Aplicație pentru Licitații - Piata Mica
versiunea 1.0

Costin Radu Ionut

Contents

1 Introducere

1.1 Descrierea temei

Proiectul are ca scop realizarea unei aplicatii web pentru licitatii si a documentatiei aferente, folosind drept inspiratie site-uri precum okazii.ro, publi24 etc. Aplicatia va permite postarea de produse si vizionarea/ editarea produselor postate de user in cazul in care licitatia nu a expirat si mai poate licita produsele altor utilizatori sau sa le caute, aceste actiuni pot fi facute in cazul unui client logat, iar un client nelogat va putea doar sa isi faca cont.

Aplicatia va fi realizata in React pe partea de frontend, Spring Boot pe partea de backend, iar datele userului si a produselor vor fi stocate intr-o baza de date PostgreSQL. De asemenea, partea de licitatie va stoca intr-un consumator Kafka userul, pretul licitat si data licitarii.

1.2 Nevoia identificata

Cu ajutorul aplicatiei date, oamenii pot licita pentru fructe si legume online din comfortul casei lor.

2 Scop & Obiective & Rezultate

2.1 Scop

- Aplicatia are ca scop licitarea de produse alimentara de orice natura in limite legale, persoanei logate care doreste sa cumpere.

2.2 Obiective

- 1 modalitate de inregistrare, logare si delogare sigura, bazata pe JWT token, accesabila printr-un meniu ce se gaseste pe prima pagina;
- 1 modalitate de filtrare produse in functie de categorie si pret, accesibil printr-un buton;
- 1 modalitate de cautare produse dupa nume/categorie;
- 1 metoda de a licita pentru un produs direct pe site;
- 1 metoda de a observa castigatorul si istoricul licitatiei;
- 1 metoda de a posta produse pe site;
- 1 interfata usor de folosit de catre clienti, axata pe culorile visiniu, roz si alb;
- 1 metoda de a vedea actualizarea produsul la fiecare 2 secunde;

- 1 modalitate de a cronometra data postarii produsului, pentru a expira licitatia dupa 2 zile;
- 1 sectiune prin in care se poate vizualiza in consola persoa suma si clipa in care un utilizator a licitat, accesibila numai din momentul in care userul intra pe pagina pe baza de Websockets si Kafka

2.3 Rezultate

- 1 pagina de inregistrare si una de logare care functioneaza pe baza de JWT si encripteaza parola;
- 1 input de tip number in care se insereaza pretul 1 sectiune cu checkbox din care se bifeaza categoria, maxim 1 odata;
- 1 camp de tip searchbar care filtreaza produsele dupa nume.
- 1 pagina in care utilizatorul poate sa introduca pretul pe care il doreste si se va actualiza in 2 secunde
- 1 sectiune pe pagina de licitaie unde apare emailul castigatorului ;
- 1 pagina in care apar preturile licitate pentru acel produs;
- 1 pagina pentru upload produse;
- 100% din actiunile pe care utilizatorul le poate face sunt sugestive si usor de inteles;
- 1 functie care odata la 2 secunde apeleaza o metoda ce actualizeaza datele preluate de pe backend;
- 1 functie care compara data inserarii produsului luata in milisecunde cu 2 zile afisate in milisecunde si seteaza licitatia pe expirat in cazul in care data (inserarii + 2 zile = data curenta)
- 1 sistem distibuit bazat pe eventimente care trimite mesaje folosind kafka la toti utilizatorii care acceseaza pagina de licitatii.

2.4 Cerintele proiectului

- Identificarea actorilor si a cazurilor de utilizare

Pe site vor fi 2 tipuri de utilizatori, adica 2 roluri

1. Utilizator nelogat;
2. Client logat;

- Cerinte functionale:

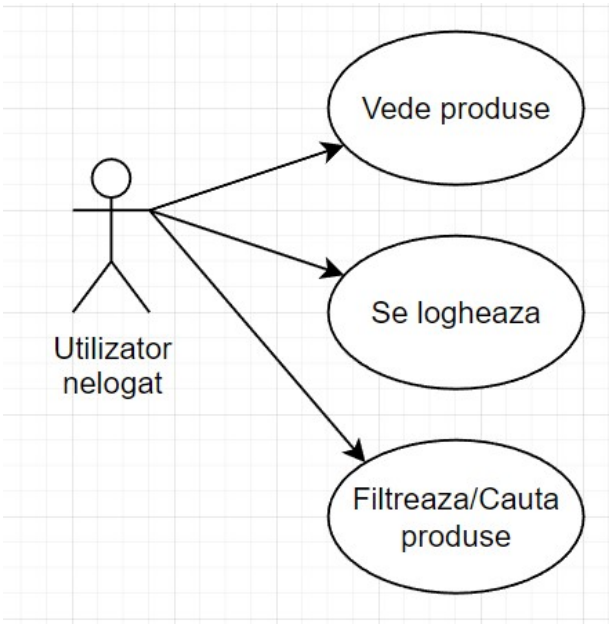


Figure 1: Clientul nelogat

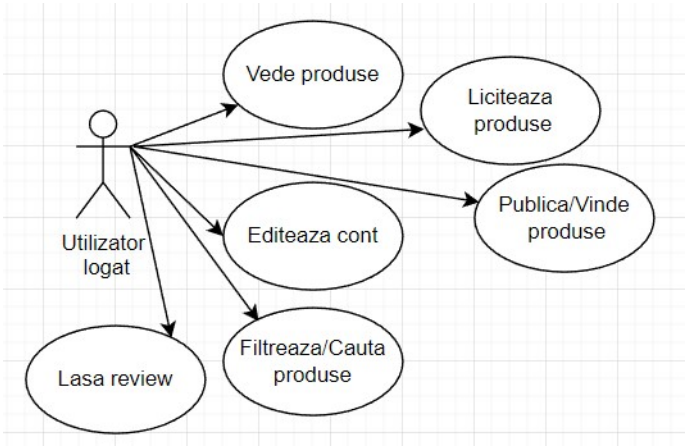


Figure 2: Clientul logat

- Aplicatia va actualiza pretul in timp real folosind un hook react;
- Aplicatia se va folosi un sistem distribuit de Kafka pentru a trimite mesaje in momentul expirarii licitatiei;
- In cazul unui update, aplicatia se va actualiza automat fara permisiunea utilizatorului, pentru a evita neplacerile unui pop-up si a mentine

siguranta si performanta aplicatiei

- Cerinte non-functionale
  - Utilizabilitate
    1. In interiorul aplicatiei va exista un link catre un ghid online de folosire pentru utilizatorii noi;
    2. Interfata se va axa pe nuante de visiniu, roz si alb.
  - Performanta
    1. Aplicatia va fi disponibila online pe tableta, laptop sau PC, in caz contrar interfata nu se va redimensiona;
    2. Aplicatia va avea o reactie de raspuns de maxim 6 secunde in cel mai rau caz posibil;
    3. Sistemul poate prelua date 24h/24 si va actualiza baza de date in timp real.
  - Fiabilitate
    1. Aplicatia va functiona cat timp comanda de Run este activa;
    2. In cazul unor intrari invalide, se va afisa mesaj de eroare de tip toast.
  - Implementare
    1. Baza de date: PostgreSQL;
    2. Back-end: Spring Boot: Java, Kafka
    3. Front-end: React Native, CSS, HTML;
    4. Design: Figma;
    5. Server: Apache Tomcat;
    6. Mediu de dezvoltare: InteliJ, Visual Studio Code, Command Prompt;
    7. Testare: Postman.

2.5 Analiza concurentei: Okazii.ro

Pros: Mai multe detalii despre produs si filtre. **Cons:** Interfata basic.

2.6 Arhitectura

Arhitectura REST, folosind Java pe partea de backend, React.Js pe partea de frontend si postgresql ca baza de date.

2.7 Plan de riscuri

Legenda:
P = Probabilitatea de apariție, I = Impact, RG= Risc Global

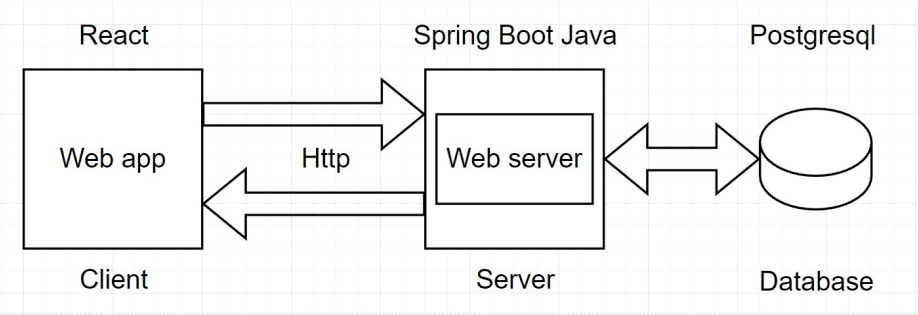


Figure 3: REST api

Activitate	Riscuri				Masuri	Mitigare
	Risc	P	I	RG		
Crearea interfetei	Interfata neatractiva	4	2	8	Desenez un mockup inainte	Reactualizez fisierul .CSS
Crearea sistemului distribuit	Sa nu primeasca/ trimita date	5	4	20	Fac o documentare ampla inainte	Refac folosind tutoriale usor de inteles
Implementarea bazei de date	Baza de date proiectata gresit	2	2	4	Se deseneaza initial arhitectura si se testeaza fiecare tabel	Se corecteaza tabel cu tabel
Implementarea comunicarii dintre front si back	Nu se trimit corect date	3	4	12	Se testeaza fiecare sectiune de cod noua	Se reface partea eronata
Implementarea sistemului de log-in	Metoda de hash nu este sigura	2	3	6	Se alege un cifru complex	Se schimba cifru
Prezentarea proiectului	Clientului (Profesorului) nu ii va placea	5	5	25	Se verifica de 2-3 ori prezentarea inainte	Se continua cu naturalete prezentarea

Table 1: Plan de riscuri

2.8 Design

Imaginile de mai au fost luate ca model pentru fiecare pagina din interfata cu utilizatorul.

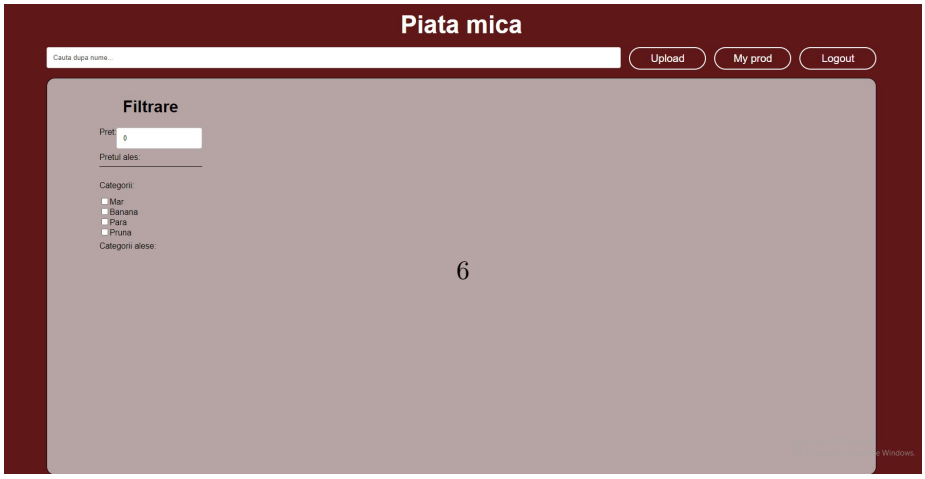


Figure 5: Interfata Client logat

2.9 Metodologia de implementare

Pentru implementarea proiectului a fost aleasa metodologia Agile.

Am ales durata fiecarui sprint (intre 1-2 saptamani/sprint), apoi am ales ce se va desfasura in fiecare sprint:

1. Primul sprint (2 saptamani): S-a realizat partea de introducere a documentatiei, s-a ales designe-ul si s-a creat circa 90% din interfata;

2. Al 2lea sprint (2 saptamani): S-a terminat implementarea bazei de date si a interfetei; s-au realizat metodele de GET (getAll si getById), POST, PUT, DELETE si s-au facut fetch pe endpoints.

3. Al 3lea sprint (2 saptamani): S-a realizat metoda de a urmari expirarea licitatiei, securitatea si sistemul de logare;

4. Al 4lea sprint (1 saptamana): S-a implementat sistemul distribuit pentru a trimite mesajele

3 Proiectare si detalii de implementare

3.1 Proiectarea bazei de date

Baza de date contine 3 tabele: products, users si istoric, legate intre ele de o cheie straina.

Baza de date a fost implementata conform Figurii 6 si a fost creata din consola folosind postgresql scripts.

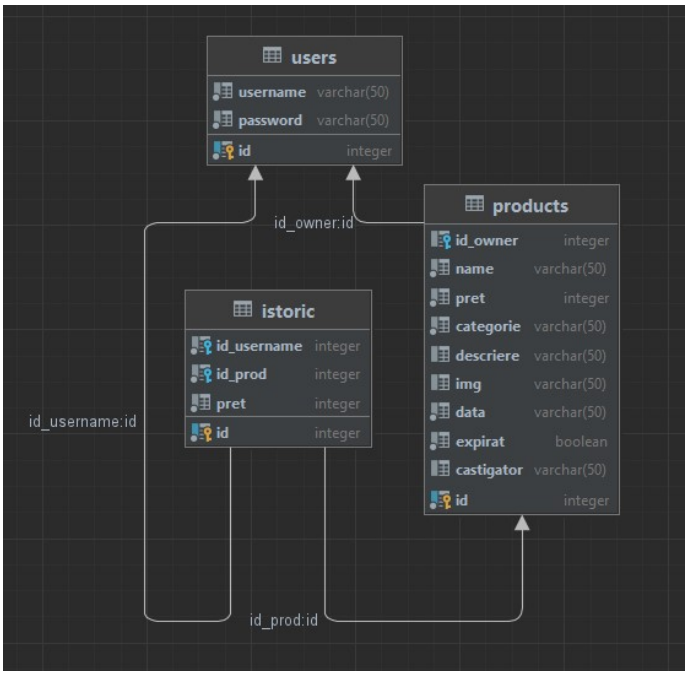


Figure 6: Proiectarea bazei de date

- Comanda pentru a crea tabela products

```
create table products
(
id serial
primary key,
id_owner integer
references users,
name varchar(50) not null,
pret integer not null,
categorie varchar(50) not null,
descriere varchar(50),
img varchar(50),
data varchar(50) not null,
expirat boolean default false not null,
castigator varchar(50)
);

alter table products
owner to postgres;
```

- Si comanda pentru a crea tabela user este

```
    create table users
(
id serial
primary key,
username varchar(50) not null,
password varchar(50) not null
);

    alter table users
owner to postgres;
```

- Similar se face pentru tabela Istoric.

3.2 Front

Proiectul de react a fost creat folosind comanda npx create-react-app front.

Folderul care contine implementara codului de react contine urmatoarele elemente:

- 1 element pentru home;
- 1 pagina pentru a licita produse;
- 1 pagina pentru logare si 1 pentru inregistrare
- 1 pagina pentru a incarca produse
- 1 pagina pentru a vedea produsele postate si 1 pentru a edita produsele;
- 1 pagina pentru a vedea istoricul licitatiilor.

Pentru inceput, am facut partea de Routare folosind ¡BrowserRouter¿ (Fig. 7) si am implementat fiecare functie pe care am exportat-o conform modelului din Fig8, de asemenea, navigarea intre paginile web se face folosind hookul useNavigate, sau ¡Link¿ in cazul in care doresc sa folosesc props. Pentru a accesa fiecare endpoint in parte am facut cate un fetch, exemplificat in Fig 9.

Partea de logare facuta de pe front reprezinta un post care returneaza un token care va fi stocat in spatiul de stocare al browserului folosind biblioteca jwtDecode si comanda localStorage.setItem(”token”, numeVariabila);

Pentru a impiedica persoanele neautorizate sa acceseze anumite pagini si pentru a afla identitatea userului, se verifica tokenul si daca nu este regasit in localStorage, userul este redirectionat pe pagina principala.

```
<BrowserRouter>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/Login" element={<Login />} />
      <Route path="/Register" element={<Register />} />
      <Route path="/Product/:id" element={<Product />} />
      <Route path="/Upload" element={<Upload />} />
      <Route path="/MyProducts" element={<MyProducts />} />
      <Route path="/EditProduct/:id" element={< EditProduct/>} />
    </Routes>
    <ToastContainer />
  </BrowserRouter>
```

Figure 7: Route

```
import React from "react";

const Home =()=>
{
  return (
    <>
    </>
  )
}

export default Home;
```

Figure 8: Model

```
const requestOptions = {
  method: "PUT",
  headers: {
    "Content-Type": "application/json",
    Authorization: "Bearer " + localStorage.getItem("token"),
  },
  body: JSON.stringify({ expirat: true }),
};
fetch(`http://localhost:8080/products/expirat/${id}`, requestOptions);
```

Figure 9: Fetch

Etapa de sistem distribuit foloseste SocketJsCLient din biblioteca react-stomp pentru a conecta aplicatia la WebSocket si pentru a primi mesajele listate si in consumatorul Kafka.

```
<SockJsClient
        url={SOCKET_URL}
        topics={["/topic/group"]}
        onConnect={onConnected}
        onDisconnect={console.log("Disconnected!")}
        onMessage={(msg) => onMessageReceived(msg)}
        debug={false}
      />
```

Figure 10: SocketJsCLien

Pentru a afisa mesajele primite de WebSocekt, am folosit functia onMessageReceived (Fig. 11), si pentru a trimite date, cea de onSendMessage (Fig. 12).

```
let onMessageReceived = (msg) => {
  console.log("New Message Received!!", msg);
  setSender(msg.sender);
  setIdLic(msg.id);
  toast.warning("userul " + sender + " a castigat licitatia cu id-ul" + idLic);
};
```

Figure 11: Primire de mesaje

```
let onSendMessage = (id, castigator, pret) => {
  const requestOptions1 = {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: "Bearer " + localStorage.getItem("token"),
    },
    body: JSON.stringify({
      id:id,
      sender: castigator,
      content: pret,
    }),
  };

  fetch(`http://localhost:8080/api/send`, requestOptions1)
    .then((response) => response.text())
    .then((data) => {
      console.log(data);
    });
};
```

Figure 12: Trimitere de mesaje

3.3 Back

Partea de Back este creata intr-un proiect SpringBoot

Am facut 1 interfata repository pentru user, unul pentru produse si unul pentru istoric, pentru a implementa metodele FindAll si findById (Fig13).

Figure 13: interfata Repository

Apoi am facut un DTO(Data transfer object) pentru fiecare tabela pentru a trimite datele si o clasa de tipul Entiry pentru a face operatii asupra bazei de date.

Dupa care s-au implementat 3 controlere, care gestioneaza metodele GET, POST, PUT, DELETE utilizate pentru a face actiuni asupra bazei de date folosind clasele de tip entity si datele primite din DTO, totdata ajutandu-ne de interfete.(Fig 14)

In partea de securitate m-am ocupat de politica CORS prin 2 clase, dupa care am facut o clasa pentru a genera si valida un token, una pentru filtrare token, una pentru encriptare parola, una pentru a permite autentificarea, una pentru a prelua datele despre user si starea lui.

Figure 14: Exemplu metoda din controller

3.4 Kafka

- Instalare si pornire Kafka si Zookeeper

Odata instalat kafka, se porneste zookeeper
$.\backslash bin\backslash windowszookeeper-server-start.bat.\backslash config\backslash zookeeper.properties$

Apoi se porneste serviciul de broker Kafka

$.\backslash bin \backslash windows \backslash kafka - server - start.bat. \backslash config \backslash server.properties$

Si se creaza un topic cu numele kafka-chat-3

$.\backslash bin \backslash windows \backslash kafka - topics.bat - -create - -topickafka - chat - 3 - -bootstrap - serverlocalhost : 9092 - -replication - factor1 - -partitions1$

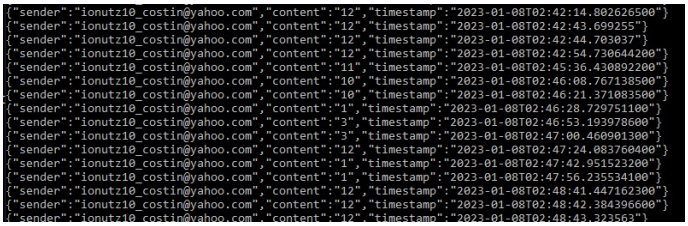
Pentru verificare, in fisierul bin-windows, intr-un terminal nou se da comanda pentru a lista topicele:

kafka-topics.bat –bootstrap-server=localhost:9092 –list

Mai apoi, se creaza un consumator:

$.\backslash bin\backslash windows\backslash kafka-console-consumer--bootstrap-serverlocalhost : 9092 - -topickafka - chat - 3$

Datele stocate in Kafka arata astfel



- Development

In aplicatia Java facuta precedent se creaza un model de trimitere mesaje, intr-o clasa "Message", in care se decide cum vor fi trimise mesajele spre consumator si utilizator (Fig 15).

In Partea de consumer, am implementat clasa "ListenerConfig.java", in care setez consumerul, cheia si valoarea pentru deserializare. in continuare, am creat clasa "MessageListener.java" care foloseste anotatia "@KafkaListener" pentru a trimite date spre stiva Kafka.

Websocketul utilizat pentru a trimite date spre client este reprezenat de metoda @PostMapping pentru a decide ce date se trimit, iar pentru a trimite date spre toti utilizatorii conectati se folosesc metodele "@MessageMapping("/sendMessage")" si "@SendTo("/topic/group")"

Apoi, am creat un Producer care trimite mesajele spre topicul de Kafka "ProducerConfiguration", (Fig 16) aici se declara beanul KafkaTemplate si se configureaza producerul (ProducerFactory) care stie sa creeze producers bazat pe configurarile pe care le setam din producerConfigurations() (port,cheie si valoare de deserializat)

Dupa care am facut consumatorul (Figura 17), similar cu Producerul.

In partea de websockets am setat securitatea (Fig 18) ca sa pot accesa de pe localhost:3000 folosind ambele metode de conectare la socket:

Dupa care am facut o metoda POST pentru a trimite mesaje in stiva Kafka si la interfata userului in functie de id si o metoda pentru a trimite la toti utilizatorii in functie de calea prin care se face etapa de subscribe.

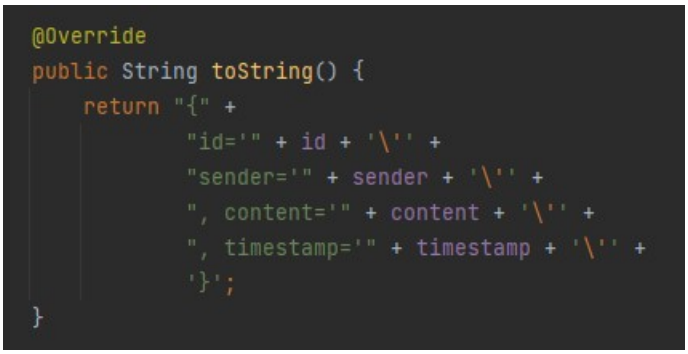


Figure 15: Modal mesaje

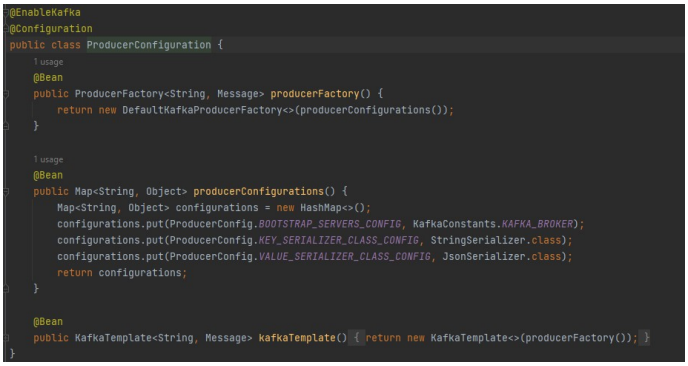


Figure 16: Configurare Producer

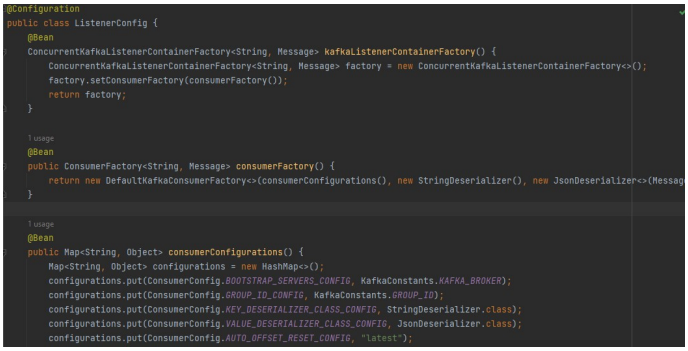


Figure 17: Configurare Consumator

4 Testare

Fiecare metoda a fost testata prin postman si apoi cu multiple date din front-end pentru a observa functionalitatea aplicatiei.

Figure 18: Securitate Websocket

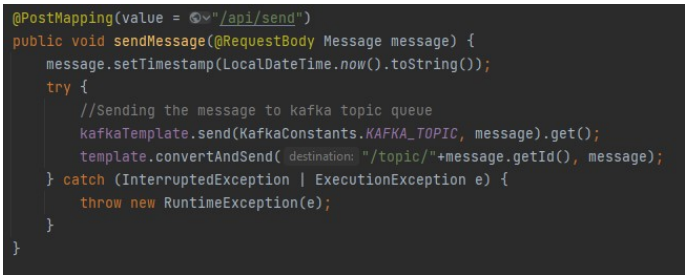


Figure 19: Securitate Websocket

5 Concluzii

In concluzie, proiectul prezinta o modalitate de a face o aplicatie pentru sustinerea licitatiilor pentru orice om care doreste sa cumpere fructe si legume.