UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Energy Management System

DISTRIBUTED SYSTEMS

Nume: Muresan Ioana Danina
Grupa: 30643

Facultatea de Automatica si Calculatoare

27 Noiembrie 2023

Cuprins

1 Arhitectura conceptuala a sistemului

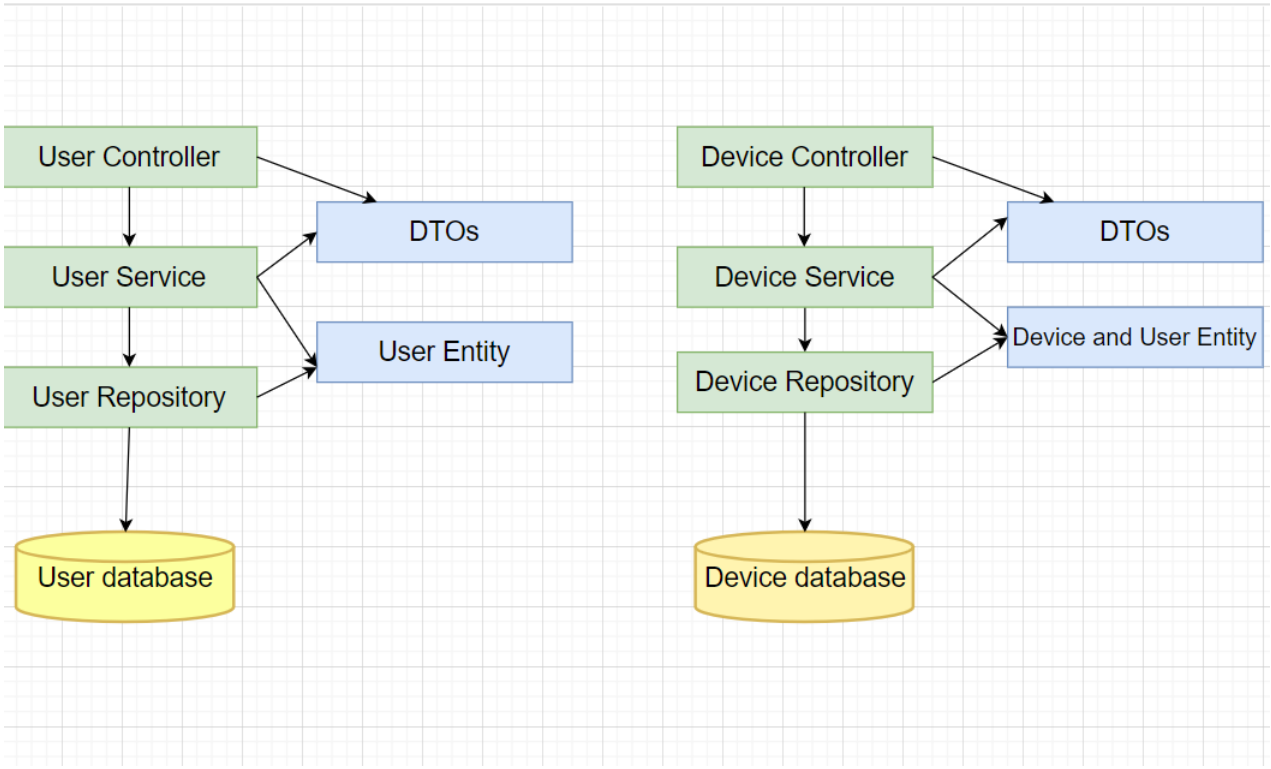
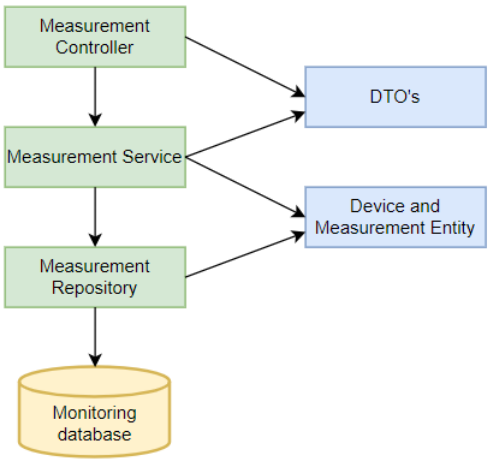


Figura 1: Arhitectura conceptuala

Am adaugat pentru tema 2



De asemenea,am modificat arhitectura conceptuala pentru tema2.

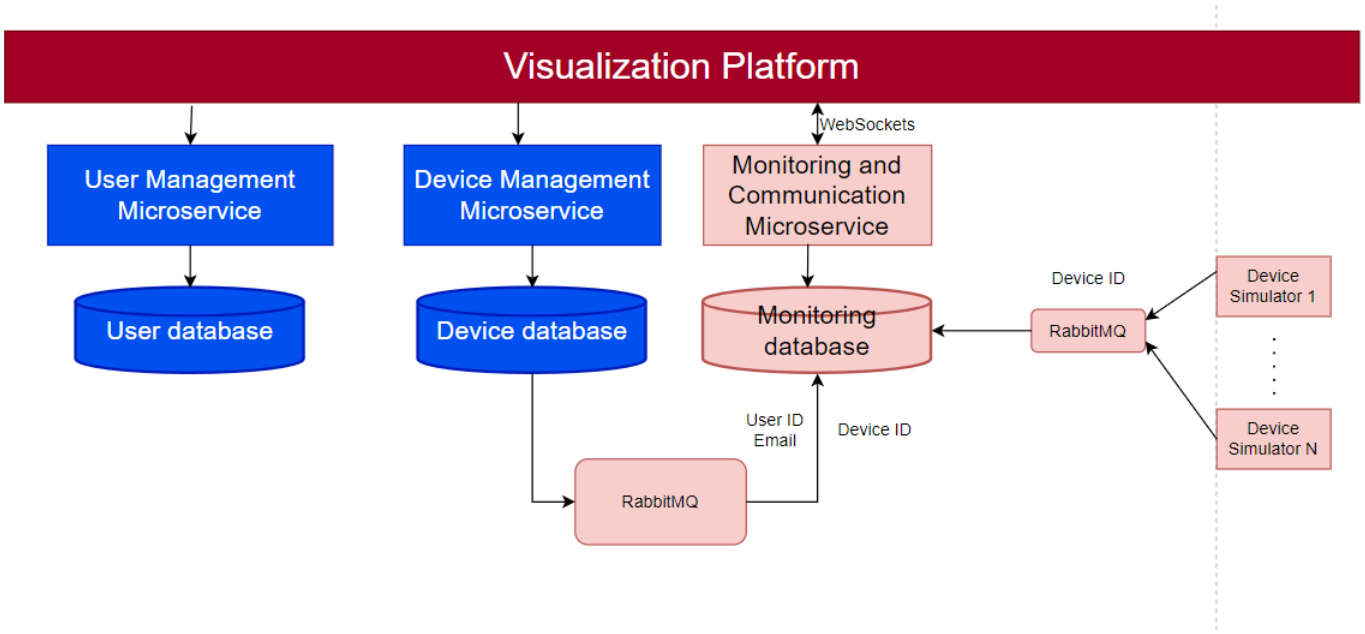
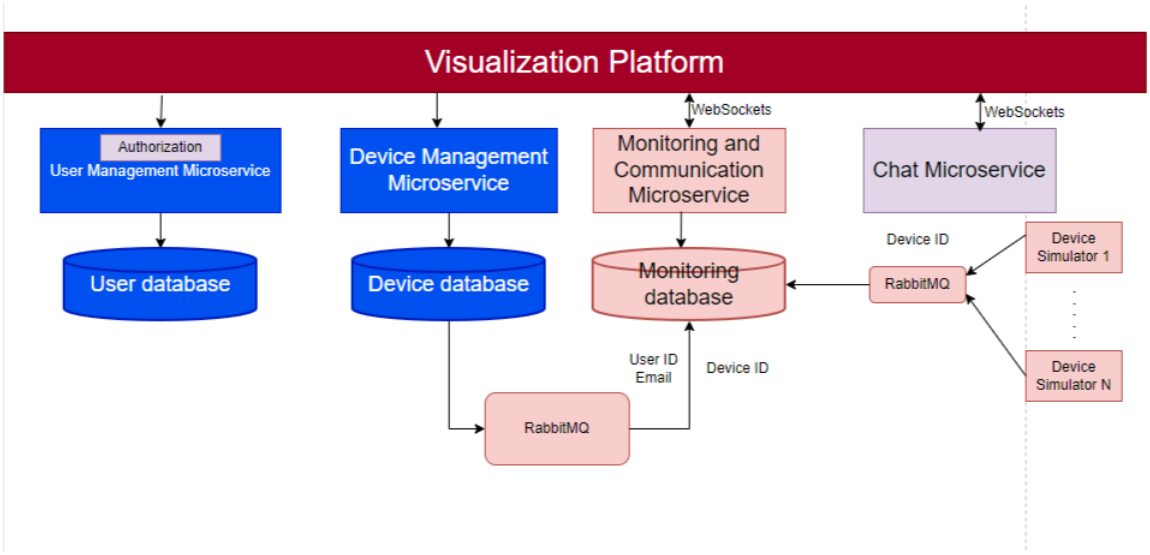


Figura 2: Arhitectura conceptuala

Am adaugat pentru tema 3



2 Diagrama UML de deploy

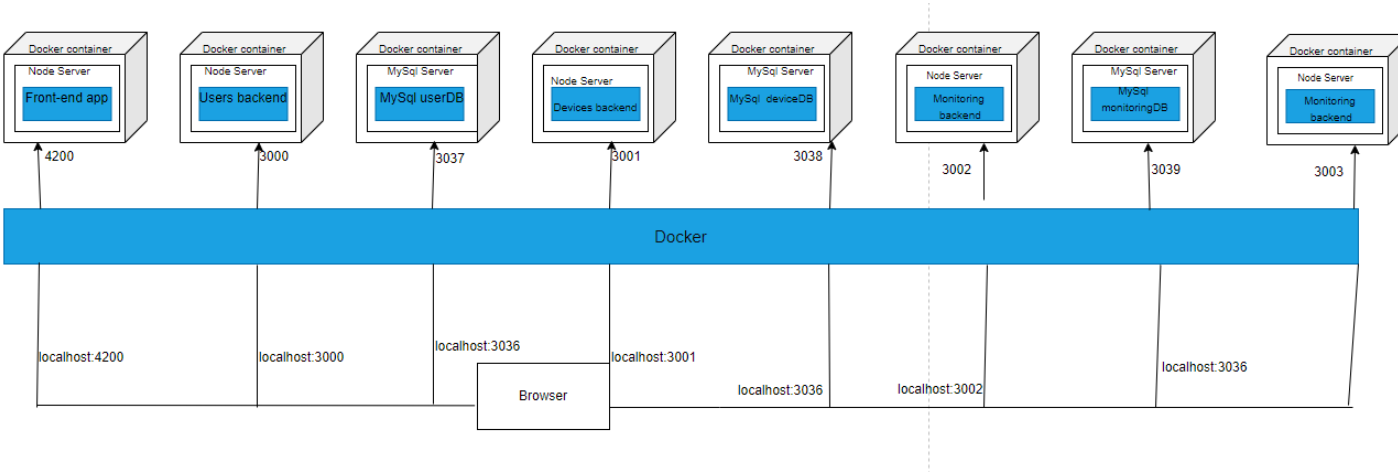


Figura 3: Arhitectura conceptuala

3 Readme

3.1 Descriere

Această aplicație constă din trei backend-uri (micro-servicii) Nest.js, fiecare cu propria bază de date,pentru care am utilizat MySql, proiectate pentru administrarea utilizatorilor și dispozitivelor lor inteligente de măsurare a energiei și un frontend Angular care facilitează interacțiunea cu aceste servere.Sistemul poate fi accesat de către două tipuri de utilizatori după un proces de autentificare: administrator (manager) și clienți.

Pentru tema 2 am implementat si o aplicatie Smart Metering Device Simulator ca Producator de mesaje. Aceasta simuleaza un contor inteligent citind datele de energie dintr-un fisier sensor.csv (adica, o valoare la fiecare 10 minute) si trimite date sub forma ¡ timestamp, deviceid, measurement-value ¿ catre Broker de mesaje (adica, o coada). Marca temporala este preluata de la ceasul local, iar deviceid este unic pentru fiecare instanta a Simulatorului dispozitivului de masurare inteligenta si corespunde dispozitivului id a unui utilizator din baza de date (asa cum este definita in Tema 1). Simulatorul de dispozitiv este dezvoltat ca o aplicatie autonoma (adica, aplicatie desktop).

3.2 Micro-servicii

Pentru utilizarea celor doua micro-servicii de Nest, se va proceda astfel: se verifica initial daca exista Node.js pe calculatorul respectiv, in caz contrar de va instala ruland intr-un terminal **install node.js**, se verifica daca este instalat **npm**. Pe urma,se deschide fisierul in care este salvat proiectul si un terminal.Se ruleaza **npm install** pentru a descarca toata librariile,dependintele utilizate. Apoi se ruleaza comanda **npm run start:dev** sau **npm run start** pentru a porni aplicatia. Daca totul functioneaza ok, terminalul va afisa **LOG [NestApplication] Nest application successfully started**. Micro-serviciul de user ruleaza pe portul localhost:3000 ,micro-serviciul de monitoring pe localhost:3002 iar micro-serviciul de deviceuri ruleaza pe portul localhost:3001. Ambele baza de date ruleaza pe portul 3036. Microserviciul de monitorizare și comunicare are o componentă Message Consumer care va procesa măsurătorile pentru a calcula consumul total de energie pe oră și îl va stoca în Bază de date. Dacă consumul total de energie pe oră calculat depășește maximul definit de dispozitiv valoare (așa cum este definită în Tema 1) notifică în mod asincron utilizatorul pe interfața sa web.

3.3 Frontend

Se navigheaza in directorul unde este salvat frontendul si se ruleaza , intr-un terminal, comanda **npm install** pentru instalarea dependintelor Angular. Apoi se ruleaza comanda **ng serve** pentru a porni aplicatia. Se deschide localhost:4200 intr-un browser web. Aplicatia se deschide cu o pagina de login, in care utilizatorul trebuie sa isi

introduca credentialele(email si parola) apoi este redirectionat la pagina specifica atributiilor pe care le are (admin sau client).

3.4 Sensor Simulator

Pentru a porni o instata de Sensor Simulator se configureaza in deviceid.txt id-ul deviceului si intr-un terminal se ruleaza: **node smart-metering-device-simulator.ts deviceid.txt**. Daca se doreste pornirea in paralel a mai multor instante, se creaza alte fisiere de configurare de unde se citeste id-ul deviceului si se ruleaza ,transmitand in linia de comanda numele fisierului de unde se citeste. Cand valoarea citita depaseste valoarea maxima pentru device-ul respectiv, userul primeste o notificare(prin websocket). Simulatorul începe prin citirea identificatorului de dispozitiv dintr-un fișier specificat în linia de comandă.Se realizează conexiunea la serverul RabbitMQ folosind adresa și credențialele specifice.Datele de la senzor sunt citite dintr-un fișier CSV, simulate și trimise în coada RabbitMQ sub formă de mesaje JSON.Fiecare mesaj conține un timestamp, identificatorul dispozitivului și o valoare de măsurare.Mesajele sunt trimise la intervale regulate pentru a simula datele de senzor care sunt actualizate periodic.Conexiunea la RabbitMQ este închisă când simularea este finalizată.

3.5 Microserviciul de monitoring and communication

Responsabilități: Procesarea și stocarea datelor de măsurare primite de la coada RabbitMQ. Reacționarea la evenimente de dispozitiv primite prin coada RabbitMQ.

Funcționare: Backend-ul este construit folosind framework-ul Nest.js și este configurat pentru a asculta coada RabbitMQ specificată. Se definește un consumator de mesaje care procesează măsurările primite și le trimite pentru stocare. Un alt consumator gestionează evenimentele de dispozitiv și execută acțiunile corespunzătoare (creare, actualizare, ștergere, asocieri). Mesajele procesate sunt logate și trimise către serviciul de procesare și stocare a măsurătorilor. Mesajele de evenimente ale dispozitivelor sunt gestionate în funcție de acțiunea specificată și sunt logate în consecință. Orice eroare în procesare este gestionată și mesajul poate fi retrimis, respins sau reîncercat în funcție de necesități.

3.6 Microserviciul de chat

Microserviciul de chat facilitează comunicarea între utilizatori și administratorul sistemului, permițându-le să pună întrebări și să primească răspunsuri într-un mod eficient. Front-end-ul afișează o casetă de chat în care utilizatorii pot introduce mesaje. Utilizatorii pot interacționa cu caseta de chat pentru a trimite întrebări și a primi răspunsuri. Utilizatorii pot trimite mesaje către administrator printr-un proces asincron. Mesajele includ identificatorul utilizatorului pentru a identifica sursa mesajului. Administratorul primește mesajele utilizatorilor și are posibilitatea de a inița o sesiune de chat. Mesajele pot fi trimise și primite în ambele direcții pe durata unei sesiuni de chat. Administratorul poate gestiona conversații cu mai mulți utilizatori simultan. Fiecare chat este tratat ca o sesiune separată, permițând interacțiunea simultană cu diferiți utilizatori. Utilizatorii primesc notificări atunci când administratorul citește mesajul lor. Administratorul primește notificări atunci când utilizatorul citește mesajul său. Utilizatorii primesc notificări când administratorul își scrie mesajul și invers. Aceste notificări indică activitatea în timp real din partea celuilalt participant la chat. Am implementat un WebSocketGateway folosind Nest.js pentru a gestiona comunicația în timp real între server și client în cadrul aplicației de chat.

Decoratorul @WebSocketGateway: - se utilizează pentru a marca clasa ca fiind un gateway WebSocket. - opțiunea cors este setată pentru a permite conexiuni doar de la http://localhost:4200.

WebSocketServer: -Decorator care injectează instanța serverului WebSocket. Permite accesul la funcționalitățile serverului WebSocket pentru gestionarea conexiunilor și emiterea mesajelor.

OnGatewayConnection și OnGatewayDisconnect: -Implementarea acestor interfețe permite gestionarea evenimentelor de conectare și deconectare a clientului. handleConnection și handleDisconnect afișează informații în consolă atunci când un client se conectează sau se deconectează.

handleMessage: -Decoratorul @SubscribeMessage('sendMessage') este utilizat pentru a asculta mesajele cu tipul 'sendMessage'.Primește un mesaj de la client și îl prelucrează adăugând metadate (timestamp, starea de citire) înainte de a-l emite către toți clienții conectați sub forma 'newMessage'.

handleTyping și handleSeen: - Implementează funcționalitatea de a emite evenimente de tip 'typing' și 'seen' către toți clienții conectați. Similar cu handleMessage, aceste metode adaugă metadate și emit evenimentele corespunzătoare.

Toate mesajele sunt trimise către toți clienții conectați prin intermediul this.server.emit. Se folosește Socket pentru a accesa funcționalitățile WebSocket ale unui client specific.

Practic **flowul** ar fi urmatorul: cand un user vrea sa trimita un mesaj, il scrie in casuta si se apeleaza metoda sendMessage si adminii fac subscribe la acea metoda. Deci orice admin primeste mesajul, daca este conectat in acel moment. Primul care face subscribe, acela poate conversa cu userul. Cand userul sau adminul vrea sa scrie un mesaj, tasteaza in casuta respectiva, dar nu apasa inca pe butonul de trimitere , prin websocket se apeleaza metoda handleTyping iar destinatarul face subscribe la ea. Acelasi lucru se intampla si cand vrea sa fie mesajul citit, doar ca atunci trebuie dat click pe mesajul respectiv. Astfel,destinatarului, celui care face subscribe la handleSeen , ii apare seen by "...", unde "..." este numele celui care a apasat click pe mesaj. De asemenea, un admin poate conversa cu mai multi useri, apar mai multe casete de mesaje cu fiecare user in parte. Asta am facut-o astfel: am salvat local o lista de persoane:mesaje cu care admin converseaza. Initial lista e goala. Cand primeste un mesaj prin socket, se adauga in lista goala numele senderului si mesajul respectiv. Cand se mai primeste un nou mesaj de la acelasi user, nu se mai adauga dinou nume:mesaj, ci se cauta numele persoanei daca exista deja in lista si se concateneaza mesajul acolo.

3.7 Docker

Aplicația beneficiază de un fișier docker-compose.yml care grupează cele 8 servicii într-un singur container Docker. Acest container permite rularea aplicației fără a fi nevoie să porniți fiecare aplicație individual din mediul de dezvoltare (IDE).Acest fișier docker-compose.yml definește configurația containerelor Docker pentru fiecare serviciu din aplicație, inclusiv bazele de date, backend-urile pentru managementul utilizatorilor și dispozitivelor, precum și frontend-ul Angular. Fiecare serviciu este configurat pentru a utiliza imagini Docker, a expune porturi necesare și a asigura dependențe corecte între servicii.

3.8 Security

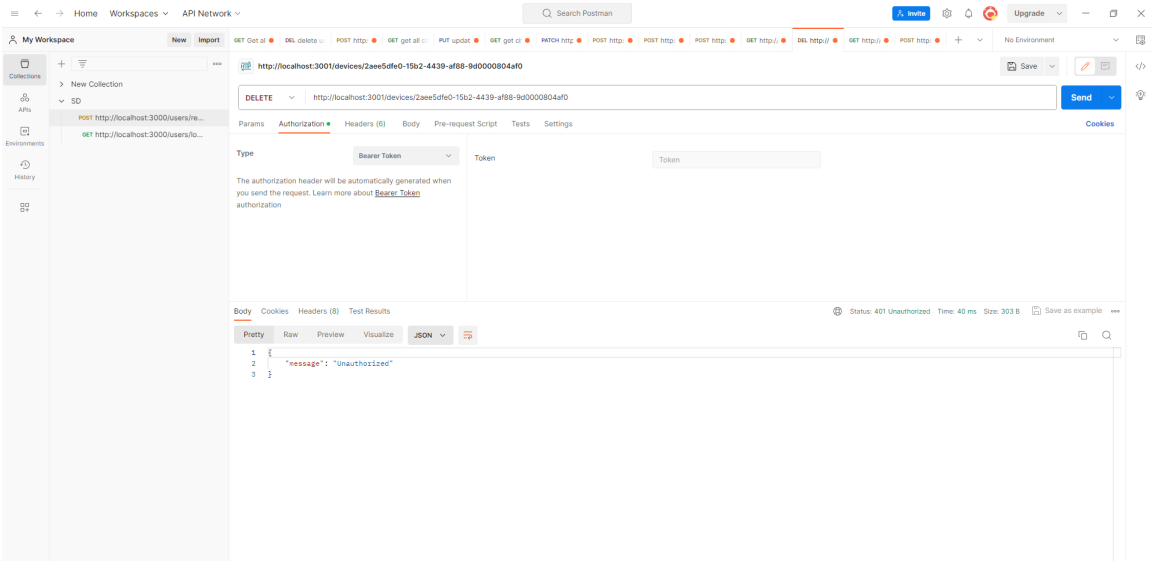
Pentru securitate, am ales ca microserviciul de user sa fie responsabil pentru autorizarea celorlalte. Am inceput prin autentificarea unui utilizator. Apoi l-am extins prin emiterea unui JWT. În cele din urmă, am creat o rută protejată care verifică un JWT valid la cerere. Am instalat dependintele necesare. Pentru a realiza acest lucru am folosit si libraria Passport. Passport este cea mai populară bibliotecă de autentificare node.js, binecunoscută de comunitate și folosită cu succes în multe aplicații de producție.

Dupa asta, am creat un Guard pentru autorizare(in functie de token-ul primit dupa autentificare- il decodific si vad daca e admin sau user), care ofera diferite atributii in functie rolul pe care userul il are.

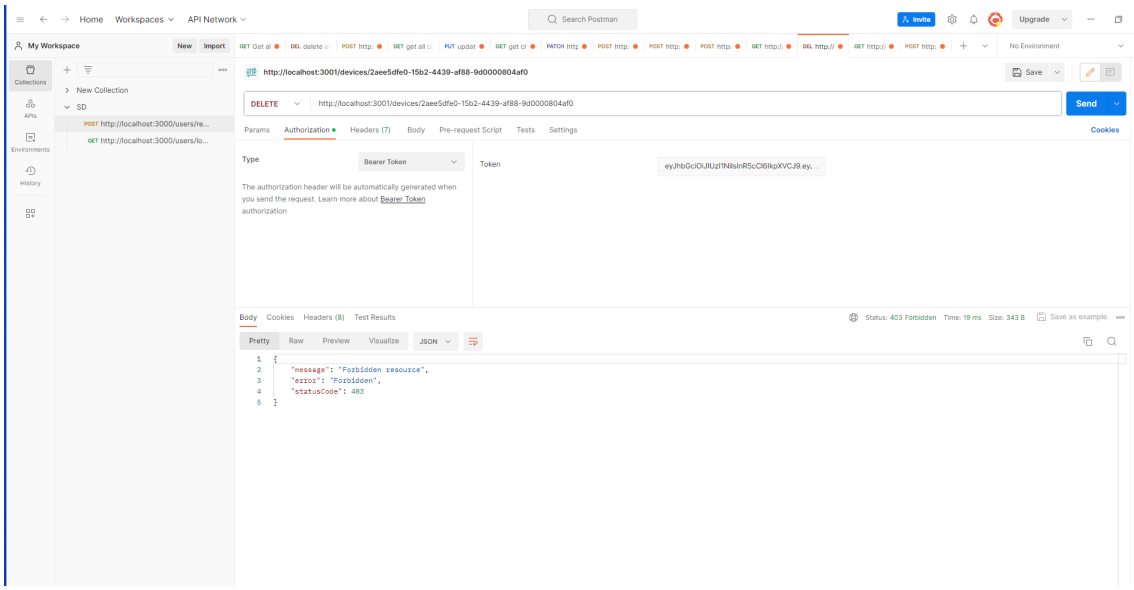
Acest micro-serviciu generează token de acces la aplicația client. Tokenii vor fi folosite pentru a accesa altele microservicii care au aceasi secretKey ca si serverul de autorizare.

Astfel vor fi 3 cazuri:

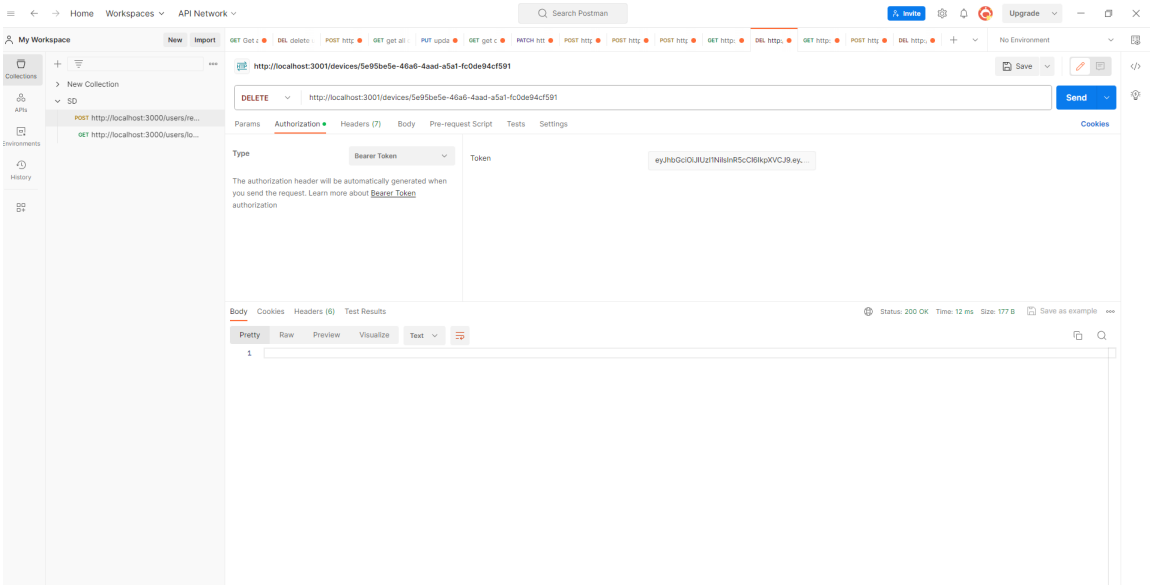
Primul se refera la cazul in care incerc sa fac un request fara a avea un user logat(fara a avea un token valid)



Al doilea caz se refera la momentul in care avem un user conectat(are un token valid) dar nu are autorizare. De exemplu aici, un user normal nu ar trebui sa aiba posibilitatea sa stearga device-uri din aplicatie.



Acesta este cazul de succes, in care este si conectat dar are si autorizare.



4 Bibliografie

https://blog.back4app.com/how-to-deploy-a-nest-js-application/

https://docs.nestjs.com/first-steps

https://dev.to/gustavocontreiras/how-to-create-a-dockerized-full-stack-environment-with-mysql-nes

https://angular.io/tutorial/tour-of-heroes/toh-pt0

https://customer.cloudamqp.com/instance

https://backendcommunity.com/how-to-integrate-socket-io-with-angular-and-nest-js/

https://https://jwt.io/

https://docs.nestjs.com/recipes/passport/

https://docs.nestjs.com/security/authentication/