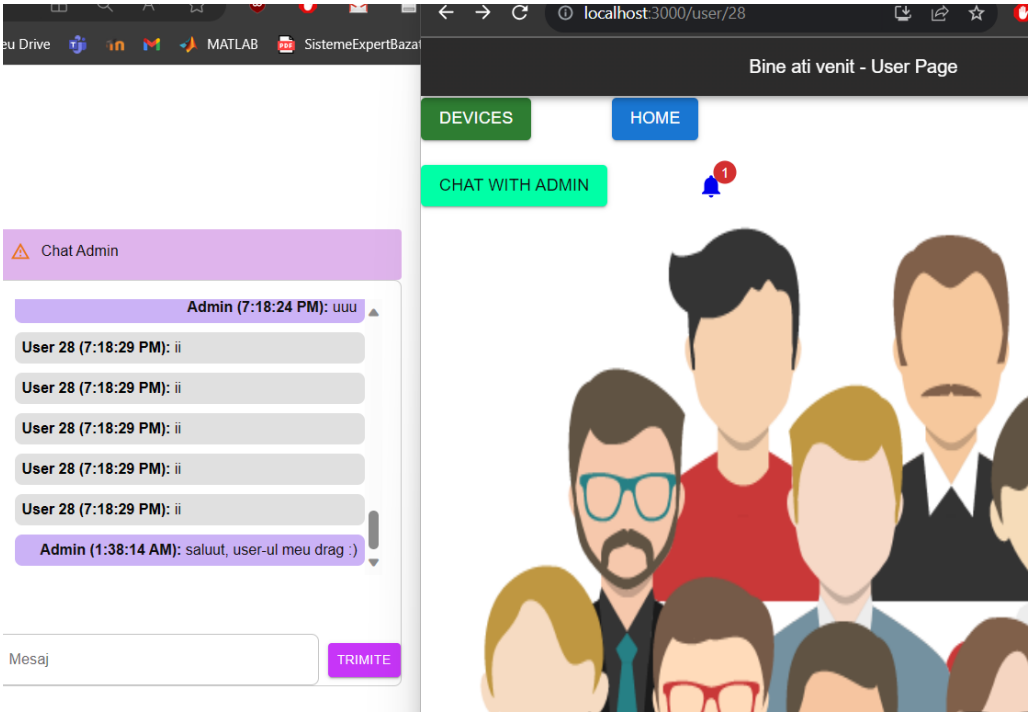
"Admin typing" : notificare trimisa Clientului cand adminul ii scrie



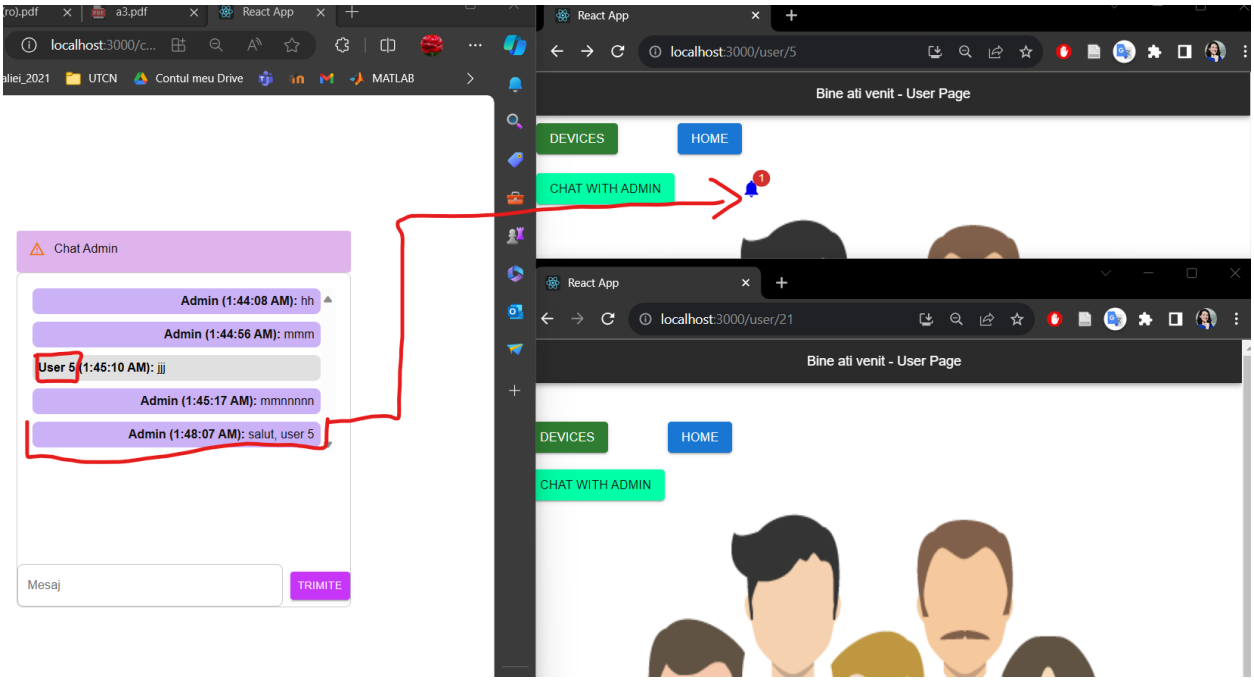
Detalii trimitere mesaje de la User la Admin (notificare)



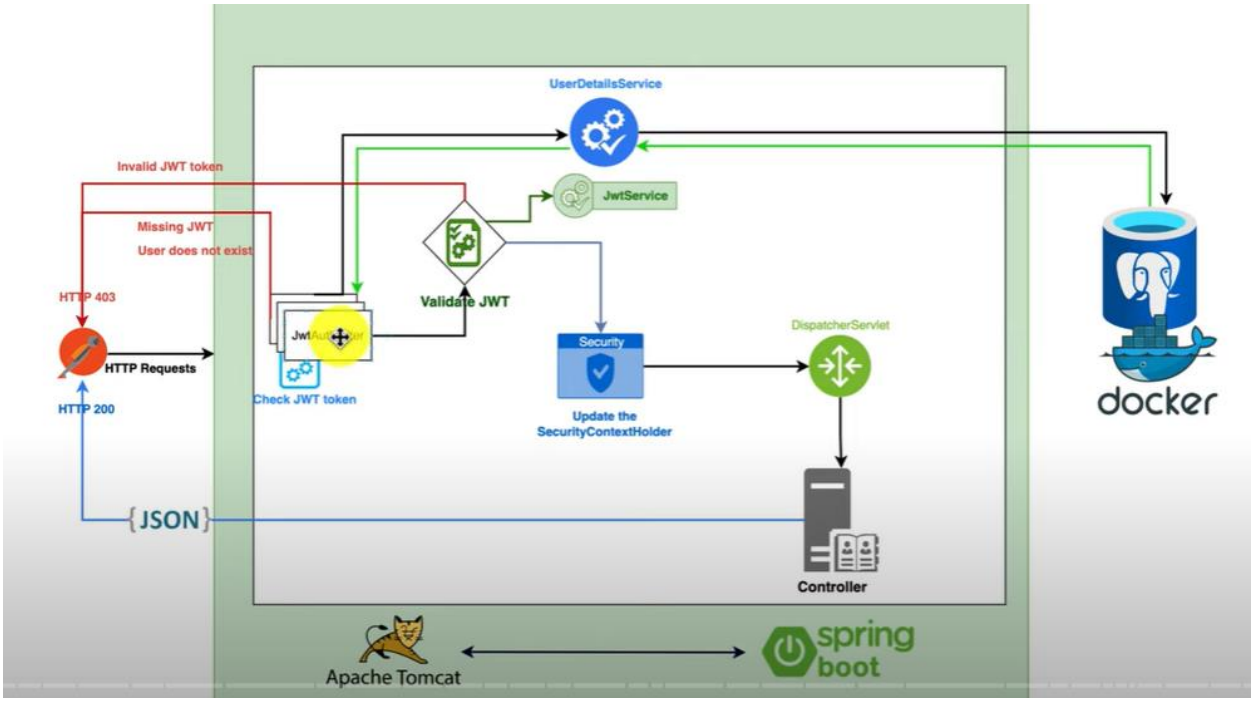
Detalii trimitere mesaje de la Admin la un anumit User(notificare)



Daca avem mai multi useri conectati, notificarea impreuna cu mesajul vor fi trimise in mod corespunzator



Spring Security



JWT foloseste un token care va extrage email/username, dupa ce acesta este obtinut, se va merge la baza de date si se vor accesa din DB informatii relevante(dorite).

Cand token-ul JWT este valid, se SecurityContextHolder se valideaza→sdispatch Server→Controller

Flow:

1. Register
2. Se genereaza token
3. Se decodeaza si se va vedea user-ul
4. Authentication
5. Doar daca user si password sunt valide, se genereaza token
6. In request →Authentication, se insereaza token-ul

WebSockers Connection

Se folosesc cookies si tokens, astfel, nu se va putea trimite un mesaj decat dupa ce se verifica token-ul!

Dependinte

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>6.2.0</version>
</dependency>
```

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.3</version>
</dependency>
```

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.3</version>
    <scope>runtime</scope>
</dependency>
```

<dependency>
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt-jackson</artifactId>
<version>0.12.3</version>
<scope>runtime</scope>
</dependency>

UserDetails Interface

```
public class User implements UserDetails
```

//Va returna rolurile

```
public Collection<? extends GrantedAuthority> getAuthorities() {
    return List.of(new SimpleGrantedAuthority(String.valueOf(role)));
}
```

```
public String getUsername() {
    return name;
}
```

```
public boolean isAccountNonExpired() {
    return true;
}
```

...

```
public String getPassword() {
    return password;
}
```

Decodificare cheie:

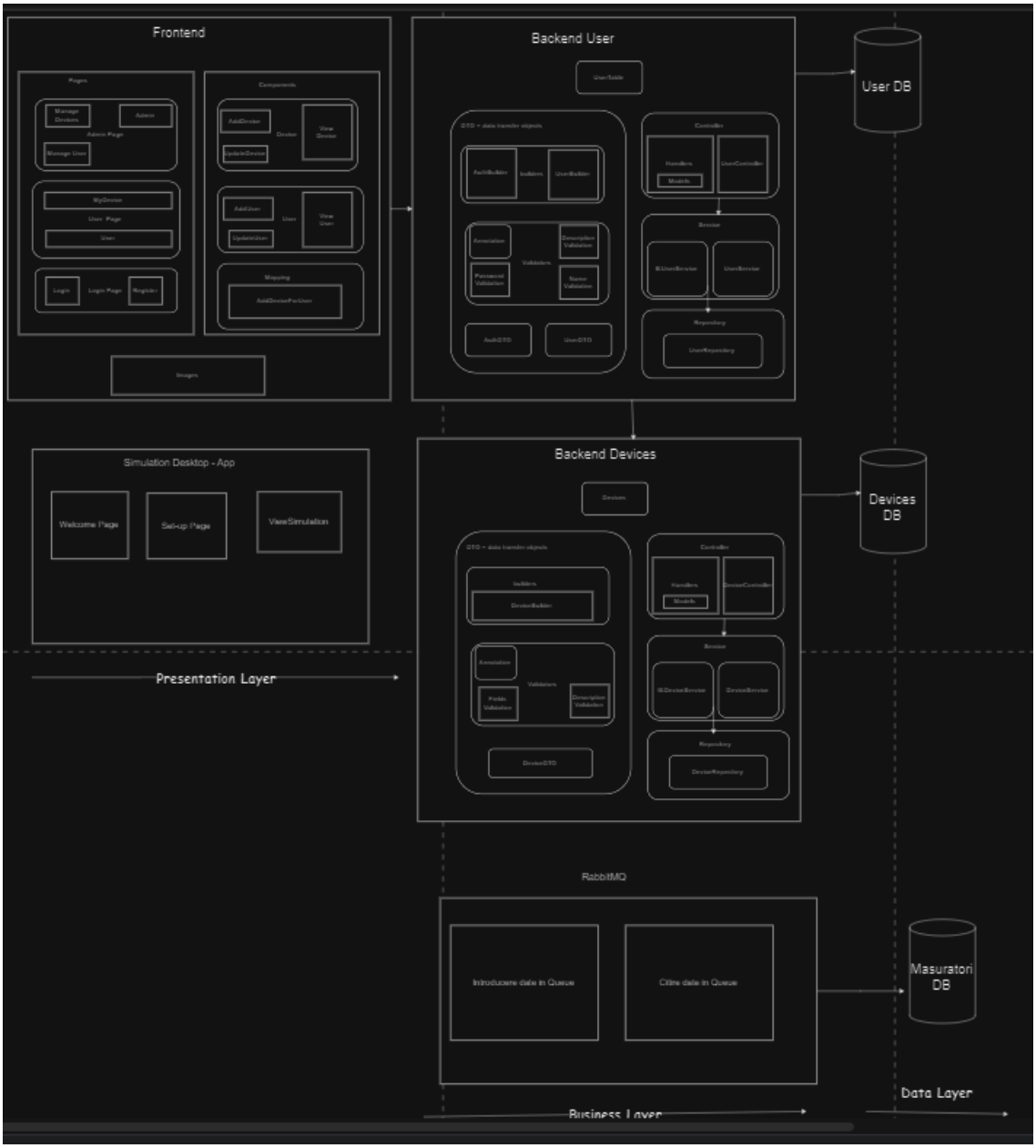
```
private Key getSignInKey() {
    byte[] keyBytes = Decoders.BASE64.decode(SECRET_KEY);
   //algoritm de decriptare
    return Keys.hmacShaKeyFor(keyBytes);
```

Obtinere/Returnare Token:

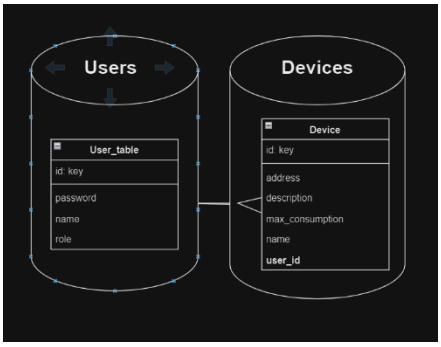
```
//va returna Token-ul
public String generateToken(Map<String, Object> extraClaims,
                            UserDetails userDetails)
{
    return Jwts
            .builder()
            .setClaims(extraClaims)
            .setSubject(userDetails.getUsername())
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis()* 1000*60*24))
            .signWith(getSignInKey(), SignatureAlgorithm.HS256)
            .compact();
}
```

1.5. Arhitectura conceptuala

Pentru o vizualizare mai bună a diagramei, puteți accesa fișierul in Drive:
https://drive.google.com/file/d/1S9irDGOKv5iKiC1tjY_h8IKeJRT5jCl_/view?usp=sharing


Frontend
Backend User
User DB
Simulation Desktop - App
Welcome Page
Set-up Page
ViewSimulation
Backend Devices
Devices DB
Presentation Layer
RabbitMQ
Introducere date in Queue
Citire date in Queue
Masuratori DB
Data Layer
Business Layer

2. Data Base design



Data Layer este constituit pe 2 baze de date, baza de date **users și devices**, fiind specice fiecărui microserviciu al aplicației. Baza de date pentru useri, conține tabelul **user_tabel,** în acesta sunt stocate câmpurile: id, name, password, rol. Iar baza de date pentru device-uri, are în componența sa tabelul **device,** în acest tabel, informațiile stocate sunt: id, address, description, name, max_consumption, user_id.

Pentru sincronizarea celor două baze de date independente, am folosit în tabela devices atributul **user_id,** astfel, când unui user i se selectează un dispozitiv, în tabela dispozitov, pentru id-ul device-ului adăugat, se va face operația de update, adăugându-se id-ul user-ului corespunzător asignării. Pentru sporirea simplității, la crearea de device-uri noi, user_id, va avea valoarea null.

Întrucât datele pe care le introducem în baza de date sunt preluate din frontend, ajung în backend și în urma operațiilor defenite în backend, se vor adăuga în baza de date, am implementat în backend partea de validare a câmpurilor, pentru a asigura calitatea datelor, iar acestea să nu fie corupte.
Motivația pentru folosirea a două baze de date diferite este reprezentată de evidențierea scalabilității, astfel, microserviciile pentru user si device-uri, au fost fundamentale pentru a îndeplini această cerință.

In alta ordine de idei, relatia intre cele doua tabele este **one-to-many,** un user, putand detine mai multe device-uri.

Update Assignment 2:

Am mai creat o vaza de date, numită monitorizare, aceasta find necesară microserviciului de simulare al dispozitivelor.

Această nouă DB, conține: valoare in KW a consumului, data la cere s-a preluat înregistrarea, ID-ul device-ului pentru care s-a efectuat simularea.

3. UML Deployment Diagram

Acest proiect, în urma implementării și testării locale, pe dispozitivul propriu, s-a rulat pe Docker, acest pas reprezentând deploy-ul aplicației.
Din punct de vedere structural. Având 3 proiecte, reprezentate de 2 microservicii: user, device și partea de frontend, în Docker am creat **5 containere,** nimite și nod-uri. Numarul de containere se justifică pentru următoarele atribuții îndeplinite:

a. Nod/Container pentru **Frontend**
b. Nod/Container pentru **Baza de date USER**
c. Nod/Container pentru **Baza de date DEVICE**
d. Nod/Container pentru **Backend USER**
e. Nod/Container pentru **Backend DEVICE**

Update Assignment 2:

f. Nod/Container pentru b**aza de date MASURATORI**
g. Nod/Container pentru **Backend Senzor**
h. Nod/Container pentru RabbitMQ

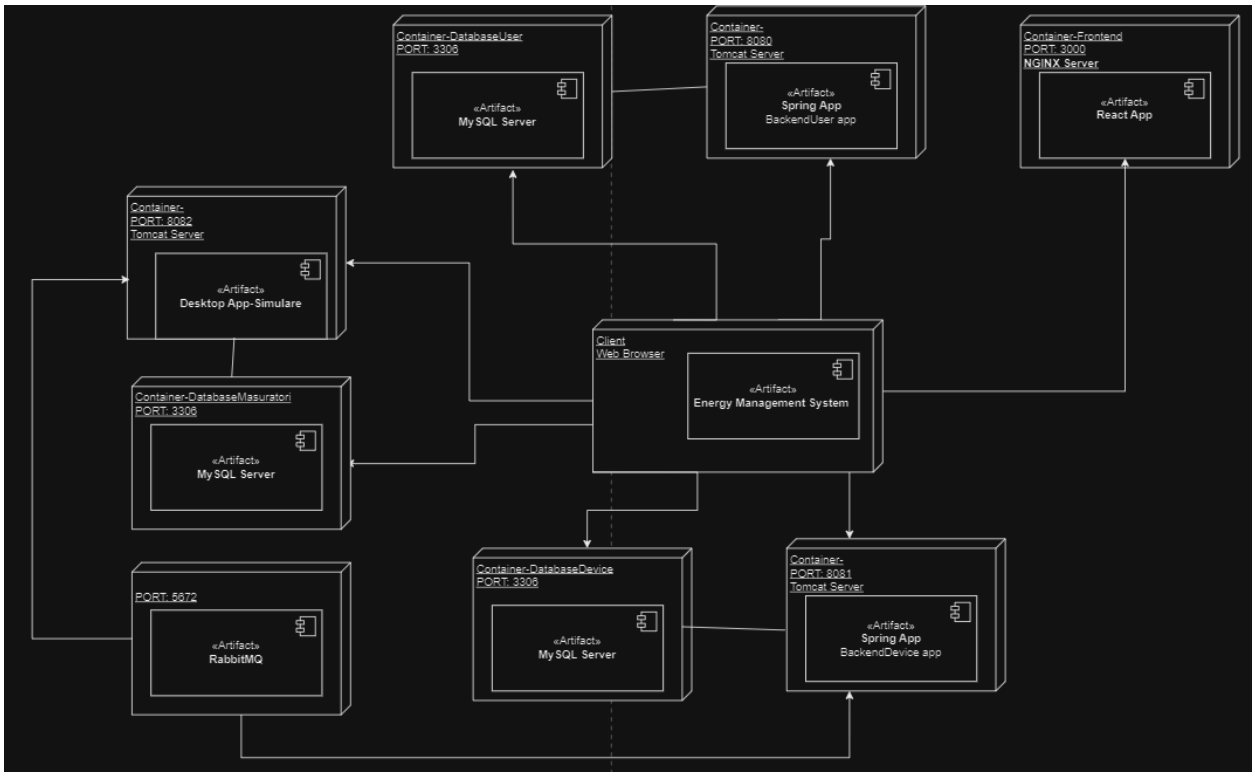
Aplicația poate fi accesată de client, prin intermediul Broser-ului Web.

De subliniat este faptul că cele 2 baze date sunt construite în MySQL, iar prin prisma faptului că proiectele de backend trebuie să ruleze simultan, fiindcă cele 2 microservicii trebuie să comunice în timpul execuției programului, proiectele rulează pe port-uri diferie.

Frontend-ul este implementat folosind React si JavaScript.

Backend-ul l-am scris, folosind Spring si Java.

Diagrama UML care sugerează deploy-ul în Docker, poate fi consultată în imaginea sau link-ul atașat: https://drive.google.com/file/d/1hm65bOJ9_VVowpp0UtISsWZu23ezXJ_z/view?usp=sharing



4. Build And Execution Considerations

Pașii pe care i-am urmat pentru realizarea acestui proiect, au fost înțelegerea cerințelor și alegerea tehnologiilor potrivite pentru atingerea obiectivului, cât și punerea în valoare a cerințelor nonfuncționale, printre care amintesc: scalabilitate.

IDE-ul pe care l-am folosit pentru support în implementarea frontend-ului, cât și a backend-ului a fost InteliJ, acesta ajutându-mă să urmăresc erorile de sintaxă, iar integrarea cu linia de comandă a fost un plus adus în alegerea acestui IDE.

Pentru crearea și monitorizarea bazei de date, am folosit MySQL, întrucât este o aplicație cu care am lucrat în trecut, și de asemenea, oferă o interfață intuitivă, existând butoane cu nume sugestive pentru crearea de baze de date, tabele, putând fi scrise comezi atat sub formă de Quey-uri, cât și ca request-uri prin apăsare de butoane. De altfel, mesajele sugestive pentru erorile apărute la diferite interogări asupra bazei de date, ajută la debug-ing.

Partea de backend, presupune comunicarea cu baza de date, astfel, toate query-urile associate butoanelor apăsate în frontend, le sunt asociate funcții scrise în proiectele specifice Având în vededere că

request-urile pot fi greșite sau nu produc rezultatul dorit, am folosit **Postman** pentru a vedea codurile de eroare, sau codurile de succes ale acțiunilor HTTP. De pildă, codul 200, sau 201, reprezintă un cod de succes, iar în cazul statusului 500, eroarea de server, poate anticipa o conexiune greșită cu baza de date, iar eroarea 400, semnifică un request scris greșit, sau lipsa body-ului cu cod JSON pentru request-uri POST/PUT. Diferența dintre POST si PUT fiind în strânsă legătură cu idempotența. PUT – idempotentă, POST – nu e idempotentă.Idempotență: același rezultat la repetări.

Odată implementat și testat, proiectul îndeplinind cerințele dorite, a fost deploy-at utilizând Doker. Pașii pentru a face deploy, au fost următorii:

a. Pentru fiecare proiect de backend și frontend s-a creat un **Dockerfile**
b. Pentru fiecare proiect de backend și frontend s-a creat un **docker-compose.yaml**
c. Construirea imaginii doker: docker-compose build
d. Rularea imaginii: docker-compose up
e. Rularea aplicației in Docker Desktop

5. ReadMe

Run Frontend app

cd ./frontend
npm install
npm start
Navigate to http://localhost:3000/

Run Spring app

cd ./backend
mvn spring-boot:run