UNIVERSITATEA TEHNICA
DIN CLUJ-NAPOCA
CENTRUL UNIVERSITAR NORD DIN BAIA MARE
FACULTATEA DE INGINERIE

DEPARTAMENTUL DE INGINERIE ELECTRICĂ, ELECTRONICĂ ȘI CALCULATOARE

Distributed IoT Monitoring System

Sistem distribuit pentru monitorizarea și procesarea datelor de senzori

Student: Hosu Răzvan

Baia Mare, 2026

Cuprins

Capitolul 1

Introducere

1.1 Contextul aplicațiilor IoT și al sistemelor distribuite

Sistemele IoT (Internet of Things) reprezintă un domeniu esențial al infrastructurilor informatice moderne, fiind alcătuite din dispozitive fizice interconectate — precum senzori, actuatori și microcontrolere — care colectează date din mediul înconjurător și le transmit către aplicații software pentru procesare și analiză. Aceste sisteme sunt, de regulă, distribuite geografic, funcționează autonom și generează fluxuri continue de date variate, cu o frecvență ridicată.

Pe măsură ce numărul dispozitivelor IoT crește, arhitecturile centralizate devin un punct unic de eșec și limitează performanța și disponibilitatea sistemului. Sistemele distribuite oferă mecanisme pentru scalare orizontală, separarea clară a responsabilităților și creșterea rezilienței în fața defecțiunilor. În acest context, proiectul de față se înscrie în paradigma sistemelor distribuite, având ca scop colectarea și procesarea în timp real a datelor provenite de la senzori distribuiți.

1.2 Necesitatea procesării distribuite a fluxurilor de date

Datele generate de dispozitivele IoT sunt continue, dinamice și pot conține evenimente critice care necesită reacții rapide. Aplicațiile IoT moderne impun procesarea fluxurilor de date imediat după generare, conform paradigmei *stream processing*, în care evenimentele sunt tratate pe măsură ce apar.

Procesarea centralizată a acestor fluxuri conduce la creșterea latenței, la supraîncărcarea resurselor și la apariția unui punct unic de eșec. În plus, extinderea sistemului prin adăugarea de noi senzori sau consumatori devine dificilă fără întreruperea funcționării. Din aceste motive, este necesară utilizarea unei arhitecturi distribuite, bazate pe componente independente.

În cadrul proiectului, această problemă este abordată prin utilizarea unui broker de

mesaje și a unui model de comunicare asincron. Protocolul MQTT, bazat pe modelul *publish–subscribe*, permite decuplarea producătorilor de date de consumatori. Această abordare este specifică sistemelor distribuite orientate pe evenimente (*event-driven systems*) și asigură o comunicare robustă și scalabilă.

1.3 Scopul proiectului

Scopul acestui proiect este realizarea unui sistem IoT distribuit care să permită colectarea, transmiterea și procesarea în timp real a datelor provenite de la senzori, cu afișarea rezultatelor într-o aplicație web. Sistemul este conceput pentru a evidenția principiile fundamentale ale sistemelor distribuite, precum comunicarea asincronă, decuplarea componentelor și procesarea evenimentelor în timp real.

Prin integrarea unui broker MQTT, a unui backend de procesare implementat în Java și a unui frontend web, proiectul demonstrează funcționarea unui flux complet de date într-un mediu distribuit. Utilizarea unui simulator de senzori permite testarea și validarea funcționalităților sistemului fără a depinde de hardware fizic.

1.4 Obiectivele principale ale sistemului

Obiectivele principale ale proiectului sunt:

- colectarea datelor de la noduri distribuite;
- transmiterea asincronă a mesajelor folosind protocolul MQTT;
- procesarea evenimentelor în timp real;
- afișarea rezultatelor într-o aplicație web.

1.5 Prioritizarea cerințelor folosind metoda MoSCoW

Pentru prioritizarea cerințelor funcționale și nefuncționale ale sistemului a fost utilizată metoda MoSCoW, care clasifică cerințele în funcție de importanța lor pentru funcționarea aplicației într-un mediu distribuit.

1.6 Cazuri de utilizare ale sistemului

Pentru a descrie funcționalitatea sistemului distribuit și interacțiunea dintre actorii principali, au fost definite mai multe cazuri de utilizare. Acestea evidențiază responsabilitățile fiecărei componente și modul în care datele sunt generate, procesate și consumate în cadrul arhitecturii IoT.

Must have	**Should have**	**Could have**	**Won't have**
Transmiterea datelor de la senzori către backend prin mecanism distribuit	Conectarea mai multor senzori simultan (scalare producători)	Persistența datelor procesate într-o bază de date	Consistență distribuită strictă între toate componentele
Comunicare asincronă bazată pe MQTT (*publish–subscribe*)	Separarea clară între colectare, procesare și afișare	Afișarea de statistici și agregări avansate	Replicare automată completă a backend-ului în această versiune
Procesare în timp real a evenimentelor (validare/filtrare/agregare)	Extinderea sistemului fără modificări majore de arhitectură	Mecanisme de alertare în timp real (threshold)	Mecanisme avansate de securitate distribuită (ex. PKI complet)
Simulare senzori pentru rulare fără hardware fizic	Gestionarea erorilor de comunicare și reconectare	Suport rapid pentru noi tipuri de senzori	Garanții formale de tip *exactly-once delivery*

Tabela 1.1: Prioritizarea cerințelor sistemului utilizând metoda MoSCoW

Primul actor analizat este sursa de date, reprezentată de senzorii IoT sau de un simulator de senzori. Aceștia au rolul de a genera și transmite periodic măsurători către infrastructura de mesagerie, fără a cunoaște consumatorii finali ai datelor.

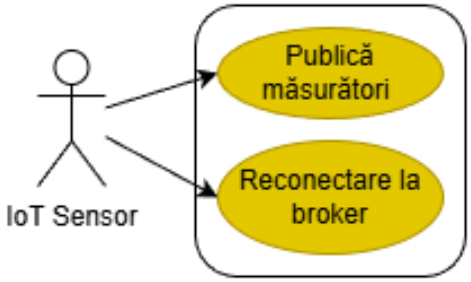

Figura 1.1: Diagramă de cazuri de utilizare pentru actorul *IoT Sensor*. Actorul publică măsurători către brokerul MQTT și gestionează reconectarea în cazul pierderii conexiunii.

Componenta centrală a sistemului este backend-ul de procesare, care face legătura între fluxul de date provenit de la senzori și aplicațiile client. Backend-ul se abonează la topic-urile MQTT, procesează mesajele recepționate și expune datele procesate prin endpoint-uri accesibile consumatorilor.

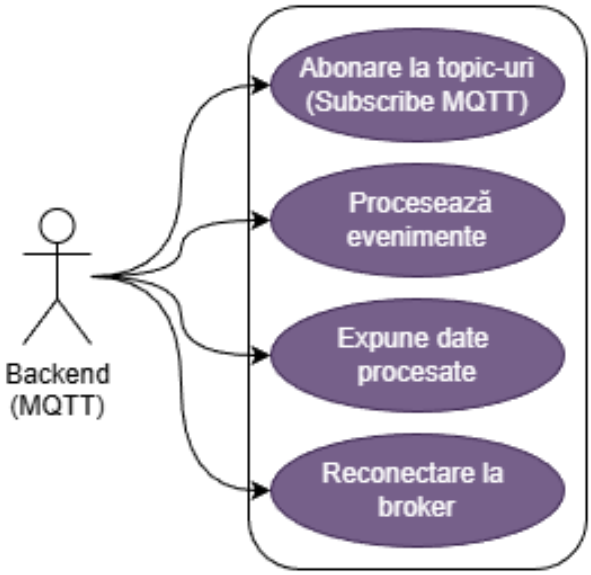

Figura 1.2: Diagramă de cazuri de utilizare pentru *Backend (MQTT)*. Backend-ul se abonează la topic-uri, procesează evenimentele recepționate, expune datele procesate și asigură reconectarea la broker.

Accesul la datele procesate este realizat prin intermediul aplicației web, care acționează ca un client al backend-ului. Frontend-ul este responsabil de inițierea conexiunii, solicitarea datelor și afișarea acestora într-o formă ușor de interpretat pentru utilizator.

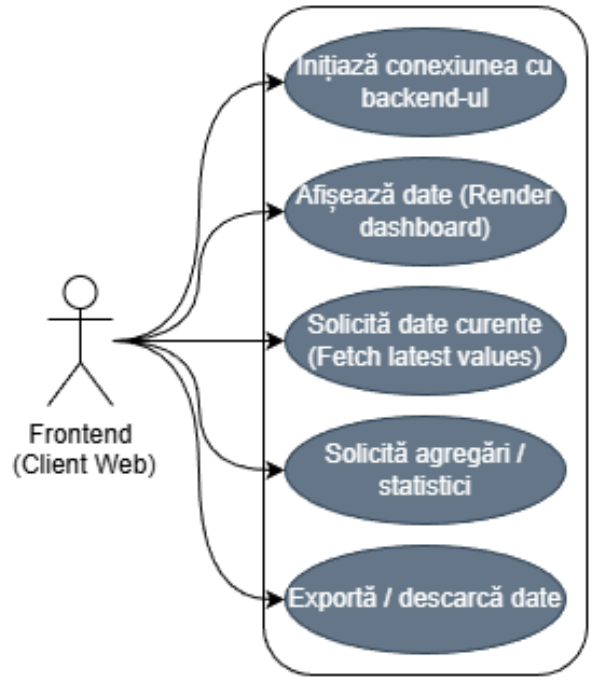

Figura 1.3: Diagramă de cazuri de utilizare pentru *Frontend (Client Web)*. Clientul inițiază conexiunea cu backend-ul, afișează datele într-un dashboard, solicită valori curente și agregări/statistici și permite exportul datelor.

Utilizatorul final reprezintă actorul care consumă informația furnizată de sistem. Acesta nu interacționează direct cu infrastructura de mesagerie sau cu backend-ul, ci utilizează interfața web pentru a monitoriza datele și starea generală a sistemului.

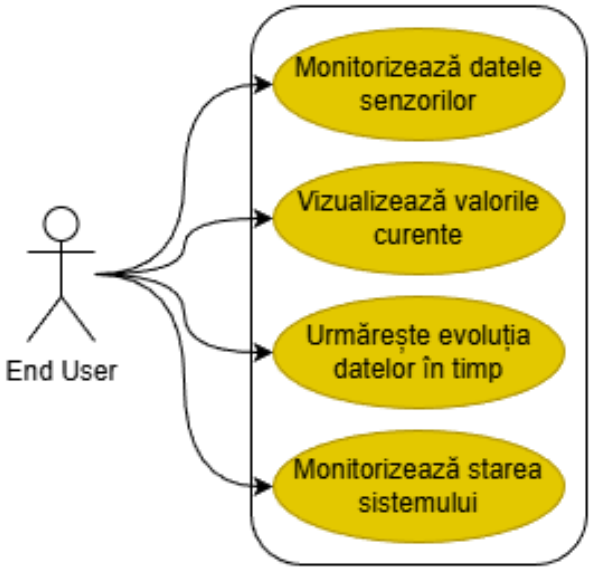

Figura 1.4: Diagramă de cazuri de utilizare pentru *End User*. Utilizatorul monitorizează valorile senzorilor, vizualizează datele curente, urmărește evoluția acestora în timp și verifică starea sistemului.

Capitolul 2

Arhitectura generală a sistemului

2.1 Componentele sistemului

Arhitectura sistemului este construită în jurul unui model distribuit bazat pe evenimente, în care componentele principale sunt decuplate și comunică prin mesaje transmise prin rețea. Această abordare reduce dependențele directe dintre module, permite evoluția independentă a fiecărei componente și facilitează scalarea orizontală în funcție de volum (număr de senzori, frecvența măsurătorilor) și de cerințe (număr de consumatori, complexitatea procesării).

Sistemul este alcătuit din următoarele componente principale:

2.1.1 Noduri IoT / Simulator de senzori

Componenta de tip *producer* este reprezentată fie de noduri IoT reale (de exemplu, microcontrolere precum ESP32), fie de un simulator de senzori utilizat pentru testare. Rolul acestei componente este de a genera periodic evenimente (măsurători) și de a le publica către brokerul MQTT.

Într-un scenariu tipic, un eveniment conține:

- un identificator al sursei (de exemplu, *sensorId* / *deviceId*);
- tipul măsurătorii (de exemplu temperatură, umiditate, curent);
- valoarea numerică și unitatea de măsură;
- un marcaj temporal (*timestamp*), necesar pentru ordonare logică și agregări.

Prin utilizarea unui simulator, sistemul poate fi evaluat în condiții controlate (de exemplu creșterea frecvenței de publicare, simularea mai multor senzori simultan, valori extreme), fără a depinde de disponibilitatea hardware-ului fizic. Acest aspect este relevant în contextul sistemelor distribuite, unde testarea sub sarcină și reproducibilitatea scenariilor sunt esențiale.

2.1.2 Broker de mesaje (MQTT)

Brokerul MQTT are rolul de intermediar (componentă *middleware*) între producători și consumatori. Alegerea MQTT este justificată de faptul că protocolul este proiectat pentru contexte IoT, având overhead redus și suport nativ pentru modelul *publish–subscribe*.

Brokerul oferă următoarele funcționalități esențiale:

- gestionează *topic*-urile și rutele de distribuție a mesajelor;
- permite publicarea asincronă de către producători, fără a cunoaște consumatorii;
- livrează mesajele către abonați, decuplând temporal și logic componentele;
- facilitează scalarea prin adăugarea de noi consumatori (abonați) fără modificări în partea de producere.

Din perspectiva sistemelor distribuite, brokerul realizează un mecanism de coordonare implicită: consumatorii se pot conecta sau deconecta independent, iar mesajele sunt propagate conform regulilor de abonare. În plus, utilizarea MQTT permite configurarea nivelurilor de calitate a serviciului (QoS) pentru controlul compromisului dintre latență și fiabilitate, în funcție de cerințe (de exemplu, telemetrie vs. alerte critice).

2.1.3 Backend (procesare și expunere servicii)

Backend-ul reprezintă componenta centrală de procesare (*stream processor*) și joacă rolul de consumator principal al evenimentelor publicate în broker. Acesta se abonează la topic-urile relevante și aplică logici de prelucrare în timp real. Operațiile tipice efectuate includ:

- **validare:** verificarea structurii mesajului și a tipurilor de date;
- **filtrare:** eliminarea mesajelor invalide/anormale sau a valorilor în afara domeniilor acceptate;
- **transformare:** normalizarea unităților sau maparea câmpurilor către un model intern;
- **agregare:** calcularea unor statistici pe ferestre de timp (de exemplu medie, minim, maxim).

După procesare, backend-ul expune rezultatele către partea de prezentare. Într-o arhitectură modernă, această expunere se realizează fie prin endpoint-uri HTTP (API), fie prin mecanisme *push* (de exemplu WebSocket), în funcție de nevoia de actualizare în timp real. Separarea consumului de evenimente (MQTT) de expunerea către UI contribuie la modularitate și la posibilitatea de a introduce consumatori suplimentari (de exemplu un serviciu de stocare sau un modul de alertare) fără a afecta fluxul existent.

2.1.4 Frontend (vizualizare și monitorizare)

Frontend-ul reprezintă componenta de consum pentru utilizatorul final, având rolul de a afișa într-o manieră intuitivă datele procesate. Din perspectiva arhitecturii distribuite, frontend-ul este un nod separat al sistemului, care comunică prin rețea cu backend-ul și nu accesează direct brokerul MQTT.

Funcționalitățile principale includ:

- afișarea stării curente a senzorilor (*live view*);
- vizualizarea evoluției valorilor în timp (de exemplu prin tabele sau grafice);
- prezentarea actualizărilor în timp aproape real, în funcție de mecanismul de transport ales (polling / WebSocket).

Separarea frontend-ului de backend are avantajul clar al dezvoltării independente și al posibilității de a înlocui interfața fără a afecta logica de procesare.

2.2 Fluxul de date între componente

Fluxul de date este orientat pe evenimente și urmează paradigma *publish–subscribe*. În această paradigmă, producătorii publică mesaje pe topic-uri, iar consumatorii se abonează la topic-uri fără a exista o legătură directă între cele două părți. Acest lucru oferă decuplare și flexibilitate, proprietăți esențiale în sisteme distribuite.

2.2.1 Descrierea fluxului operațional

Fluxul operațional poate fi descris în pașii următori:

1. **Generarea datelor:** nodurile IoT/simulatorul generează periodic măsurători și construiesc evenimente conform unui format stabilit (de exemplu JSON).
2. **Publicarea:** evenimentele sunt publicate către broker pe topic-uri corespunzătoare tipurilor de senzori sau dispozitivelor.
3. **Recepția și procesarea:** backend-ul, abonat la topic-urile relevante, primește mesajele imediat după publicare și aplică logica de procesare în timp real.
4. **Expunerea rezultatelor:** datele procesate sunt puse la dispoziție prin servicii accesibile frontend-ului (de exemplu API HTTP) și/sau canale de actualizare în timp real.
5. **Vizualizarea:** frontend-ul afișează valorile și actualizările către utilizator, oferind o perspectivă asupra stării sistemului și a evoluției măsurătorilor.

2.2.2 Proprietăți asigurate de fluxul bazat pe mesaje

Arhitectura propusă asigură o serie de proprietăți relevante pentru sisteme distribuite:

- **Decuplare logică și temporală:** producătorii și consumatorii nu trebuie să fie activați simultan și nu depind de implementarea celuilalt.
- **Comunicare asincronă:** publicarea și livrarea mesajelor se realizează fără blocarea componentelor, reducând latența percepută și crescând capacitatea de procesare.
- **Scalabilitate:** se pot adăuga noi senzori sau consumatori prin publicare/abonare la topic-uri, fără modificări structurale majore ale sistemului.
- **Reziliență la erori parțiale:** căderea unui consumator (de exemplu frontend) nu afectează publicarea datelor de către senzori; similar, componentele se pot reconecta independent.

2.2.3 Observații privind consistența și ordonarea evenimentelor

Într-un sistem distribuit bazat pe mesaje, ordonarea globală a evenimentelor nu este garantată în mod implicit, iar latențele pot varia. Prin urmare, includerea unui *timestamp* în evenimente și aplicarea unei logici de procesare care tolerează întârzieri (de exemplu ferestre de timp pentru agregări) reprezintă practici recomandate. În plus, alegerea nivelului *QoS* în MQTT influențează compromisul dintre performanță și fiabilitate, fiind o decizie arhitecturală care poate fi adaptată în funcție de criticitatea datelor.

Capitolul 3

Sistemul ca aplicație distribuită

Acest capitol analizează proiectul din perspectiva sistemelor distribuite, punând accent pe nodurile componente, tehnologiile utilizate pentru comunicare și procesare, precum și pe deciziile de proiectare care au condus la alegerea acestor tehnologii în defavoarea altora.

3.1 Identificarea nodurilor distribuite și rolul lor tehnologic

Sistemul implementat este alcătuit din mai multe noduri distribuite, fiecare rulând ca proces independent și comunicând exclusiv prin intermediul rețelei. Aceste noduri pot fi pornite, oprite sau mutate pe alte mașini fără a afecta funcționarea globală a sistemului, proprietate esențială pentru o aplicație distribuită.

3.1.1 Noduri IoT / Simulator de senzori

În cadrul proiectului, nodurile de tip producător sunt reprezentate de un simulator de senzori implementat sub forma unui script de tip shell (`simulate_sensors.sh`). Acest script generează periodic mesaje care simulează măsurători reale, precum temperatură sau umiditate, și le publică către brokerul MQTT.

Utilizarea unui simulator software, în locul exclusiv al unor senzori fizici, are mai multe avantaje:

- permite testarea sistemului fără dependență de hardware;
- facilitează generarea unui volum mare de evenimente într-un timp scurt;
- face posibilă reproducerea unor scenarii specifice (de exemplu creșterea bruscă a frecvenței de publicare).

Din punct de vedere al sistemelor distribuite, fiecare instanță a simulatorului reprezintă un nod autonom care publică evenimente fără a cunoaște existența altor noduri sau a consumatorilor, respectând principiul decuplării producător–consumator.

3.1.2 Brokerul de mesaje MQTT

Brokerul MQTT constituie componenta centrală de rutare a mesajelor și este utilizat ca mecanism de comunicare între noduri. Alegerea MQTT în acest proiect este motivată de caracteristicile sale tehnice, adaptate contextului IoT și sistemelor distribuite orientate pe evenimente.

Spre deosebire de o abordare bazată pe HTTP, unde senzorii ar trebui să trimită cereri directe către backend:

- MQTT reduce overhead-ul comunicației;
- permite comunicarea asincronă, fără a aștepta răspunsuri;
- suportă în mod nativ modelul *publish–subscribe*.

Brokerul gestionează topic-uri MQTT, iar mesajele publicate de simulator sunt livrate automat tuturor consumatorilor abonați. Această abordare permite adăugarea de noi consumatori (de exemplu un modul de stocare sau un sistem de alertare) fără a modifica producătorii de date, ceea ce reprezintă un avantaj major față de soluțiile bazate pe apeluri directe.

3.1.3 Backend-ul de procesare (Java / Spring Boot)

Backend-ul este implementat folosind platforma Java și framework-ul Spring Boot și reprezintă nodul principal de procesare a evenimentelor. Acesta se conectează la brokerul MQTT printr-un client dedicat și se abonează la topic-urile relevante.

Rolul backend-ului este multiplu:

- recepționează evenimentele publicate de senzori;
- validează structura mesajelor (format, tipuri de date);
- aplică logici de filtrare și agregare;
- expune datele procesate către alte componente prin API-uri.

Separarea backend-ului de producători este o decizie arhitecturală specifică sistemelor distribuite, deoarece permite:

- modificarea logicii de procesare fără impact asupra senzorilor;
- scalarea procesării prin rularea mai multor instanțe backend;
- introducerea de noi fluxuri de procesare în paralel.

Utilizarea Spring Boot facilitează dezvoltarea rapidă a serviciilor și integrarea componentelor necesare pentru procesare concurentă și expunerea interfețelor REST.

3.1.4 Frontend-ul ca nod consumator

Frontend-ul reprezintă un nod distinct al sistemului, responsabil exclusiv de prezentarea informațiilor către utilizator. Acesta nu comunică direct cu brokerul MQTT, ci interacționează cu backend-ul prin interfețe bine definite.

Această decizie elimină necesitatea ca frontend-ul să gestioneze direct complexitatea protocolului MQTT și mută responsabilitatea procesării și validării datelor către backend. Comparativ cu o soluție în care frontend-ul s-ar abona direct la broker, abordarea aleasă oferă un control mai bun asupra datelor expuse și reduce riscurile de securitate.

3.2 Modelul de comunicare și justificarea tehnologiilor

Modelul de comunicare al sistemului este complet asincron și bazat pe mesaje. Alegerea acestui model este justificată de natura aplicației, în care datele sunt generate continuu și nu necesită un răspuns imediat către producători.

Protocolul MQTT este utilizat pentru comunicarea între producători și backend, deoarece:

- este optimizat pentru transmiterea de evenimente frecvente;
- permite tolerarea pierderilor temporare de conexiune;
- suportă mai mulți consumatori simultan pentru același flux de date.

În comparație, utilizarea unui protocol sincron precum HTTP ar introduce dependențe temporale între componente și ar crește latența, fiind mai puțin potrivită pentru un sistem de tip stream processing.

3.3 Concepte de sisteme distribuite evidențiate prin implementare

Implementarea concretă a proiectului evidențiază mai multe concepte fundamentale din sistemele distribuite:

- **Decuplare:** senzorii nu cunosc backend-ul sau frontend-ul, ci doar brokerul MQTT.
- **Concurență:** backend-ul poate procesa simultan mesaje provenite de la mai mulți senzori.
- **Toleranță la erori:** oprirea temporară a backend-ului nu afectează publicarea mesajelor de către senzori.
- **Scalabilitate:** pot fi lansate mai multe instanțe de simulator sau backend fără modificări de cod.

Un exemplu concret de funcționalitate care evidențiază concepte specifice sistemelor distribuite este pagina *Distributed Insights*. Aceasta integrează date provenite de la mai multe noduri IoT și le corelează pe ferestre de timp aliniate, ilustrând efectele latenței, ale lipsei de sincronizare și beneficiile procesării distribuite.

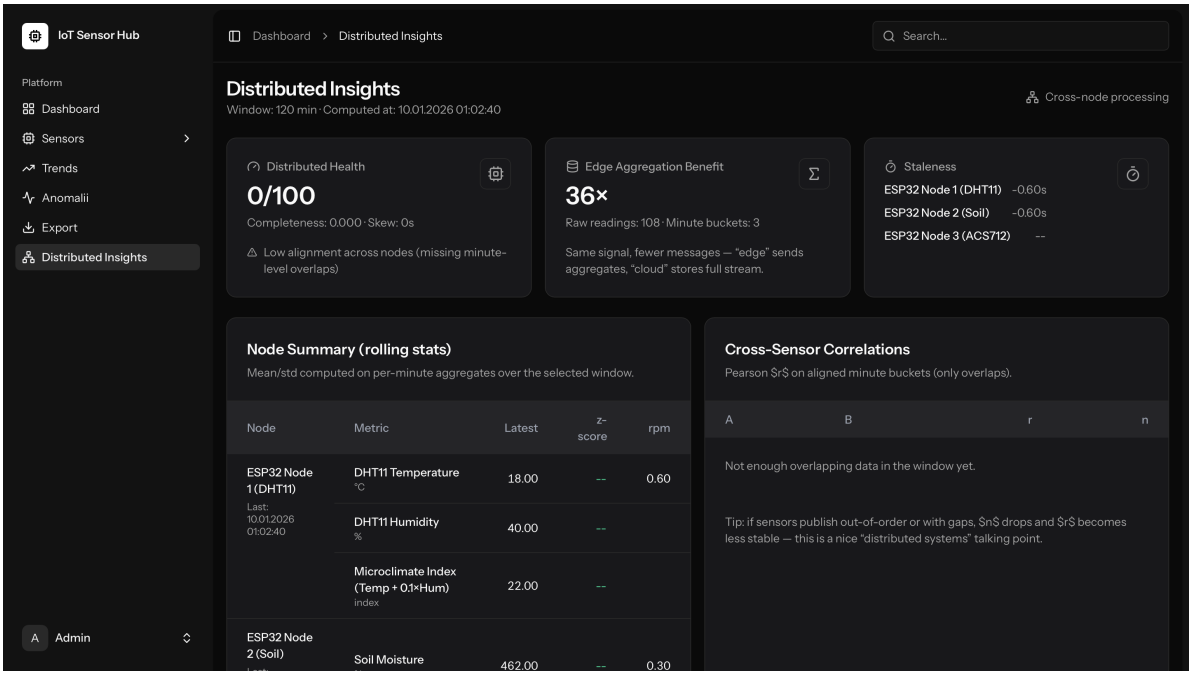

Figura 3.1: Interfață de analiză distribuită (*Distributed Insights*). Sunt ilustrate agregări pe ferestre de timp, corelații între noduri și metrici relevante pentru procesarea distribuită a fluxurilor de date.

3.4 Avantaje ale arhitecturii distribuite în contextul proiectului

Arhitectura distribuită aleasă oferă avantaje clare în raport cu o arhitectură monolitică:

- permite extinderea sistemului prin adăugarea de noi noduri;
- reduce impactul erorilor locale asupra sistemului global;
- susține procesarea în timp real a fluxurilor de date;
- facilitează testarea și dezvoltarea independentă a componentelor.

Prin utilizarea tehnologiilor MQTT, Java Spring Boot și a unui simulator de senzori, proiectul demonstrează în mod concret modul în care conceptele teoretice din sistemele distribuite pot fi aplicate într-un sistem IoT funcțional.

Capitolul 4

Tehnologii utilizate

Acest capitol descrie tehnologiile utilizate în cadrul proiectului și justifică alegerea acestora din perspectiva implementării unui sistem distribuit. Pentru fiecare componentă sunt prezentate rolul tehnologic, modul de utilizare în proiect și motivele pentru care tehnologia aleasă este potrivită în defavoarea altor alternative.

4.1 MQTT ca protocol pentru sisteme distribuite

În cadrul proiectului este utilizat un broker MQTT (de exemplu Eclipse Mosquitto) care acționează ca punct central de comunicare între nodurile distribuite. Senzorii sau simulatorul publică mesaje pe topic-uri MQTT, iar backend-ul se abonează la aceste topic-uri pentru a recepționa datele.

Topic-urile sunt organizate logic, de regulă pe tipuri de senzori sau pe identificatori de noduri, permițând o rutare flexibilă și extensibilă a mesajelor.

Modelul de comunicare implementat este *publish–subscribe*:

- producătorii (senzori sau simulator) publică mesaje către broker;
- brokerul primește mesajele și le distribuie către consumatori;
- consumatorii (backend-ul) primesc datele fără a cunoaște producătorii.

Mesajele transportă evenimente sub formă de date structurate (de exemplu JSON), care conțin informații precum tipul senzorului, valoarea măsurată și momentul generării.

Protocolul MQTT este potrivit pentru acest proiect din mai multe motive:

- comunicarea este asincronă, evitând blocarea nodurilor;
- overhead-ul este redus comparativ cu protocoale precum HTTP;

- permite conectarea unui număr mare de dispozitive simultan;
- suportă mecanisme de reconectare și livrare fiabilă.

În comparație cu o soluție bazată pe cereri HTTP directe către backend, MQTT reduce cuplarea între componente și este mai eficient pentru transmiterea frecventă a evenimentelor în sisteme IoT distribuite.

4.2 Backend Java pentru procesarea evenimentelor

4.2.1 Tehnologia utilizată

Backend-ul este implementat în limbajul Java, folosind un framework de tip Spring Boot. Această aplicație rulează ca un serviciu independent și este conectată la brokerul MQTT printr-un client dedicat.

Backend-ul acționează ca nod de procesare a fluxurilor de evenimente și reprezintă componenta centrală de analiză a datelor în sistem.

4.2.2 Utilizarea în cadrul proiectului

Fluxul de procesare în backend este următorul:

1. abonarea la topic-urile MQTT relevante;
2. recepția mesajelor publicate de senzori;
3. validarea structurii și a tipurilor de date;
4. filtrarea valorilor invalide sau anormale;
5. agregarea datelor (de exemplu pe ferestre de timp);
6. expunerea rezultatelor către frontend.

Backend-ul produce date procesate, precum valori agregate sau stări ale senzorilor, care sunt consumate de interfața web.

4.2.3 Justificarea alegerii Java

Utilizarea Java este justificată prin:

- suportul solid pentru procesare concurentă;
- stabilitatea și maturitatea platformei;
- ușurința extinderii cu noi reguli de procesare;

- integrarea facilă cu protocoale și servicii externe.

Comparativ cu soluții mai simple (de exemplu scripturi sau aplicații monolitice), backend-ul Java permite gestionarea unui volum mai mare de evenimente și evoluția sistemului către arhitecturi mai complexe.

4.3 Frontend web ca nod de consum

Frontend-ul este implementat sub forma unei aplicații web care consumă datele procesate de backend. Acesta rulează ca nod separat și comunică exclusiv prin interfețe bine definite.

Componentă	**Tehnologie**	**Rol în sistemul distribuit**
Noduri IoT / Simulator	Script Bash / senzori	Generare de evenimente și publicare MQTT
Broker de mesaje	MQTT	Intermediere publish–subscribe
Backend	Java (Spring Boot)	Procesare, validare și agregare evenimente
Frontend	Aplicație web	Consum și vizualizare date procesate

Tabela 4.1: Sinteză a tehnologiilor utilizate și a rolurilor acestora

4.4 Reprezentare grafică a arhitecturii

Frontend-ul primește datele de la backend prin intermediul unui API sau al unui mecanism de actualizare în timp real (de exemplu WebSocket). Aplicația afișează:

- starea curentă a senzorilor;
- valorile măsurate și procesate;
- evoluția datelor în timp, sub formă de tabele sau grafice.

Actualizarea datelor se realizează periodic sau în timp real, în funcție de mecanismul de comunicare ales.

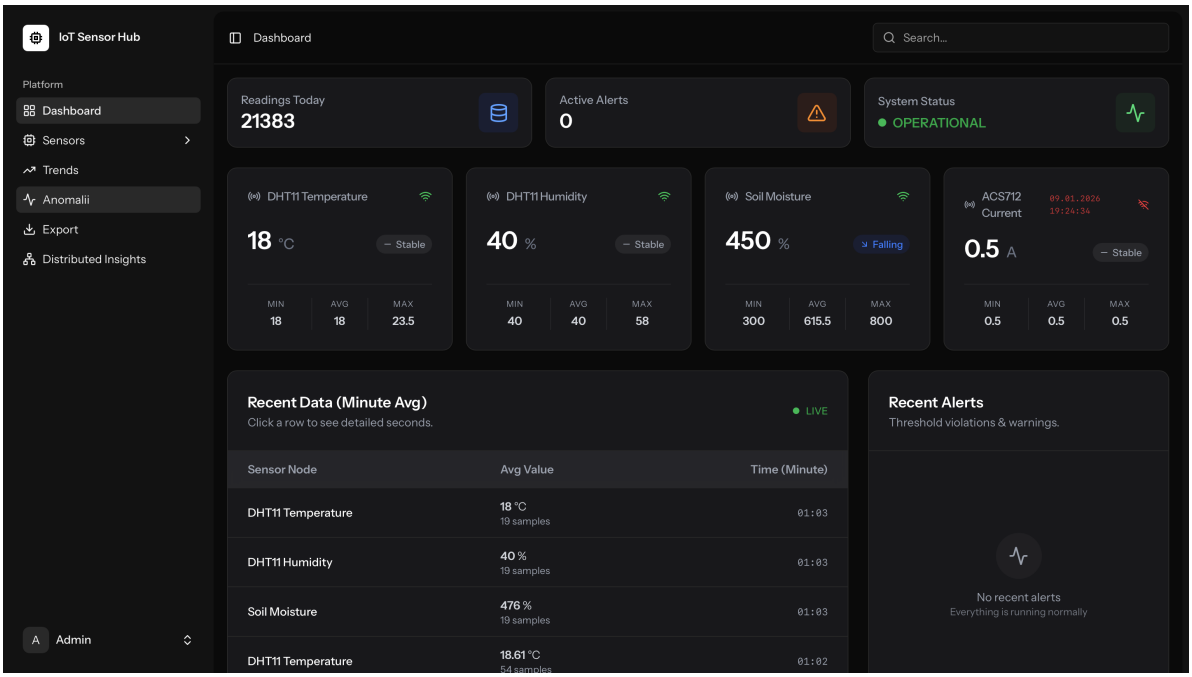

Figura 4.1: Interfața principală a aplicației web (Dashboard). Sunt afișate valorile curente ale senzorilor, starea sistemului și sumarul evenimentelor recente, oferind utilizatorului o imagine de ansamblu asupra funcționării sistemului IoT distribuit.

Alegerea unei aplicații web ca interfață de consum este potrivită deoarece:

- oferă acces rapid și multiplatformă utilizatorilor;
- permite vizualizarea centralizată a stării sistemului;
- separă clar prezentarea de logica de procesare.

În contextul sistemelor distribuite, frontend-ul este un consumator pasiv, ceea ce reduce complexitatea și crește securitatea.

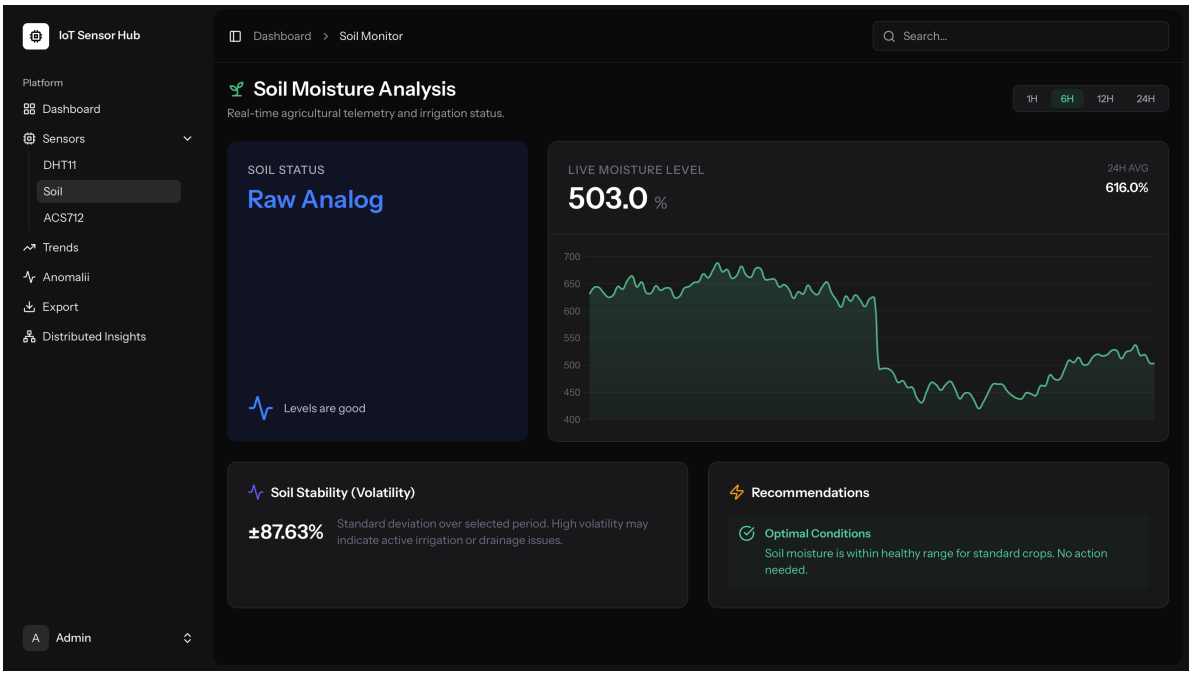

Figura 4.2: Pagină de detaliu pentru monitorizarea unui senzor (Soil Moisture). Sunt afișate valorile în timp real, istoricul măsurătorilor, indicatori de stabilitate și recomandări generate pe baza datelor procesate.

4.5 Script de simulare ca generator distribuit de evenimente

Pentru simularea nodurilor IoT este utilizat un script de tip Bash (sau echivalent), care generează evenimente artificiale și le publică către brokerul MQTT. Scriptul poate fi configurat pentru a simula unul sau mai mulți senzori.

Scriptul de simulare permite configurarea:

- numărului de senzori simulați;
- frecvenței de trimitere a mesajelor;
- tipurilor de date generate.

Fiecare instanță a scriptului acționează ca un nod distribuit independent, similar unui senzor real.

Utilizarea unui simulator este avantajoasă deoarece:

- permite testarea sistemului fără hardware fizic;
- facilitează generarea unui volum mare de date;
- reproduce condiții de încărcare specifice sistemelor distribuite.

Pentru a oferi o imagine de ansamblu clară asupra arhitecturii sistemului distribuit, sunt utilizate reprezentări grafice care evidențiază atât relațiile funcționale dintre componente, cât și distribuția acestora pe noduri și tehnologii. Aceste diagrame completează descrierea textuală și facilitează înțelegerea fluxului de date și a responsabilităților fiecărei componente.

Prima reprezentare este diagrama de componente, care ilustrează modulele software principale ale sistemului și interacțiunile dintre acestea. Diagrama evidențiază sursele de date (senzorii fizici și simulatorul), brokerul MQTT ca element central de intermediere, backend-ul responsabil de procesarea și persistența datelor, precum și aplicația frontend utilizată pentru vizualizare. Fluxul de date este realizat conform modelului *publish–subscribe*, iar accesul utilizatorului final la informație se face prin intermediul unui API REST.

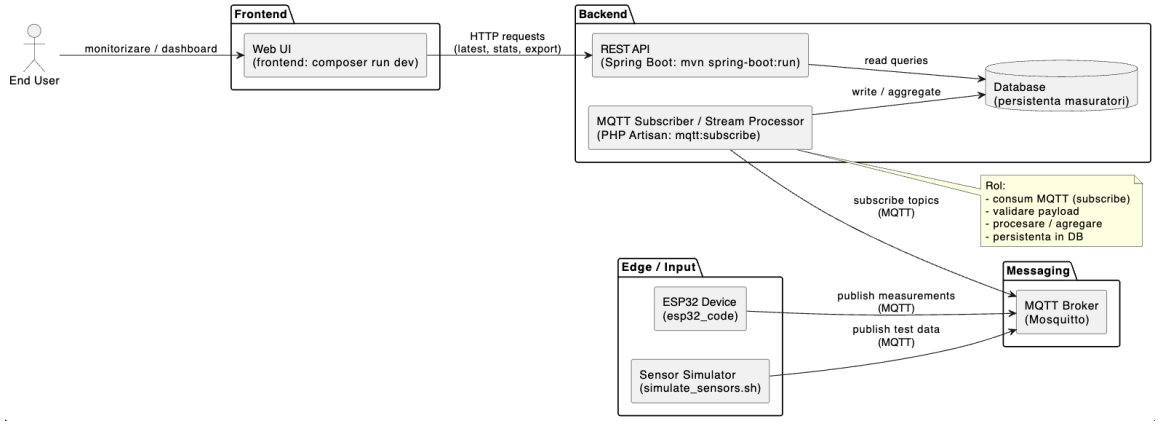

Figura 4.3: Diagramă de componente a sistemului. Senzorii și simulatorul publică date prin MQTT, backend-ul procesează și persistă informația, iar frontend-ul consumă datele prin API REST.

Pentru a evidenția modul de distribuire a componentelor pe infrastructură, este utilizată diagrama de deployment. Aceasta prezintă nodurile implicate în sistem (dispozitiv IoT, mașină de dezvoltare/testare, server/host și client), precum și tehnologiile utilizate pe fiecare nod. Diagrama subliniază separarea clară între componentele de tip edge, serviciile backend, infrastructura de mesagerie și interfața utilizatorului, precum și protocoalele de comunicație utilizate (MQTT și HTTP).

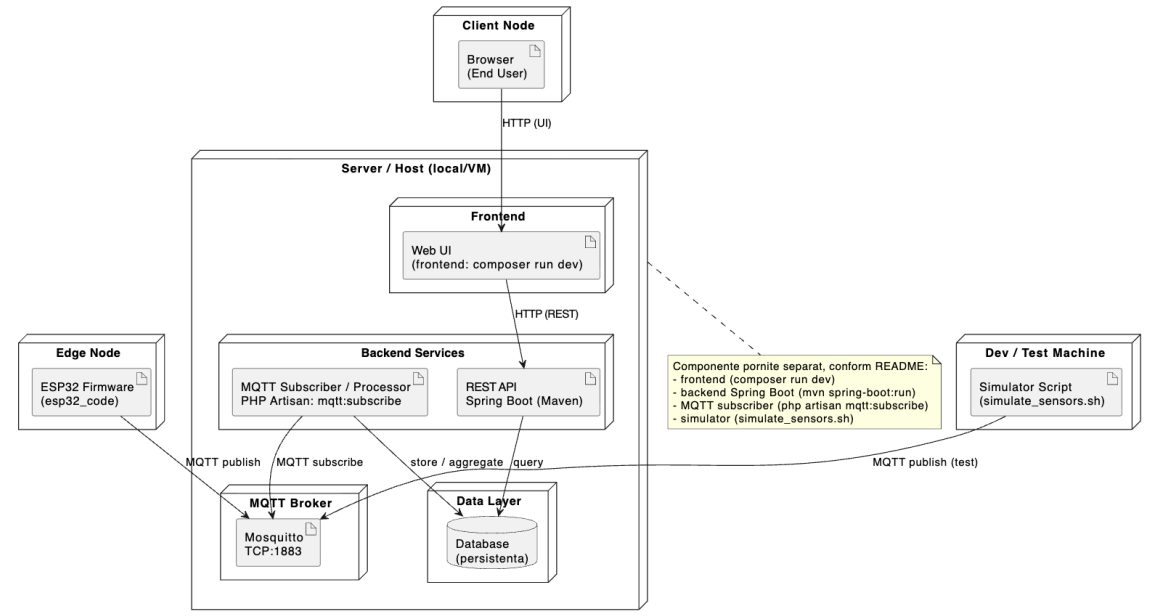

Figura 4.4: Diagramă de deployment a sistemului distribuit. Sunt ilustrate nodurile fizice/logice, componentele rulate pe fiecare nod și protocoalele de comunicație utilizate.

4.6 Modelul de date al aplicației

Pentru a descrie structura datelor gestionate de sistem și relațiile dintre entitățile principale, a fost definit un model conceptual de tip Entity–Relationship (ER). Acest model evidențiază modul în care măsurătorile sunt asociate surselor (senzorilor), precum și existența unor structuri auxiliare pentru agregări și evenimente de sistem, corespunzătoare procesării în timp real și monitorizării funcționării aplicației.

Diagrama ER de mai jos prezintă entitățile principale (*Sensor*, *Measurement*, *Aggregation*), împreună cu entități suport pentru organizarea topic-urilor MQTT și jurnalizarea evenimentelor. Relațiile sunt reprezentate prin săgeți orientate, reflectând legăturile de tip cheie străină către cheie primară.

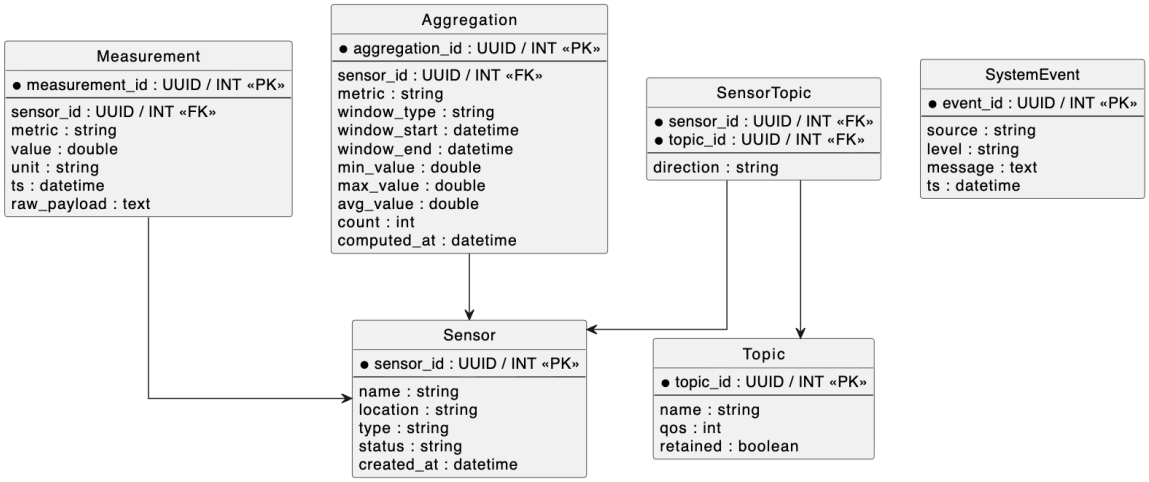

Figura 4.5: Diagramă ER pentru modelul de date al aplicației IoT: senzori, măsurători, agregări și entități suport.

Capitolul 5

Implementarea sistemului

Acest capitol descrie modul concret în care sistemul este implementat pe baza arhitecturii prezentate anterior. Sunt detaliate structura repository-ului și principalele funcționalități ale componentelor software: backend, frontend și simulatorul de senzori. Accentul este pus pe separarea clară a responsabilităților și pe rularea independentă a fiecărui nod distribuit.

5.1 Structura repository-ului

Repository-ul proiectului este organizat pe componente, fiecare reflectând un nod distinct al sistemului distribuit. Această structurare facilitează dezvoltarea, testarea și rularea independentă a fiecărei componente.

Structura principală a repository-ului este următoarea:

- `java-backend/` – conține serviciul backend responsabil de procesarea evenimentelor provenite de la senzori;
- `frontend/` – conține aplicația web utilizată pentru afișarea și monitorizarea datelor;
- `esp32_code/` – codul destinat nodurilor IoT reale, utilizat în cazul în care sunt disponibile dispozitive hardware;
- `simulate_sensors.sh` – scriptul de simulare care generează evenimente artificiale și le publică în brokerul MQTT.

Această organizare reflectă principiile sistemelor distribuite, în care fiecare componentă poate fi rulată ca proces separat, pe aceeași mașină sau pe mașini diferite, fără dependențe directe de implementare între ele.

5.2 Implementarea backend-ului

Backend-ul reprezintă componenta centrală de procesare a datelor și este implementat ca o aplicație Java independentă. Acesta rulează ca serviciu de sine stătător și comunică cu

celelalte componente exclusiv prin rețea.

5.2.1 Conectarea la brokerul MQTT

La pornire, backend-ul se conectează la brokerul MQTT folosind parametri de configurare precum adresa brokerului, portul și eventualele credențiale. Conectarea este realizată printr-un client MQTT integrat în aplicație, care permite gestionarea automată a reconectării în cazul întreruperilor de rețea.

5.2.2 Abonarea la topic-uri

După stabilirea conexiunii, backend-ul se abonează la topic-urile relevante pentru tipurile de senzori simulate sau reale. Abonarea permite recepția asincronă a mesajelor, fără a necesita apeluri explicite către producători.

Această abordare este specifică sistemelor distribuite bazate pe evenimente și permite adăugarea de noi surse de date fără modificări în codul backend-ului.

5.2.3 Procesarea mesajelor

La recepția unui mesaj MQTT, backend-ul aplică următorii pași:

- parsarea mesajului și extragerea câmpurilor relevante;
- validarea structurii și a tipurilor de date;
- filtrarea valorilor invalide sau anormale;
- aplicarea logicii de procesare (de exemplu agregări sau transformări).

Procesarea este realizată imediat după recepție, conform modelului de procesare a fluxurilor de date în timp real (*stream processing*). Backend-ul poate gestiona simultan mesaje provenite de la mai mulți senzori, beneficiind de mecanismele de concurență oferite de platforma Java.

5.2.4 Expunerea rezultatelor către frontend

După procesare, rezultatele sunt puse la dispoziția frontend-ului prin interfețe dedicate, cum ar fi endpoint-uri HTTP sau mecanisme de actualizare în timp real. Backend-ul joacă astfel rolul de intermediar între fluxul brut de date și interfața de prezentare.

Separarea acestei responsabilități permite introducerea ulterioară a altor consumatori, precum module de stocare sau sisteme de alertare, fără a modifica fluxul existent.

5.3 Fluxul de navigare și rutarea interfeței

Pentru a descrie structura interfeței web și modul de navigare între paginile aplicației, a fost realizată o diagramă de flux a rutelor frontend. Aceasta evidențiază separarea dintre rutele publice și cele protejate prin mecanisme de autentificare, precum și relațiile dintre paginile principale ale aplicației.

Diagrama surprinde traseele tipice de navigare ale utilizatorului, pornind de la paginile publice și continuând cu accesul la funcționalitățile disponibile după autentificare. Pentru fiecare pagină sunt indicate apelurile HTTP către backend, ilustrând integrarea dintre interfața web și serviciile distribuite ale sistemului.

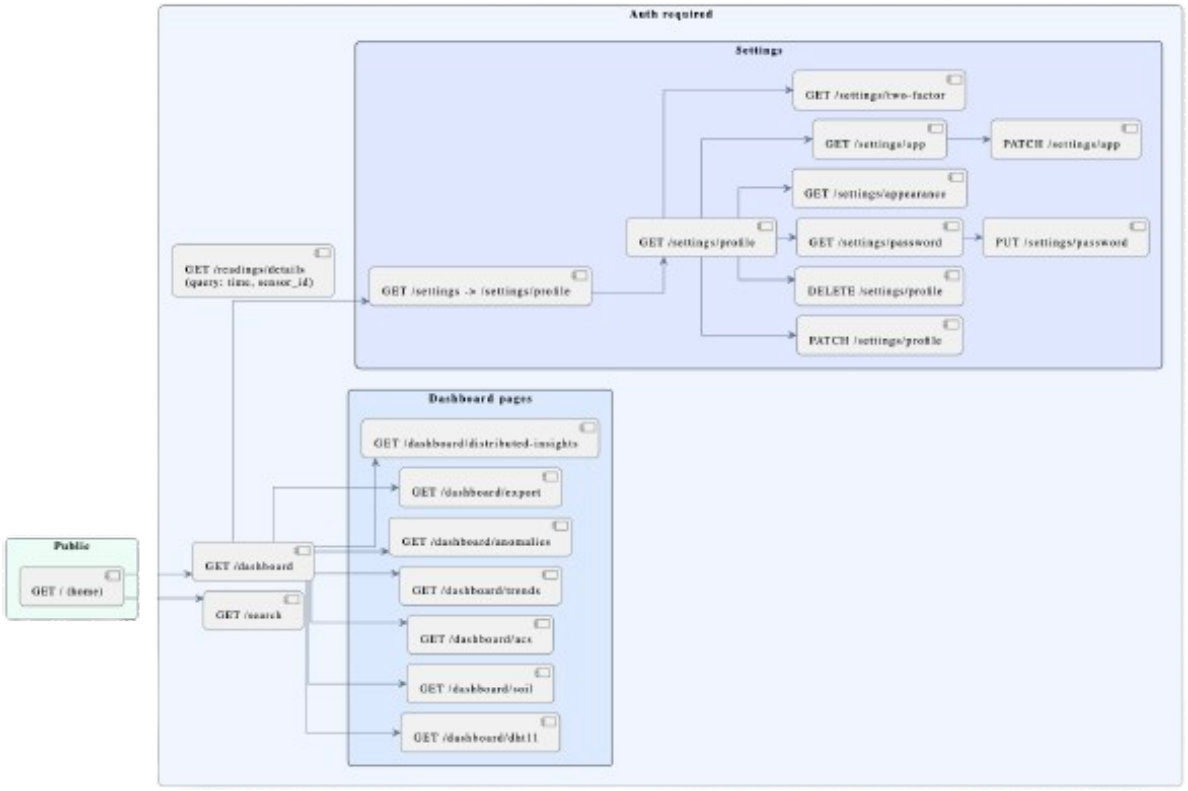

Figura 5.1: Diagramă de flux a rutelor interfeței web. Sunt evidențiate rutele publice, rutele protejate prin autentificare și apelurile către endpoint-urile backend asociate fiecărei pagini.

5.4 Implementarea frontend-ului

Frontend-ul este implementat ca o aplicație web independentă, care consumă datele procesate de backend și le afișează utilizatorului final. Această componentă nu interacționează direct cu brokerul MQTT și nu gestionează logica de procesare a datelor.

5.4.1 Consumarea datelor

Frontend-ul obține datele prin apeluri către backend, fie sub formă de cereri periodice, fie prin mecanisme de actualizare în timp real. Datele primite sunt deja validate și prelucrate, ceea ce reduce complexitatea aplicației de prezentare.

5.4.2 Funcționalități de vizualizare

Principalele funcționalități oferite de frontend includ:

- afișarea valorilor curente ale senzorilor;
- actualizarea automată a informațiilor;
- prezentarea datelor sub formă de elemente vizuale ușor de interpretat (de exemplu tabele sau grafice).

Frontend-ul reprezintă un nod consumator al sistemului distribuit, fiind complet decuplat de sursa directă a datelor și de mecanismele de comunicare MQTT.

5.5 Simularea fluxului de date

Pentru testarea și validarea sistemului în absența hardware-ului fizic, proiectul utilizează un mecanism de simulare a senzorilor.

5.5.1 Scriptul de simulare

Scriptul `simulate_sensors.sh` este utilizat pentru a genera evenimente artificiale care simulează măsurători reale. Scriptul publică periodic mesaje către brokerul MQTT, respectând același format utilizat de senzorii reali.

Parametrii de simulare pot include:

- numărul de senzori simulați;
- frecvența de trimitere a mesajelor;
- tipurile de date generate.

5.5.2 Rolul simulării în sistemul distribuit

Din punct de vedere al arhitecturii, fiecare instanță a scriptului de simulare acționează ca un nod distribuit independent. Acest lucru permite:

- testarea fluxului complet de date;
- evaluarea comportamentului sistemului sub sarcină;

- demonstrarea funcționării aplicației fără dispozitive IoT reale.

Simulatorul este util atât în etapa de dezvoltare, cât și în cadrul demonstrațiilor sau evaluărilor academice, oferind un control ridicat asupra condițiilor de testare.

Capitolul 6

Concluzii și direcții viitoare

6.1 Concluzii

Proiectul realizat demonstrează implementarea unui sistem IoT distribuit, în care datele sunt generate de noduri independente, transmise printr-un broker de mesaje și procesate în timp real înainte de a fi afișate într-o aplicație web. Arhitectura aleasă evidențiază în mod clar principiile fundamentale ale sistemelor distribuite, precum comunicarea asincronă, decuplarea logică dintre componente și rularea independentă a nodurilor.

Utilizarea protocolului MQTT permite un model de comunicare eficient și scalabil, adecvat pentru aplicații IoT caracterizate prin fluxuri continue de date. Separarea clară între producători (senzori sau simulator), componenta de procesare (backend) și consumatorul final (frontend) contribuie la modularitatea sistemului și facilitează extinderea acestuia fără modificări structurale majore.

Prin integrarea unui backend de procesare implementat în Java și a unui frontend web pentru vizualizare, proiectul demonstrează un flux complet de date într-un mediu distribuit. De asemenea, utilizarea unui simulator de senzori permite testarea și validarea sistemului fără a depinde de hardware real, aspect important în context academic și de dezvoltare.

În ansamblu, proiectul ilustrează modul în care conceptele teoretice din domeniul sistemelor distribuite pot fi aplicate practic într-o aplicație IoT funcțională și extensibilă.

6.2 Limitări actuale

Deși sistemul este funcțional și îndeplinește obiectivele propuse, există anumite limitări inerente implementării curente:

- procesarea evenimentelor este realizată de o singură instanță de backend activă, ceea ce limitează disponibilitatea în cazul unei defecțiuni;
- datele procesate nu sunt persistate într-un sistem de stocare distribuit, fiind utilizate doar pentru afișare în timp real;

- toleranța la erori este asigurată în principal prin mecanismele de reconectare oferite de protocolul MQTT, fără replicare explicită a componentelor critice.

Aceste limitări sunt acceptabile în contextul unui proiect academic, dar evidențiază direcții clare pentru îmbunătățiri ulterioare.

6.3 Direcții viitoare

Pe baza arhitecturii actuale, pot fi identificate mai multe direcții de dezvoltare care ar îmbunătăți caracteristicile sistemului din perspectiva sistemelor distribuite:

- **Replicarea backend-ului** – rularea mai multor instanțe de backend pentru a crește disponibilitatea și a elimina punctele unice de eșec;
- **Load balancing** – distribuirea fluxurilor de date între mai multe instanțe de procesare pentru creșterea performanței;
- **Persistență distribuită** – integrarea unei baze de date distribuite pentru stocarea istoricului măsurătorilor;
- **Mecanisme avansate de toleranță la erori** – replicarea brokerului MQTT sau utilizarea unor soluții de failover pentru componentele critice;
- **Extinderea funcționalităților de analiză** – introducerea unor mecanisme de alertare și analiză avansată a datelor în timp real.

Implementarea acestor direcții ar transforma sistemul într-o platformă IoT distribuită mai robustă și mai apropiată de soluțiile utilizate în medii de producție.

Bibliografie

[1] MQTT Official Documentation. *MQTT – The Standard for IoT Messaging.* `https://mqtt.org/`

[2] Eclipse Foundation. *Eclipse Mosquitto – An Open Source MQTT Broker.* `https://mosquitto.org/`

[3] Spring Framework. *Spring Boot Reference Documentation.* `https://spring.io/projects/spring-boot`

[4] Tanenbaum, A. S., Van Steen, M. *Distributed Systems: Principles and Paradigms.* Pearson Education, 2nd Edition, 2007.

[5] Kleppmann, M. *Designing Data-Intensive Applications.* O'Reilly Media, 2017.

[6] Copernicus12. *SDI IoT Sensor Stream Processor – GitHub Repository.* `https://github.com/Copernicus12/SDI_IoT_Sensor_Stream_Processor`