



UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI INFORMATICĂ



BAZE DE DATE ȘI TEHNOLOGII SOFTWARE

Lucrare de Disertație

SOLUȚIE CLOUD PENTRU MONITORIZAREA CASEI INTELIGENTE

Absolvent
CHIRIGIU DELIA MARIA

Coordonator științific
Lect. dr. Letiția Marin

București, iunie 2022

Rezumat

Scopul acestei lucrări este de a proiecta o soluție modulară, extensibilă și ușor de folosit pentru o casă inteligentă. În dezvoltarea acesteia, sunt îmbinate atât concepte *Internet of Things* prin arhitectura distribuită în noduri și *gateway*, cât și de Inteligență Artificială pentru a procesa datele captate și de a produce diferite perspective pe care o persoană le-ar fi omis. De asemenea, soluția dezvoltată este *cloud native*, susținând decuplarea diverselor componente și oferind un sistem definit de reziliență. Datele sunt transmise de către noduri folosind protocolul MQTT către un *broker* găzduit pe *gateway*, păstrând transmisia datelor în Intranet. Mai departe, acestea sunt trimise către microservicii în internet folosind HTTPS.

Abstract

The aim of this paper is to develop a modular, extensible, and easy to use solution for monitoring a smart home. During its development, Internet of Things concepts have been used by having a distributed architecture featuring several nodes and a gateway, but also Artificial Intelligence concepts for processing the captured data to obtain a different perspective which a person could have omitted. Also, the solution is cloud native, embracing decoupling of the components and offering a system defined by resilience. The data is sent from the nodes using the MQTT protocol to a broker hosted on the gateway, transmitting the data in Intranet. Next in their transmission, they are sent to the microservices in internet using HTTPS.

Cuprins

I. Introducere	1
II. Concepte de natură tehnică.....	3
II.1. <i>Internet of Things</i>	3
II.2. <i>Publish/Subscribe</i>	4
II.3. MQTT	5
II.3.1 Protocolul MQTT	5
II.3.2 Protocolul <i>WebSocket</i>	5
II.3.2 MQTT <i>Broker</i>	5
II.3.2 MQTT <i>Client</i>	6
II.4. Docker, Containerizare și Izolare	6
II.5. <i>Machine Learning</i>	8
III. Descrierea soluției	10
III.1. Arhitectura soluției	10
III.2. Lista de componente	13
III.3. Descrierea nodului IoT	15
III.4. Descrierea <i>Gateway</i> -ului	18
III.5. Microservicii dezvoltate	21
III.5.1. Microserviciul <i>Cloud Gateway</i>	22
III.5.2. Microserviciul <i>Config Server</i>	25
III.5.3. Microserviciul <i>Service Discovery</i>	25
III.5.4. Microserviciul OLTP	25
III.5.5. Microserviciul <i>Data Warehouse</i>	29
III.5.6. Microserviciul ML	32
III.5.7. Microserviciul de interfață grafică.....	35
Concluzii	40
Bibliografie.....	42

I. Introducere

Conectivitatea a devenit omniprezentă în viața modernă, transformând monitorizarea și controlul de la distanță într-un lucru facil, îmbunătățind calitatea vieții prin creșterea calității aerului sau prin automatizarea diferitelor activități casnice.

Potrivit Cisco [1], numărul de conexiuni *machine-to-machine* va crește până la 50% în 2023 față de anul 2018 când aceasta înregistra un procent de 34%, ceea ce înseamnă o schimbare semnificativă a cotei de piață datorită popularizării *Internet of Things*. Aceste obiecte inteligente pot lua forma unui senzor de temperatură și umiditate conectat la Internet, a unei camere de monitorizare a copilului, până la forma electrocasnicelor inteligente precum mașina de spălat, aerul condiționat sau robotul aspirator.

Scopul acestei lucrări este de a proiecta o soluție modulară, extensibilă și ușor de folosit pentru o casă inteligentă. În dezvoltarea acesteia, s-au îmbinat atât concepte *Internet of Things* prin arhitectura distribuită în noduri și *gateway*, cât și de inteligență artificială pentru a procesa datele captate și a produce diferite perspective pe care o persoană le-ar fi omis. De asemenea, soluția dezvoltată este *cloud native*, susținând decuplarea diverselor componente și oferind un sistem definit de reziliență. Datele sunt transmise de către noduri folosind protocolul MQTT către un broker găzduit pe *gateway*, păstrând transmisia datelor în Intranet. La nivel de *gateway* rulează și o aplicație dezvoltată în NodeJS ce este abonată la *topicul* ce conține date de la senzori și le va scoate în Internet prin intermediul HTTPS, asigurând un transport securizat al încărcăturii. Destinația acesteia este un microserviciu *web-facing* găzduit într-o instanță Cloud, care constituie rolul de dispecer spre diferitele microservicii internet responsabile. Printre ele se află și microserviciul ce oferă persistența datelor prin operațiuni de tip CRUD, acestea fiind salvate într-o bază de date Oracle Express Edition 21c. Alte microservicii notabile sunt cele de *Data Warehouse* și cel de inteligență artificială, decuplând astfel stocarea în rapoarte de procesarea datelor folosind algoritmi de învățare automată.

Structura lucrării este păstrată simplă pentru brevităte, conținând 3 capitole:

Capitolul II, intitulat „Concepte de natură tehnică”, prezintă succint conceptele necesare realizării lucrării, precum concepte de bază *Internet of Things*, o descriere a protocolului MQTT și modelul folosit de acesta, *Publish-Subscribe*. Alte concepte atinse sunt și cele specifice de containerizare și izolare a proceselor, dar și de inteligență artificială.

Capitolul III, intitulat „Descrierea soluției”, prezintă arhitectura concepută și fluxul datelor. De asemenea, capitolul oferă mai multe detalii despre nodurile IoT, *gateway*, microserviciile dezvoltate, interfața grafică disponibilă utilizatorului, cât și algoritmi de inteligență artificială folosiți.

Concluziile includ un rezumat al acestei soluții, contribuțiile personale, precum și lucruri de avut în vedere pentru dezvoltări viitoare.

II. Concepte de natură tehnică

II.1. *Internet of Things*

Conceptul de *Internet of Things* reprezintă modul prin care obiectele fizice încorporate cu senzori și *software* sunt folosite pentru a colecta și distribui prin internet date din mediul înconjurător către alte dispozitive, pentru a putea fi apoi prezentate și analizate.

Într-un ecosistem IoT vom găsi un dispozitiv *smart* ce are capacități de a colecta și analiza local datele sau de a le transmite la rândul său către o altă aplicație.

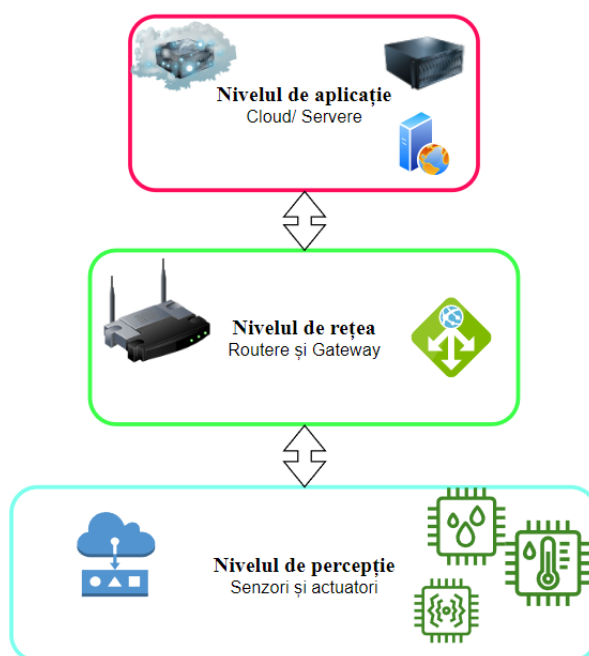


Figura 2.1. Arhitectura IoT

Arhitectura IoT este formată din trei niveluri de bază:

- Nivelul de percepție – este primul nivel ce integrează obiectele fizice: senzorii sau actuarii. Obiectivul acestuia este de a colecta informația asociată și de a o transmite către nivelul superior.
- Nivelul rețea (de transmisie) – este nivelul intermediar destinat *Gateway*-ului pentru a procesa și transmite datele colectate de la nodurile IoT. Acesta face legătura între senzori și server.
- Nivelul de aplicație – este ultimul nivel și poate fi reprezentat prin mai multe servere sau un sistem Cloud unde datele pot fi afișate, analizate și agregate.

II.2. Publish/Subscribe

Modelul *Publish-Subscribe* este un mod de comunicare asincron serviciu către serviciu [2]. Acesta este un *design pattern* ce oferă o cale de comunicare între *publishers (host)* și *subscribers* prin mesaje (evenimente).

Fiecare mesaj are asociat un *Topic*. Clientul *Publisher* trimite mesaje cu un anumit *Topic*, iar Clientul *Subscriber* abonat la acel *topic* va primi mesajele.

La baza acestui model de arhitectură stă un intermediar numit *broker* de mesaje cunoscut atât de *publisher* cât și de *subscriber*. Acesta primește mesajele publicate și le trimite către *subscriber*-ii ce s-au abonat topicului, păstrând anonimă identitatea fiecărei componente.

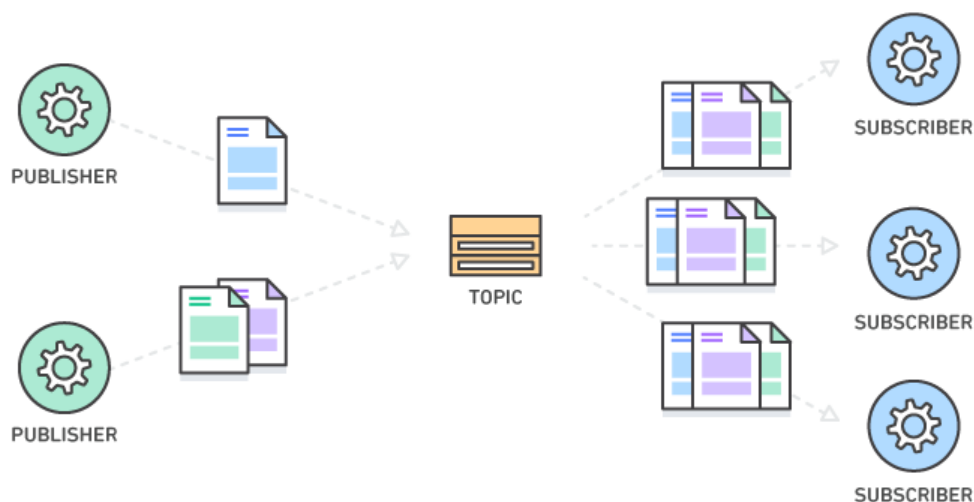


Figura 2.2. Modelul Publish/Subscribe

Avantajele acestui model sunt:

- Decuplarea între componente – permite izolarea completă a componentelor, făcând sistemul ușor de menținut.
- Elasticitatea soluției – o soluție ce integrează acest model de mesagerie este elastic deoarece nu depinde de numărul de *publisher*-i sau *subscriber*-i abonați.
- Sporește receptivitatea – fiind o soluție asincronă, transferul de mesaje nu blochează *publisher*-ul, iar *subscriber*-ul este ocupat doar atunci când se abonează unui *topic*.

II.3. MQTT

II.3.1 Protocolul MQTT

MQTT (*Message Queuing Telemetry Transport*) este un protocol *lightweight* folosit în arhitecturile de tip *publish/subscribe* și destinat comunicării *machine to machine* în medii cu lățime de bandă redusă [3].

Acesta este un protocol de mesagerie în timp real, oferind o trimitere rapidă a datelor. Mesajul transmis pe rețea conține ca parametri:

- *Payload*-ul ce este reprezentat de încărcătura de date înregistrată de nod;
- QoS ce asigură trimiterea mesajului către *subscriber*. Un nivel înalt de QoS oferă o siguranță mai mare ca mesajul să fie trimis, dar și o latență și o lățime de bandă mai mari. Nivelurile QoS sunt:
 - Maximum o dată (0), fără confirmare. Nu oferă o garanție a sosirii mesajului.
 - Minimum o dată (1), cu confirmare.
 - Fix o dată (2), cu un *handshake* realizat în 4 pași, fiind cel mai sigur, dar și cel mai încet.
- Colecția de proprietăți
- Numele *topicului*

MQTT este identificat ca fiind un protocol *lightweight* deoarece mesajele pot fi de minimum 2MB respectiv 256MB maximum.

II.3.2 Protocolul WebSocket

WebSocket este un protocol de comunicare ce vine peste TCP/IP, folosit în comunicarea server-client. Acesta este un protocol bidirecțional, *stateful*, păstrând conexiunea între client și server deschisă până când unul dintre cei doi o va închide.

Acesta este folosit în special în aplicațiile de tip *real time*, oferind posibilitatea de a transmite datele încontinuu și mult mai rapid, folosind aceeași conexiune deja deschisă.

II.3.2 MQTT Broker

Serverul de mesaje sau MQTT *Broker* este responsabil atât de gestionarea conexiunilor clienților MQTT, de interceptarea *request*-urilor *publish*, *subscribe*, *unsubscribe*, cât și de distribuirea mesajelor.

Printre *broker*-ele MQTT existente se află și Eclipse Mosquitto, un *broker open source* ce implementează versiunile 5.0, 3.1.1 și 3.1 ale protocolului MQTT. Mosquitto este un *broker*

lightweight, fiind dezvoltat în limbajul de programare C. Acesta oferă și o bibliotecă în C pentru implementarea clienților MQTT .

II.3.2 MQTT Client

Clientul MQTT este o aplicație ce implementează MQTT peste TCP/IP pentru a trimite și primi mesaje. Clientul MQTT ce trimite mesajele către *broker* este *Publisher*, iar cel ce le primește de la *broker* este *Subscriber*. Fiecare client poate fi *Subscriber*, *Publisher*, dar și ambele în același timp.

II.4. Docker, Containerizare și Izolare

Docker este o platformă definită de mai multe componente, menită să faciliteze dezvoltarea, livrarea și rularea de aplicații într-un mediu izolat. Acest lucru permite eliminarea diferențelor de sistem, asigurând o compatibilitate maximă și un timp de *setup* minim. Astfel, mediul este uniform între mașinile folosite în dezvoltare și cele din infrastructură.

Arhitectura folosită de către Docker este client-server, unde serverul este un proces *daemon*, fiind responsabil cu împachetarea imaginilor, rularea containerelor și distribuirea lor. Clientul este folosit pentru management. Interfața dintre aceștia o reprezintă un API REST, fie expus pe *socket*-uri UNIX, fie prin intermediul unui adaptor de rețea.

Serverul este reprezentat prin procesul *dockerd*, ce ascultă pentru noi cereri și administrează obiecte Docker precum imagini, containere, rețele și volume, însă poate și orchestra alți *daemons* în modul Docker Swarm pentru a rula distribuit containerele. Clientul *docker* este modul primar pentru a interfața cu serverul, folosirea de comenzi precum *docker run* fiind traduse în apelul REST corespunzător și apoi tratat de către server. De asemenea, un client *docker* poate interfața cu mai multe servere.

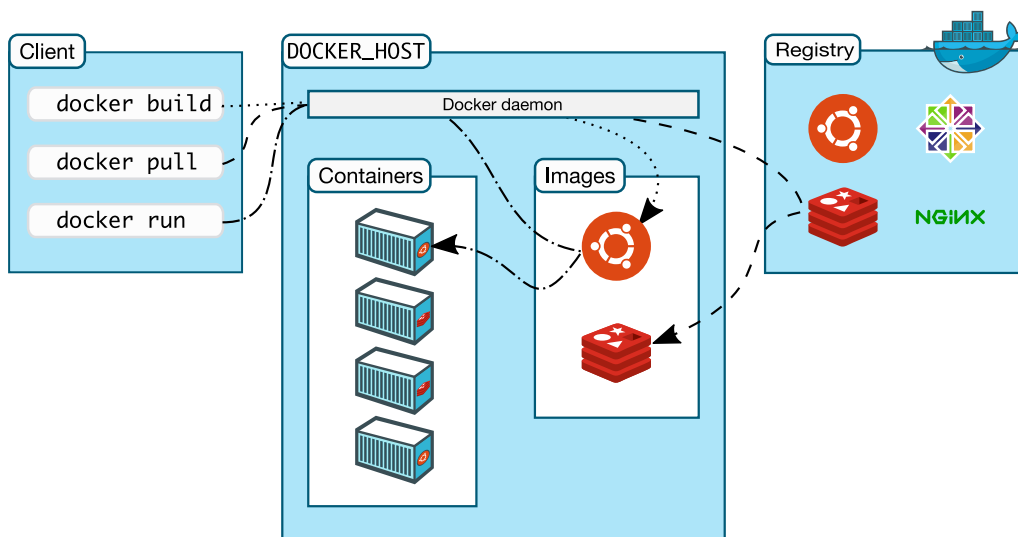


Figura 2.3. Arhitectura Docker

Serverul administrează mai mult obiecte precum imagini, containere, rețele, volume și alte tipuri de obiecte [4].

Imaginile reprezintă șabloane folosite pentru a instanția containere. Acestea pot fi descărcate atât dintr-un Docker Registry precum Docker Hub, dar și construite prin folosirea unor fișiere de tipul *Dockerfile* ce specifică o imagine de bază, de exemplu OpenJDK 17. Fiecare instrucțiune creează câte un strat al imaginii, permițând astfel *caching*-ul ce face procesul de reconstruire al imaginii mult mai rapid, dar și facilitează reutilizarea de imagini, fapt ce reduce semnificativ cerințele la nivelul spațiului de stocare.

Containerul reprezintă instanțierea unui imagini, rulând implicit în izolare față de sistemul gazdă. Această izolare poate fi configurată prin atașarea unui rețele Docker ce permite comunicarea cu alte containere, prin atașarea la rețeaua sistemului gazdă sau atașarea unui volum prin montarea de directoare. Avantajul oferit de un container față de soluțiile clasice de virtualizare constă în reutilizarea elementelor din sistemul gazdă cum ar fi sistemul de operare și librării esențiale, fapt ce duce la o reducere semnificativă a resurselor consumate. Pentru orchestrarea unui soluții bazate pe mai multe containere, folosirea lui *docker compose* este recomandată. Acesta este un instrument pentru definirea și rularea de aplicații cu mai multe containere, unde un fișier YAML este folosit pentru a declara structura soluției. Apoi, comanda *docker compose up* poate fi folosită pentru a rula întreaga soluție într-un mod facil și rapid.

II.5. Machine Learning

Machine Learning este știința programării calculatoarelor astfel încât ele să învețe din date. [5] Pentru a fi util, un sistem *Machine Learning* trebuie prima dată antrenat cu date de *training*, iar în funcție de tipul sistemului pot fi folosite și etichete reprezentând valoarea corectă a predicției cu ajutorul căreia urmează să fie efectuat antrenamentul. Clasificarea antrenamentelor poate fi făcută pe baza mai multor criterii precum:

- Necesitatea supervizării de către un operator uman:
 - Supervizat: sunt folosite etichete în procesul de antrenare;
 - Ne-supervizat: nu sunt folosite etichete în procesul de antrenare;
 - Semi-supervizat: este antrenat inițial în modul supervizat, apoi rafinat în modul ne-supervizat;
 - Învățare ranforsată: învățare bazată pe recompense.
- Dacă pot învăța incremental:
 - Învățare *online*: este antrenat incremental fie secvențial cu date individuale, fie în *mini-batch*-uri;
 - Învățare *batch*: sistem ce nu poate învăța incremental, ce trebuie antrenat folosind toate datele disponibile.
- Modul de învățare:
 - Bazat pe instanță: sistemul învață datele;
 - Bazat pe model: sistemul creează un model pe baza datelor de antrenament.

Învățarea bazată pe model este cea mai întâlnită formă de *Machine Learning*, fiind disponibile atât modele simple precum regresia liniară, până la modele complexe precum pădurile stochastice. Antrenarea ponderilor se face fie folosind o funcție de *fitness*, măsurând nivelul cu care modelul surprinde caracteristicile datelor, scopul fiind maximizarea, fie o funcție de cost, scopul fiind minimizarea. Pentru probleme ce necesită valori discrete (regresiile), o funcție de cost este favorabilă. Printre cele mai populare funcții de cost se află Eroarea Medie Absolută, dar și Rădăcina Erorii Medii Pătratice.

Rețelele neuronale reprezintă evoluția modelelor clasice de *Machine Learning*, creând o paralelă între modul în care funcționează creierul uman și felul în care acest sistem procesează datele, conectând mai mulți neuroni responsabili cu calcule simple. Unitatea de bază a unei rețele neuronale este *perceptronul*, ce acceptă o valoare de intrare, aplică o funcție de activare

apoi trimite valoarea rezultată către următorul neuron. În funcție de necesitatea aplicației, există mai multe funcții de activare ce pot fi folosite, precum:

- Liniară: nu alterează valoarea;
- Funcția de pas Heaviside: transformă valorile mai mici ca 0 în 0 și cele mai mari în 1;
- Unitate liniară rectificată: transformă valorile mai mici decât 0 în 0, iar restul nu sunt modificate;
- Funcția sigmoid: folosită când o probabilitate este necesară, având valori între 0 și 1;
- Funcția tangentă hiperbolică: similară cu funcția sigmoid, însă cu valori între -1 și 1.

Similar unui sistem clasic de inteligență artificială, antrenamentul este necesar. Acesta este efectuat prin procesul de *back-propagation*, bazat pe o funcție de cost. Concret, execută un *forward-pass* efectuat cu ponderile inițializate stochastic, iar apoi calculează costul relativ al etichetei de antrenament. Apoi, folosind procesul de *back-propagation*, ponderile sunt ajustate conform tehnicii de optimizare folosite, multiplicată cu rata de învățare.

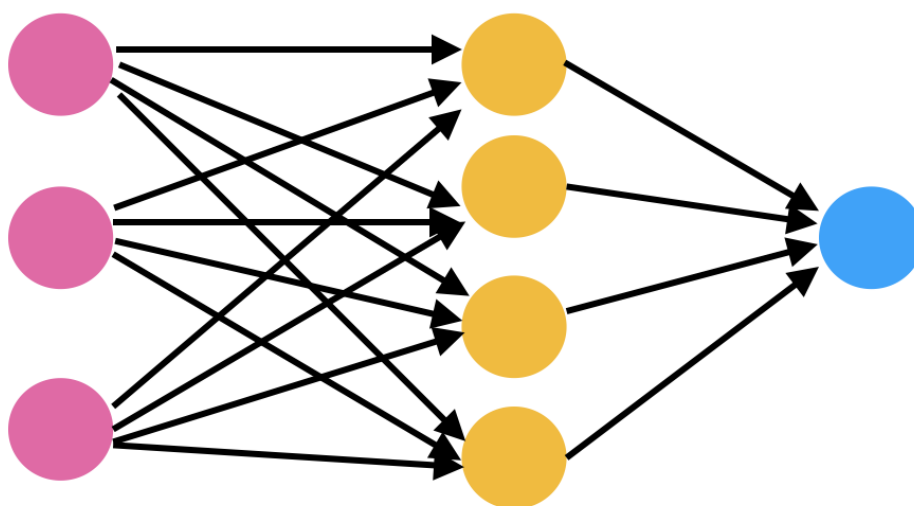


Figura 2.4. Rețea neuronală simplă

Cazuri particulare ale acestor rețele neuronale sunt rețelele neuronale convoluționale și cele recurente, ce oferă rezultate sporite pentru cazuri bine definite. Rețelele neuronale convoluționale sunt cel mai des folosite pentru clasificarea de imagini, fiind definite de stratul convoluțional ce oferă invarianță spațială. Rețelele neuronale recurente sunt folosite adesea pentru reprezentări secvențiale în timp, precum procesarea limbajului natural. Acestea sunt folosite și pentru regresii pe serii de timp, fiind capabile să memoreze date anterioare folosite în efectuarea predicției curente [6].

III. Descrierea soluției

III.1. Arhitectura soluției

Scopul acestei lucrări de disertație este de a proiecta o soluție de tip casă inteligentă modulară și extensibilă. Arhitectura soluției constă în mai multe noduri IoT de design propriu, *gateway* IoT și mai multe micro-servicii lansate într-o instanță cloud de tip IaaS.

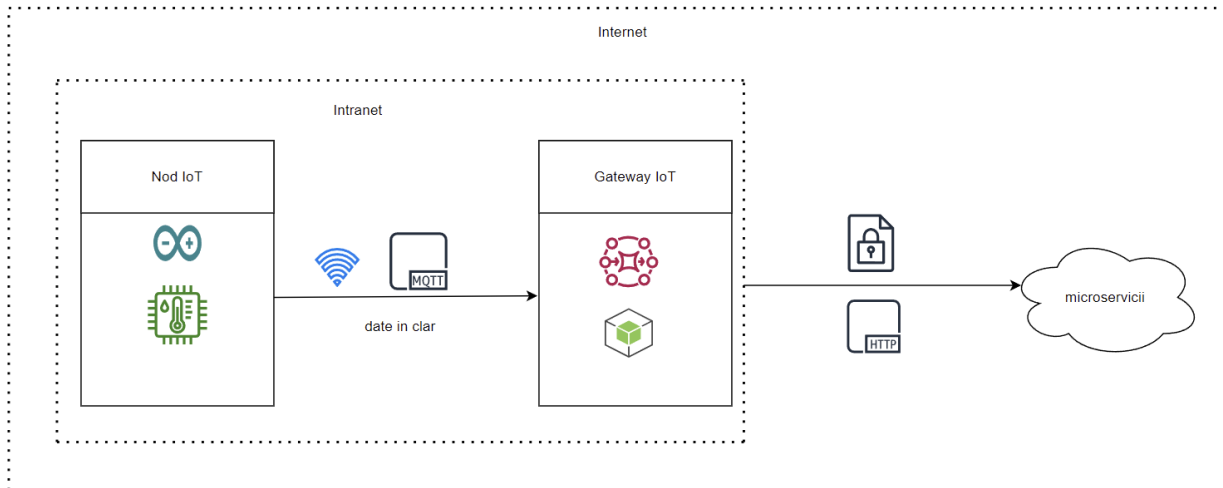


Figura 3.1. Arhitectura soluției

Nodurile IoT folosite sunt plăci de dezvoltare ESP8266, conectate la rețeaua Wi-Fi ce au atașați diverși senzori pe interfața serială. Datele acestora sunt transmise în clar și transportate prin intermediul protocolului MQTT către *broker*-ul găzduit pe *gateway*-ul IoT. *Gateway*-ul folosit este un Raspberry Pi 4 cu 1GB RAM rulând sistemul de operare Raspberry Pi Lite.

La nivel de *gateway*, a fost configurat *broker*-ul Eclipse Mosquitto care ascultă pe portul TCP 1883 noi cereri de conectare. *Broker*-ul va distribui mesajele publicate către abonații acestora, având de asemenea posibilitatea păstrării ultimului mesaj pentru a fi distribuit noilor abonați, cât și a ajustării calității serviciului. Autentificarea clienților se face pe bază de nume de utilizator și parolă.

În componența soluției se află de asemenea și o aplicație NodeJS ce deservește două scopuri, client MQTT și client HTTPS. Aceasta se abonează la *broker* pentru diverse topicuri la care sunt publicate datele senzorilor urmând a fi împinse către *cloud* prin intermediul HTTPS POST. Pentru a rezuma, *gateway*-ul IoT va concentra toate datele primite în Intranet de la diversele noduri, urmând a le împinge în *cloud* pentru stocare și analizare. Clientul de MQTT

ales este MQTT.js ce este un proiect *Open-Source* ce permite conectarea folosind protocolul TCP.

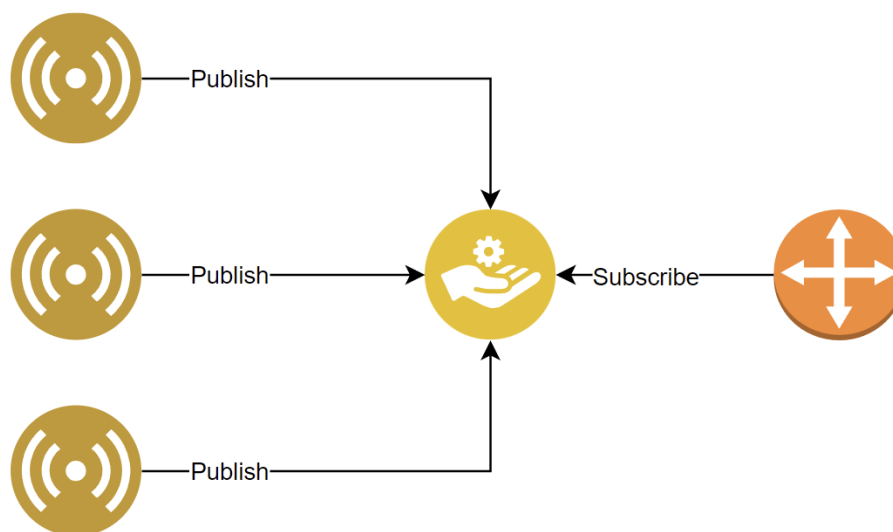


Figura 3.2. Modelul publish/subscribe

În cloud, datele sunt trecute printr-un microserviciu *web facing*, reprezentând *gateway*-ul, urmând a fi trimise mai departe către microservicii private. Aceste microservicii sunt găzduite într-o instanță Oracle Cloud Instance și rulate în containere Docker pentru a facilita atât punerea în funcțiune a aplicației, cât și securitatea prin izolare oferită de containerizare. De asemenea, modul Swarm este activat, fapt ce determină atât un grad sporit de reziliență la diverse probleme ce pot apărea pe durata execuției, cât și un răspuns la cerere foarte rapid prin scalabilitatea oferită de către acesta. Pentru dezvoltarea microserviciilor private au fost folosite atât Spring Boot și Java pentru serviciul OLTP și pentru cel de tip Data Warehouse, cât și Flask și Python pentru dezvoltarea microserviciului responsabil cu operațiile de tip *Machine Learning*. Pentru a facilita utilizarea acestei soluții, o interfață grafică dezvoltată în Next.js folosind componente din populara librărie de React Material UI a fost adăugată. *Framework*-ul acesta a fost folosit pentru performanța deosebit de ridicată pe care o oferă, dar și pentru posibilitățile de optimizare pentru motorul de căutare.

Toate cererile HTTP sunt adresate *gateway*-ului, care pe lângă rolul principal de rutare are și rol de autentificare și autorizare. Cererile către interfața grafică sunt permise și utilizatorilor ne-autentificați, iar pentru utilizarea microserviciilor este necesară autentificarea. Autentificarea și autorizarea utilizatorilor este facilitată de utilizarea furnizorului de identitate

Auth0 și este realizată folosind JSON Web Tokens, permițând decuplarea stării de autentificare de server-ul unde aceasta a fost efectuată.

Această arhitectură a fost aleasă pentru flexibilitatea și extensibilitatea pe care le oferă, având de asemenea un grad sporit de securitate. Toate nodurile vor publica în același topic, nefiind necesară vreo configurare pentru introducerea de noi senzori în rețea. Pentru a putea accesa datele de la noduri, este necesară asocierea *gateway*-ului unui cont. Acest proces este facilitat de către interfața grafică dezvoltată în Next.js. De îndată ce împerecherea a fost încheiată cu succes, datele vor fi împinse în *cloud* către *gateway*, care va delega rezolvarea cererii către microserviciul OLTP. Acesta este dedicat stocării informației într-o bază de date relațională Oracle Express Edition 21c ce efectuează operații de tip CRUD. Acest pas va crea un strat de persistență și va facilita accesul la date printr-un format ușor de manipulat. De asemenea, acest microserviciu este folosit și în cadrul interfeței grafice pentru a popula date semnificative despre valorile actuale ale senzorilor, precum și date salvate de către utilizator despre locuința sa și amplasamentul senzorilor per cameră.

Pe lângă microserviciul OLTP, a mai fost implementat serviciul ce permite accesarea datelor structurate în modul *Data Warehouse* ce sunt stocate într-o bază de date Oracle Express 21c și servite prin intermediul protocolului HTTPS, dar și serviciul responsabil cu efectuarea operațiilor *Machine Learning* folosind un model pre-antrenat.

De asemenea, această soluție este ușor scalabilă și fiabilă, fiind concepută *cloud-native* și *fault-tolerant*. Modul Docker Swarm permite aprovizionarea rapidă cu noi containere în cazul în care procesul inițial este încheiat de către o eroare, dar și creșterea sau scăderea numărului de replici pentru a ajusta corespunzător nivelul de încărcare.

III.2. Lista de componente

Componentă	Preț(lei)	Cantitate(buc)	Preț Total(lei)
Placă de dezvoltare ESP8266	36.99	2	73.98
Senzor de temperatură și umiditate DHT11	8.98	2	17.96
Raspberry Pi 4 1 GB RAM	173	1	173
Card MicroSD 32GB	36	1	36
Total(lei)		300.94	

Tabel 3.1. Lista de componente și prețuri estimative în iunie 2022

1. Placa de dezvoltare ESP8266

ESP8266 este un microcip cu capabilități Wi-Fi produs de către compania Espressif. Acesta este definit de o arhitectură pe 32 de biți, având un procesor *Tensilica Diamond Standard* 106Micro tactat în mod implicit la 80MHz, însă care poate fi setat să ruleze și la o frecvență de 160Mhz. Ca și memorie volatilă, prezintă 32KB rezervați pentru instrucțiuni, și 80KB disponibili pentru rularea de programe de către utilizator.

Un factor important în alegerea unei plăci de dezvoltare cu acest *chip* a fost suportul pentru stiva TCP/IP precum și suportul pentru funcții criptografice cu o intensitate scăzută, pentru funcțiile *hash* și calculul de coduri de autentificare a mesajelor.

Două plăci de acest tip au fost folosite pentru realizarea lucrării, ele având rolul de a transmite date de la senzori către *gateway*.

2. Senzor de temperatură și umiditate DHT11

DTH11 este un senzor de temperatură și umiditate ce produce un semnal digital pe *pin*-ul de date. Temperatura este măsurată folosind un termistor cu coeficient negativ iar umiditatea relativă este măsurată folosind un senzor capacitiv. Senzorul poate citi valori ale umidității în intervalul 20-90% RH cu o acuratețe la măsurare de +/- 5% RH. Temperatura poate lua o valoare între 0 și 60°C cu o acuratețe de +/-2°C. Senzorul permite un voltaj de intrare în intervalul 3.3V și 5V.

3. Raspberry Pi 4 1GB RAM

Raspberry Pi 4 este o placă de dezvoltare ce conține un procesor actualizat față de generațiile precedente, având o arhitectură pe 64 de biți și un procesor cu 4 nuclee tactate la 1.5GHz ce are încorporat un disipator de căldură din metal. De asemenea, pe partea de conectivitate acesta are 3 *port*-uri USB 3.0, placa de rețea Wi-Fi ce funcționează atât în modul 2.4GHz cât și 5GHz, Bluetooth 5.0 *Low Energy*, *Gigabit Ethernet* cu capabilități de *Power over Ethernet*, dar și *port*-uri Micro HDMI ce suportă rezoluții până la 4K.

Versiunea folosită pentru realizarea proiectului este dotată cu 1GB de memorie volatilă și rulează sistemul de operare Raspberry OS Lite. Această placă de dezvoltare este folosită pe post de *gateway*, susținând infrastructura necesară conectării senzorilor și transmiterea de date atât de la senzor la *gateway* cât și de la *gateway* în *cloud*. Concret, *gateway*-ul găzduiește atât un *broker* Mosquitto MQTT cât și aplicația ce colectează datele publicate și le transmite securizat folosind HTTPS către *Cloud*.

4. Card MicroSD 32GB

Cardul MicroSD este folosit pentru a dota Raspberry Pi-ul cu memorie non-volatilă, alegându-se capacitatea de 32GB pentru a permite instalarea sistemului de operare și a tuturor aplicațiilor necesare.

Conectarea fizică a senzorilor cu placa de dezvoltare a fost efectuată conform tabelului de mai jos.

Componentă	Pin Componentă	Pin ESP8266
DHT11	VCC	3V
	DATA	D1
	GND	G

Tabel 3.2. Tabelul asociat schemei electrice

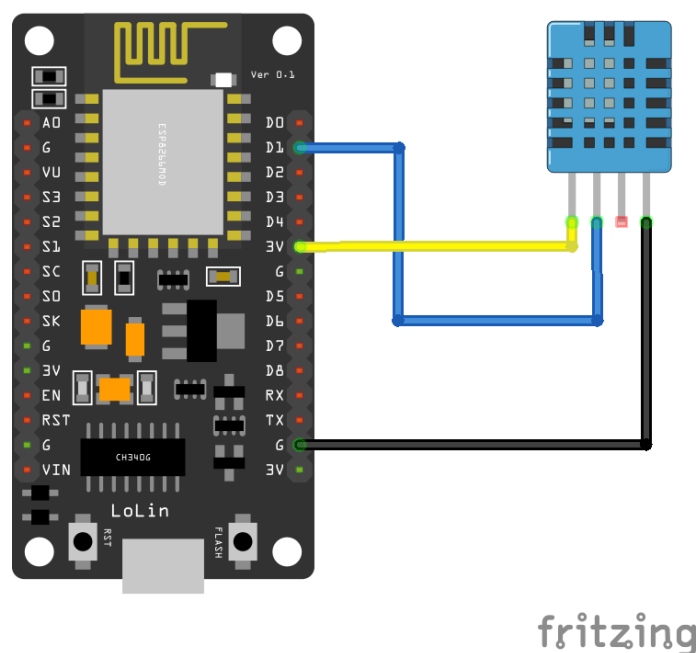


Figura 3.3. Schema electrică

Pentru stratul de percepție al nodului, senzorul de umiditate și temperatură DHT11 a fost conectat la placa de dezvoltare ESP8266. Această placă de dezvoltare este dotată cu 17 pini GPIO, având suport pentru interfețele SPI, I2C, UART dar și Analog-to-Digital pe 10 biți. Alimentarea senzorului a fost efectuată folosind pinul de 3V prezent pe placă, voltaj permis conform fișei tehnice a acestuia. Pinul de date a fost ulterior conectat direct către pinul GPIO5, marcat pe placă cu simbolul D1, iar împământarea a fost legată de pinul G, conform sursei [7].

III.3. Descrierea nodului IoT

Nodurile IoT sunt constituite din plăci de dezvoltare ESP8266 având conectate pe interfața serială diverși senzori. Un exemplu de acest fel ar fi senzorul de umiditate(%) și temperatură(°C) DHT11. Nodurile IoT comunică cu *gateway*-ul prin intermediul protocolului la nivel de aplicație MQTT, iar la nivel de transport prin TCP.

Aceste noduri sunt conectate la rețeaua Wi-Fi de tip intranet, ele nefiind menite să se conecteze în internet din considerente de securitate. Datele recepționate de la senzori sunt trimise prin intermediul protocolului MQTT. Concret, nodurile publică pe același topic, fiind clienți MQTT. Plasarea nodurilor în intranet a fost aleasă strategic, permițând economisirea bateriei și limitarea consumului de resurse prin renunțarea implementării securității la nivel de transport prin TLS, bazându-se strict pe confidențialitatea oferită de către WPA3-PSK la nivelul legăturii de date. Autentificarea nodurilor la *broker* se face pe baza unor credențiale scrise în

EEPROM. De îndată ce nodurile sunt conectate la Wi-Fi, acestea vor începe să trimită date către *broker*, nefiind necesară o configurare adițională.

Dezvoltarea componentei *software* aferente nodului IoT a fost făcută în *framework*-ul Arduino, ce permite folosirea limbajului C++ și o paradigmă orientată obiect. De asemenea, conform sugestiei din documentație [8], abstractizarea sistemului de asamblare a codului binar și de gestionare a dependențelor a fost realizată cu ajutorul extensiei de Visual Studio Code numită PlatformIO. Această extensie permite, de asemenea, și un sistem avansat de completare și sugestie a codului, suport pentru depanare, dar și o portabilitate a codului ridicată. Compilatorul folosit este o versiune modificată a GCC (*GNU Compiler Collection*), ce suportă standardul C++ 11 și un subset al librăriei *std* pe lângă librăriile specifice *framework*-ului Arduino. Acesta conține și librării destinate accesării funcțiilor prezente pe ESP8266, precum Wi-Fi sau librării ce oferă posibilitatea de a restarta nodul programatic. Pe lângă librăriile standard oferite, dependența *PubSubClient* a fost adăugată prin intermediul *managerului* oferit de către PlatformIO. Aceasta include un client de MQTT sub forma unei clase omonime cu numele librăriei, ce primește ca parametru al constructorului mediul peste care este efectuată comunicarea, în acest caz fiind Wi-Fi prin intermediul *WiFiClient*, ce face parte din librăria standard *ESP8266WiFi.h*.

Structura codului este marcată de cele 2 metode specifice *framework*-ului Arduino, *setup* și *loop*, unde *setup* reprezintă logica de inițializare a nodului ce este executat de fiecare dată când nodul este pornit, iar *loop* rulează în buclă logica de *business*.

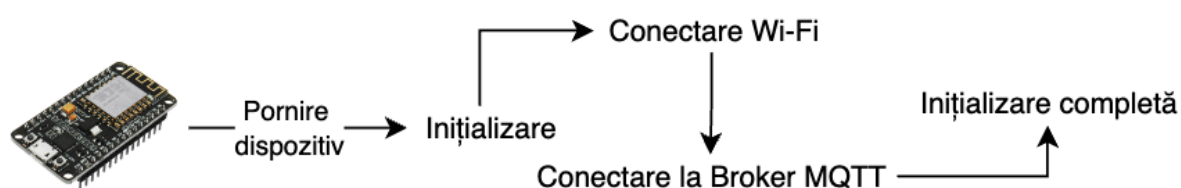


Figura 3.4. Schemă inițializare nod

Inițializarea nodului este marcată de 2 procese, conectarea la rețeaua Wi-Fi și conectarea la *broker*-ul MQTT. Conectarea la rețeaua Wi-Fi este efectuată folosind librăria standard Wi-Fi *ESP8266WiFi.h*, operațiunile fiind abstractizate prin obiectul *WiFi*. Placa de rețea este setată în modul *Station* ce permite conectarea la puncte de acces, iar credențialele sunt citite din

EEPROM și folosite ca parametri pentru metoda *begin*. Acest proces este efectuat în buclă la un interval de 5 secunde până ce conexiunea va fi stabilită cu succes.

Următoarea secvență de cod realizează conectarea la rețeaua Wi-Fi.

```
static void connectWifi()
{
    auto status = WiFi.status();
    if (status == WL_CONNECTED)
    {
        return;
    }

    Serial.printf("Current WiFi status: %d\n", status);

    while ((status = WiFi.status()) != WL_CONNECTED)
    {
        Serial.printf("Connecting to WiFi %s. Current status: %d. Current IP: %s\n",
            ssid, status, WiFi.localIP().toString().c_str());
        WiFi.mode(WIFI_STA);
        WiFi.config(localIP, gateway, subnet);
        WiFi.begin(ssid, pass);

        delay(5000);
    }

    Serial.println("Connected to WiFi!");
}
```

În urma stabilirii conexiunii Wi-Fi, conectarea la *broker*-ul MQTT este încercată. Metoda *setServer* aferentă clientului MQTT este apelată, folosind ca parametri IP-ul și *portul broker*-ului, iar apoi identificatorul unic. Acest identificator este format în urma concatenării șirului de caractere "esp8266-" cu adresa MAC a plăcii de rețea și va fi ulterior utilizat ca parametru al metodei *connect*. În cazul în care metoda întoarce un cod de eroare, nodul se va restarta și va reîncerca conexiunea, altfel inițializarea va fi completă și metoda *loop* va prelua controlul execuției.

Următoarea secvență de cod realizează conectarea la *broker*-ul MQTT.

```
mqttClient.setServer(broker, port);

Serial.printf("Attempting to connect to MQTT broker %s:%d\n", broker, port);
strcat(clientId, WiFi.macAddress().c_str());
if (!mqttClient.connect(clientId))
{
    Serial.println("Failed to connect. Will restart...");
}
```

```
ESP.restart();
}
Serial.println("Connection to MQTT broker established successfully");
```

Metoda *loop* execută logica de business în buclă, folosind aceeași metoda *connectWifi* pentru a asigura stabilitatea conexiunii, urmând apoi să folosească metoda *loop* a clientului MQTT. Apelarea acesteia este necesară deoarece clientul funcționează în mod sincron, fapt fără de care schimbul de informații dintre *broker* și client nu s-ar putea efectua. Nodul IoT manipulează transparent schimbul de date, colectarea de date din senzor fiind delegată către o metodă specializată pentru a păstra flexibilitatea soluției. Datele sunt publicate în topicul *data* sub format JSON, având ca proprietăți *origin*, ce reprezintă identificatorul nodului, tipul datei și valoarea acesteia.

Următoarea secvență de cod realizează publicarea datelor.

```
static void publishData(const char *capability)
{
    Serial.printf("Publishing data about %s\n", capability);

    char    payload[128];
    const float generatedValue = collectData(capability);
    sprintf(payload, "{\"origin\": %s, \"type\": %s, \"value\": %.2f}", clientId,
        capability, generatedValue);

    mqttClient.publish(topic, payload);
    Serial.println("Data published");
}
```

Logica de *business* este efectuată în buclă la o frecvență de 5 secunde, aceasta fiind impusă prin apelarea metodei *delay* specifice *Arduino Framework*.

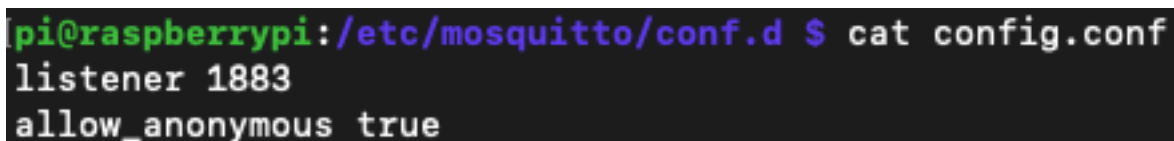
III.4. Descrierea *Gateway*-ului

Gateway-ul IoT este constituit dintr-o placă de dezvoltare Raspberry Pi 4 cu 1 gigaoctet de memorie volatilă și 32 de gigaocteți memorie non-volatilă, rulând sistemul de operare Raspberry Pi OS Lite. Acest sistem de operare are un consum scăzut de resurse, fiind gândit pentru a fi folosit strict prin intermediul SSH, neavând vreun mediu de lucru instalat.

Acest dispozitiv găzduiește atât *broker*-ul MQTT, cât și o aplicație NodeJS ce este abonată la topicul *data* și trimite toate mesajele mai departe către *gateway*-ul din cloud. Concret,

scopul acestuia este să centralizeze toate datele de la nodurile reprezentate de dispozitive cu putere de procesare redusă și să le împingă pentru a fi procesate și persistate.

Broker-ul MQTT folosit este Mosquitto, fiind dezvoltat de către Eclipse Foundation. Acesta este configurat să asculte pe *portul* TCP 1883 pentru noi conexiuni, permițând conectarea fără a fi necesare credențiale.



```
pi@raspberrypi:/etc/mosquitto/conf.d $ cat config.conf
listener 1883
allow_anonymous true
```

Figura 3.5. Configurarea broker-ului Mosquitto

Conform protocolului MQTT, acesta suportă multiple strategii de trimitere a mesajelor, configurabile prin intermediul parametrului QoS (*Quality of Service*):

- 0, modul „*fire and forget*”, garantează trimiterea mesajului o singură dată, fără a mai fi necesară confirmarea primirii;
- 1, modul „*at least once*”, garantează trimiterea mesajului cel puțin o dată, fiind necesară confirmarea;
- 2, modul „*exactly once*”, garantează trimiterea mesajului exact o singură dată, fiind necesară confirmarea și lipsa duplicatelor.

Păstrarea ultimului mesaj cu scopul distribuirii către noii abonați este de asemenea importantă pentru topicurile actualizate rar.

Gateway-ul nu deschide portul 1883 pentru comunicații din internet, fiind astfel restricționată utilizarea *broker*-ului doar de către utilizatorii aceleiași rețele, precum nodurile IoT.

Pe lângă *broker*-ul MQTT, dispozitivul găzduiește și o aplicație NodeJS scrisă în Typescript ce este abonată la topicul *data* și împinge mai departe datele în *cloud* prin intermediul protocolului HTTPS pentru a asigura confidențialitatea încărcăturii. Pentru dezvoltarea acestei aplicații au fost folosite pachete precum *mqtt*, ce reprezintă un client MQTT bazat pe conceptul de evenimente, *npmlog*. Acesta permite înregistrarea de mesaje fie cu scopul informării, fie cu scop de înregistrare a erorilor produse sau strict pentru depanare. Un alt pachet este *node-fetch*, ce permite utilizarea Fetch API în mediul Node JS.

Datele necesare configurării acestei aplicații au fost externalizate în fișiere JSON precum *credentials.json*, unde sunt stocate informații necesare conectării către *broker*, dar și *jwt.json* unde sunt stocate variabile necesare obținerii unui *JSON Web Token* pentru autentificarea cererilor către *Cloud Gateway*.

Conectarea către *broker* este efectuată prin apelul funcției *connect* aferente obiectului *mqt*, ce va declanșa publicarea de evenimente pe bucla NodeJS. Evenimentului *connect* îi este asociat un *callback* ce efectuează abonarea la topicul *data*, iar evenimentului *message* îi este asociat un *callback* ce trimite prin HTTPS POST datele către *Cloud Gateway*.

Următoarea secvență de cod reprezintă partea responsabilă cu conectarea la *broker*-ul MQTT din aplicația NodeJS de pe IoT *gateway*.

```
client.on("connect", async () => {
  log.info(
    tag,
    `Connection to broker ${credentials.broker}:${credentials.port} has been established successfully`
  );

  await obtainBearerToken();
  await signalUp();

  client.subscribe(topic, (error) => {
    if (error) {
      log.error(tag, `Error subscribing to ${topic}. ${JSON.stringify(error)}`);
      return;
    }

    log.info(tag, `Subscribed to topic ${topic} successfully`);
  });
});

client.on("message", (topic: string, payload: Buffer) => {
  log.info(tag, `Message from topic ${topic} has arrived`);
  log.info(tag, `Content: ${payload.toString()}`);

  postMessageToGw(JSON.parse(payload.toString()) as Payload);
});
```

Autentificarea dispozitivului este efectuată prin intermediul unui *token* purtător obținut în urma *Client Credentials Flow* oferit de către Auth0 [9], ce implică un HTTPS POST către furnizorul de identitate cu scopul obținerii unui *JSON Web Token*. Datele necesare efectuării acestei cereri sunt citite din fișierul *jwt.json* ce conține date despre client, dar și despre audiența cerută.

```

info gw-app Connection to broker 192.168.0.165:1883 has been established successfully
info gw-app Bearer token obtained: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6Ikh0S0JZN2I1T2IsYy1lNTd0TWJudCJ9.eyJp
c3MiOiJodHRwczovL2Rpc3NlcnRhdGlvbnVhYSSldS5hdXRoMC5jb20vIiwic3ViIjoicEppRutDQzJ2SFRnNzFkaFRWZnV4WEMxOHk2MlNuMzhAY2xp
ZW50cyIsImF1ZCI6Imh0dHA6Ly9sb2NhbgVhc3Q6ODAA4MC9kYi8iLCJpYXQiOiE2NTQ3MDM0MTcsImV4cCI6MTY1NDc4OTgxNywiYXpwIjoicEppRutD
QzJ2SFRnNzFkaFRWZnV4WEMxOHk2MlNuMzgiLCJndHkiOiJibGllbn0tY3JlZGVudGllbHhMifQ.a1JlUwZDNfUVZnoZRJEPG0TGuaxIacBDVRVYa1v0SR
9rLMse3FcDlgyM1ZyFTswXEsKI9raprky1PVeTXp4cqqSuQNNaRNwbyJYH3De-MDqgBiLLSLnzgpFNx3mgef0jvDBkP6E22pHU3mhoelFCBepRTakNM
FHFF-cqorMw6Iig8hYxiE3zxwM5RWixplwHD-FfHJ0wM0160RxVAJwVA2ZIKCYnCNyf7 An Whjpuh8iHm9Ff2W0HoQFtRzaxWjRV0mVarnpqCF9lCuC
YcI6GHpieEzuLLVDH-q-LNeRdWopm4rNXDLPhDoPX21yPfPt7D74j79UbNGiczR7EV0g
info gw-app Subscribed to topic data successfully

```

Figura 3.6. Obținerea token-ului

În urma conectării la *broker* și a obținerii *tokenului* purtătorului, aplicația va semnaliza disponibilitatea IoT *gateway*-ului către *cloud* printr-un apel HTTPS POST. Dacă răspunsul conține câmpul *pairedTo* gol, *gateway*-ul va fi pus în modul *pairing*, fapt ce determină apelul acestei metode în buclă până când un utilizator asociază aparatul folosind codul de împerechere specific. După împerechere, aplicația va trimite în *cloud* toate înregistrările publicate pe topicul *data*.

III.5. Microservicii dezvoltate

Partea integrantă a soluției este constituită de microserviciile dezvoltate folosind fie *framework*-ul Spring Boot în Java, fie Flask în Python, ce sunt găzduite pe o instanță de *cloud* Oracle. Arhitectura aceasta a fost folosită în detrimentul monolitului din prisma beneficiilor multiple ce sunt oferite pe partea de distribuire și mentenanță, în ciuda efortului de dezvoltare asociat.

Funcțional, acestea realizează fie rol de aplicație OLTP asociată cu o bază de date relațională Oracle Express Edition 21c, fie rol de aplicație Data Warehouse asociată cu aceeași bază de date menționată anterior, însă folosind o altă schemă. De asemenea, în cadrul microserviciilor se poate regăsi și o aplicație specializată pentru *machine learning*. Toate cererile sunt securizate prin intermediul protocolului HTTPS și autentificate folosind JSON *Web Token*-uri emise și verificate de către autoritatea emitentă Auth0.

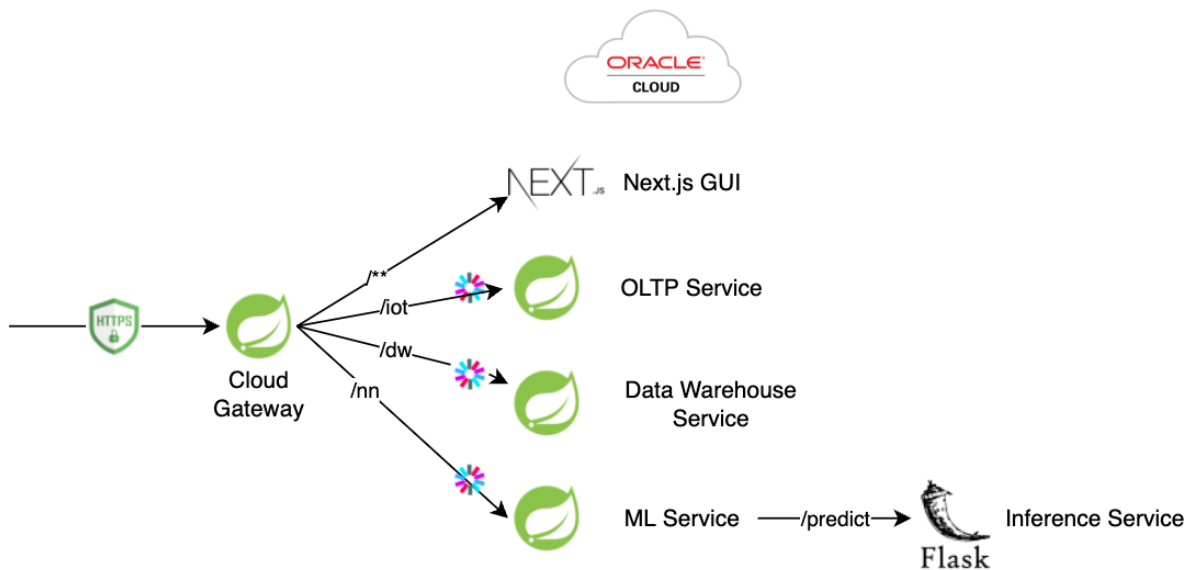


Figura 3.7. Diagrama microserviciilor

III.5.1. Microserviciul *Cloud Gateway*

Tehnic, toate aceste cereri sunt trecute printr-un *gateway* care prima dată autentifică în funcție de calea folosită și apoi rotează către microserviciul responsabil. Partea de autentificare este facilitată de folosirea *starter*-ului Spring Cloud Security dar și al *starter*-ului ce verifică transparent JSON *Web Token*-urile primite în câmpul *Authorization* al cererii, numit OAuth 2.0 Resource Server. Transparența aceasta este obținută prin folosirea datelor din fișierul de configurare pentru a injecta un *Bean* ce folosește URL-ul autorității emitente pentru a verifica validitatea *token*-ului. Pentru a modifica setările implicite care obligă autentificarea fiecărui *request*, a fost creat un *WebFilter* ce permite acces neautentificat către interfața grafică pentru a permite utilizatorilor să-și creeze cont. Crearea de utilizatori, emiterea și verificarea de *token*-uri sunt delegate către Auth0, permițând simplificarea arhitecturii de securitate.

Următoarea secvență de cod configurează filtrul de securitate aferent *Cloud Gateway*-ului.

```

@Configuration
@EnableWebFluxSecurity
public class SecurityConfiguration {

    @Bean
    public SecurityWebFilterChain
springSecurityFilterChain(ServerHttpSecurity http) {
        http
            .authorizeExchange()
            .pathMatchers("/**", "/_next/**", "/favicon.ico",
                "/api/auth/**", "/api/gw/**", "/api/home/**",
                "/api/country/**", "/api/city/**",

```

```

        "/api/home/**", "/api/room/**", "/api/record/**",
        "/devices", "/settings")
        .permitAll()
        .anyExchange()
        .authenticated()
        .and()
        .csrf()
        .disable()
        .oauth2ResourceServer()
        .jwt();

    return http.build();
}
}

```

Rutarea cererilor de către *gateway* spre celelalte microservicii este făcută declarativ prin intermediul fișierului de configurare.

Următoarea secvență de cod reprezintă configurarea *Gateway*-ului prin intermediul unui fișier YAML servit de către microserviciul *Config Server*.

```

server:
  port: 8080
spring:
  application:
    name: gateway
  zipkin:
    base-url: http://zipkin:9411/
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri:
https://dissertationuaa.eu.auth0.com/
  cloud:
    gateway:
      routes:
        - id: dbapp
          uri: lb://dbapp
          predicates:
            - Path=/iot/**
          filters:
            - StripPrefix=1

```

```

-CircuitBreaker=dbapp
  - id: mlapp
    uri: lb://mlapp
    predicates:
      - Path=/nn/**
    filters:
      - StripPrefix=1
      - CircuitBreaker=mlapp
  - id: dwapp
    uri: lb://dwapp
    predicates:
      - Path=/dw/**
    filters:
      - StripPrefix=1
      - CircuitBreaker=dwapp
  - id: frontend
    uri: http://nextapp:3000
    predicates:
      - Path=/**
    filters:
      - CircuitBreaker=frontend

```

```

eureka:
  client:
    register-with-eureka: false
    service-url:
      defaultZone: http://registry:8761/eureka
    healthcheck:
      enabled: true

```

Partea de rutare este implementată folosind Spring Cloud Gateway, însă și alte microservicii destinate infrastructurii a trebuit să fie adăugate. Printre acestea se numără un microserviciu de Service Discovery, dar și un microserviciu destinat servirii de configurații dintr-un *repository* de GitHub.

Un alt aspect notabil aparținând infrastructurii îl constituie utilizarea Zipkin, un sistem distribuit de *tracing* ce permite o vizualizare mult mai ușoară a cererilor, precum și depanarea problemelor de latență întâlnite.

Librăria Spring Cloud Gateway a fost aleasă în detrimentul altor soluții similare datorită integrării puternice pe care o oferă cu server-ul de Service Discovery Eureka. Acesta permite rezolvarea serviciului cu cea mai mică încărcătură pentru a servi cererea, dar și integrarea cu librării ce aplică design *pattern*-ul Circuit Breaker. Acest *pattern* vine în completarea celui de Retry și este util în cazurile în care erorile apar din cauza unor evenimente ce nu pot fi anticipate, permițând folosirea unei metode de *fallback* în caz de circuit „deschis”. Librăria Resilience4j a fost aleasă pentru a îndeplini această sarcină, specificând un *circuit breaker* pentru fiecare cale disponibilă. Ca și client de Service Discovery, *starter*-ul Netflix Eureka Client a fost folosit, configurarea acestuia făcându-se prin intermediul fișierului YAML, specificând URL-ul serverului, în speță microserviciul de Service Discovery, dar și faptul că acest client n-ar trebui să se înregistreze server-ului. Pentru o mai rapidă rutare a cererilor către serviciul destinație, URL-urile rezolvate de Eureka sunt păstrate într-un *cache* gestionat prin abstractizarea specifică Spring oferind implementarea concretă Caffeine injectată în context prin intermediul unui *Bean*.

Următoarea secvență de cod reprezintă configurarea *cache*-ului din *Cloud Gateway*.

```
@EnableCaching
@Configuration
public class CacheConfiguration {

    @Bean
    public Caffeine<?, ?> caffeineConfig() {
        return Caffeine
            .newBuilder()
            .expireAfterWrite(10, TimeUnit.MINUTES);
    }

    @Bean
    public CacheManager cacheManager(Caffeine<?, ?> caffeine) {
        var cacheManager = new CaffeineCacheManager();
        cacheManager.setCaffeine((Caffeine<Object, Object>) caffeine);

        return cacheManager;    }
}
```

III.5.2. Microserviciul *Config Server*

Microserviciul destinat configurațiilor este dezvoltat tot în Spring Boot, folosindu-se de *starter*-ul Spring Cloud Config Server. Acesta folosește fișierul de configurare pentru a-și extrage informațiile necesare creării unui server ce încarcă în timp real din GitHub fișiere YAML ce urmează a fi servite către alte microservicii. Pentru rezolvarea fișierului corect, formatul *nume-profil.yml* trebuie să fie folosit, exemplul fiind *gateway-prod.yml*, unde *prod* este numele profilului. Pentru a evita eventualele probleme asociate pornirii multiplelor servicii, portul 8800 a fost folosit. Concret, fișierele de configurare pentru fiecare microserviciu au fost încărcate într-un *repository* privat, iar credențialele de autentificare au fost plasate în fișierul *application-prod.yml* ca și nume de utilizator și *token* de acces generat prin intermediul setărilor destinate dezvoltatorilor din cadrul interfeței *web*.

III.5.3. Microserviciul *Service Discovery*

Rezolvarea serviciilor la momentul rulării este realizată prin intermediul microserviciului de *Service Discovery* [10], dezvoltat de asemenea în Spring Boot și folosind *starter*-ul Netflix Eureka Server. Având în vedere existența server-ului de configurări, configurarea este încărcată la momentul inițializării, declarând în fișierul împachetat cu microserviciul doar numele aplicației și URL-ul serverului. În cadrul fișierului servit din *repository*-ul de GitHub, *portul* a fost setat să fie 8761, iar clientul de Eureka configurat pentru a evita înregistrarea serverului.

Următoarea secvență de cod reprezintă configurarea microserviciului de *Service Discovery*.

```
spring:
  application:
    name: registry

server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

III.5.4. Microserviciul OLTP

Microserviciul OLTP este adresabil folosind prefixul */iot*, iar scopul acestuia este de a persista datele aferente senzorilor, dar și date despre locuința utilizatorului și încăperile acesteia. Serviciul este dezvoltat folosind *starter*-ul Web, ce oferă o convenție asupra configurării prin

folosirea de adnotări precum *@RestController* sau *@Service*. Acesta suportă de asemenea și *tracing* prin configurarea Zipkin, dar și rezolvarea fișierelor de configurare la momentul inițializării prin Spring Cloud Config. Acesta se înregistrează la registrul de aplicații prin clientul Eureka pentru a putea fi identificat de către Gateway. De asemenea, toate cererile sunt autentificate și autorizate prin folosirea de JSON Web Tokens, folosind același *starter* OAuth 2.0 Resource Server, autoritatea emitentă fiind tot Auth0. Pentru îmbunătățirea performanței, stratul de abstractizare a *cache*-ului oferit de către Spring Boot a fost folosit în conjuncție cu implementarea Caffeine. Pentru îmbunătățirea rezilienței la erori a soluției a fost activat *management*-ul tranzacțiilor.

Acesta se folosește de o bază de date Oracle Express Edition 21c pentru a persista datele, folosind *driver*-ul de JDBC *thin*. Schema bazei de date a fost generată folosind Hibernate, setând proprietățile aferente în fișierul de configurare a profilului *dev*, urmând a fi creat un fișier SQL la pornirea serviciului.

Următoarea secvență de cod reprezintă fișierul de configurare pentru profilul *dev*.

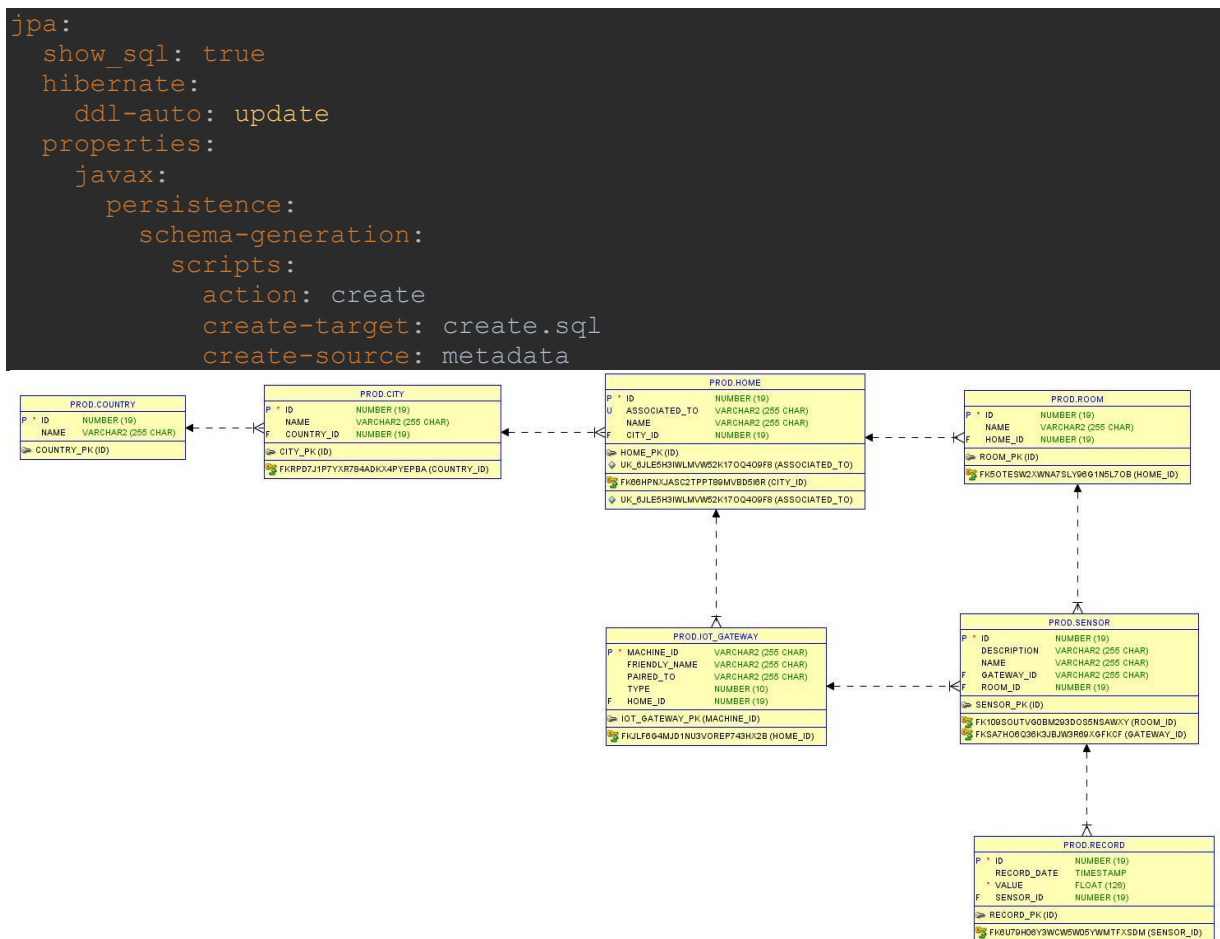


Figura 3.8. Schema OLTP

Spring Data JPA a fost folosit pentru a crea stratul corespunzător accesului la date, reducând semnificativ codul *boiler-plate* și îmbunătățind viteza de dezvoltare. Entitățile sunt reprezentate de clase adnotate cu `@Entity`, iar membri acestora pot reprezenta atât câmpuri, dar și chei primare sau de legătură. Accesul la date este facilitat prin folosirea de *repository*-uri reprezentate de interfețe ce extind tipuri definite de Spring precum *CrudRepository* sau *PagingAndSortingRepository*.

Următoarea secvență de cod definește entitatea *Record*.

```
@Data
@Entity
public class Record {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    private double value;

    @Column(name = "record_date")
    private Date date;

    @ManyToOne
    @JoinColumn(name = "sensor_id")
    Sensor sensor;
}
```

Următoarea secvență de cod definește *repository*-ul destinat entității *Record*.

```
public interface RecordRepository extends
PagingAndSortingRepository<Record, Long> {

    List<Record> findAllBySensorOrderByDateDesc(Sensor sensor, Pageable
pageable);
}
```

Microserviciul este structurat conform stilului arhitectural REST, fiind dispuse diferite *endpoint*-uri reprezentate de clase adnotate cu `@RestController`. Acestea sunt găsite și injectate de către Spring Boot în context, ulterior fiind expuse printr-un container Tomcat. Au fost create *controller*-e pentru toate entitățile, aplicându-se conceptul *Separation of Concerns*, structurând proiectul în *controller*-e, *service*-uri și *repository*-uri, îmbunătățind astfel lizibilitatea proiectului și ușurând viitoarea mentenanță. De asemenea, serviciul nu are nicio dependență exceptând pe cele de infrastructură, fiind astfel ușor de distribuit și scalat în funcție de încărcătură.

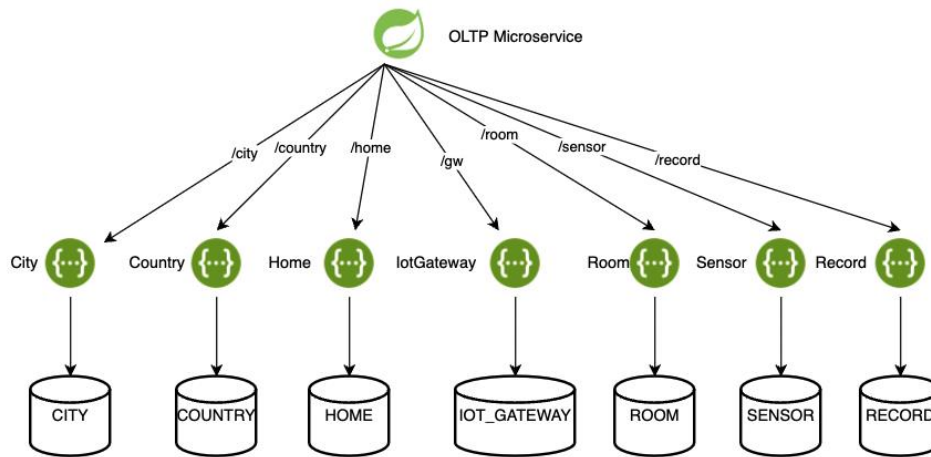


Figura 3.9. Structura serviciilor

Fluxul de folosire al acestui serviciu implică crearea de țări și orașe de către administratorul aplicației. Utilizatorul trebuie să își creeze o casă înainte de orice acțiune de împerechere cu un *gateway*. Identificarea utilizatorului se face prin *token*-ul purtător, urmând să aibă valabilă posibilitatea de salvare a unei case, regăsire a detaliilor acesteia dar și editare a numelui. Pentru a asocia o casă unui user se folosește câmpul *sub* din JWT, iar la salvarea casei se creează și o cameră implicită unde urmează a fi salvați senzorii la momentul împerecherii.

Controller-ul *IotGateway* are o metodă *signal* destinată pentru a fi apelată de către dispozitivul IoT pentru a semnala disponibilitatea acestuia. În corpul cererii se află detalii precum codul de împerechere, tipul acestuia, dar și senzorii instalați. Nodul este identificat prin intermediul câmpului *sub* al *token*-ului trimis, iar dacă nodul este deja împerecheat, este întors ID-ul utilizatorului. Detaliile acestea sunt salvate într-o structură de tip *map* concurentă, marcând astfel disponibilitatea dispozitivului de a fi împerecheat. Această metodă este apelată în buclă până va primi ID-ul unui utilizator.

Endpoint-ul *pair* este destinat pentru a fi apelat de către utilizator, trimițând codul mașinii și codul de împerechere pentru a-și însuși dispozitivul, fiind astfel scos din *pool*-ul dispozitivelor pregătite pentru împerechere. În urma acestor operațiuni, utilizatorul poate modifica numele *gateway*-ului, șterge un dispozitiv împerecheat, dar și întoarce toate *gateway*-urile împerecheate. Un *gateway* are unul sau mai mulți senzori, editabili folosind *endpoint*-ul *sensor*, aceștia putând fi mutați în alte camere definite de către utilizator prin intermediul *endpoint*-ului *room*. În privința *endpoint*-ului *record*, acesta permite salvarea de noi intrări de către dispozitivul IoT, acesta fiind identificat prin intermediul *token*-ului purtător. Este

disponibilă și întoarcerea de înregistrări către utilizator în regim paginat, identificarea fiind făcută analog ca și la *endpoint*-ul precedent.

III.5.5. Microserviciul *Data Warehouse*

Microserviciul de *Data Warehouse* existent la calea *dw* este introdus în proiect pentru a întruni nevoia de stocare a volumului mare de date istorice asupra cărora se pot dezvolta rapoarte și analize. Acesta folosește o instanță de bază de date Oracle Express Edition 21c, unde s-au agregat datele relevante din schema OLTP.

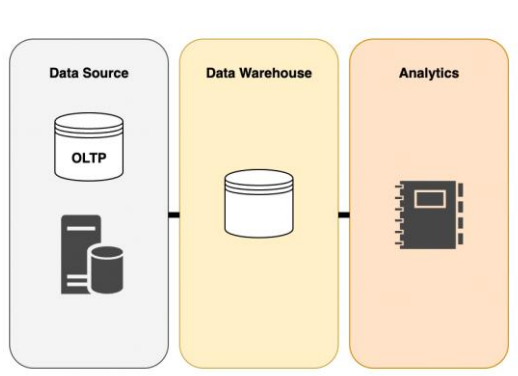


Figura 3.10. Flow-ul datelor

Pentru modelarea depozitului de date s-a folosit proiectarea schemei sub formă de stea, având în vedere performanța pe care o aduce stocarea întregii informații într-o singură înregistrare [11].

Centrul acesteia constă într-o tabelă de fapte, *Record*, aceasta fiind subiectul principal al aplicației și sursa de generare a diferitelor analize. Tabelele dimensiune ce stochează informația adițională tabelului de fapte sunt: *Sensor*, *Address*, *Home*, *Time*.

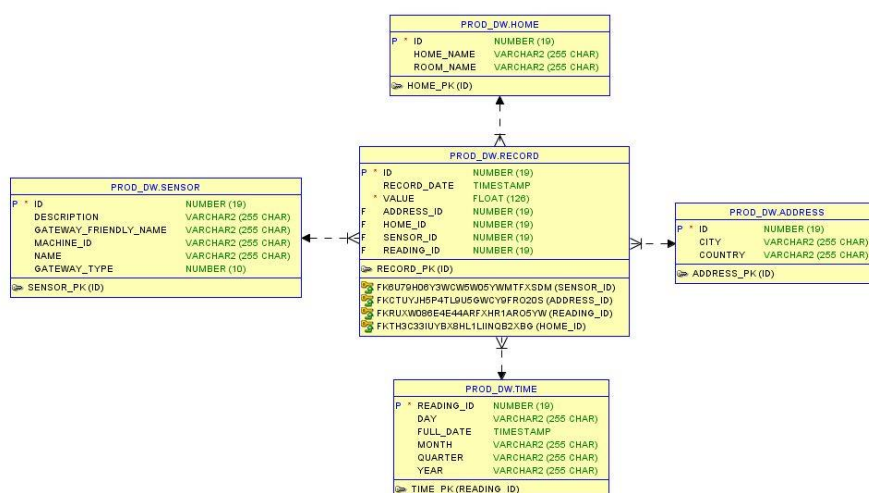


Figura 3.11. Schema bazei de date depozit

Toate tabelele dimensiune au fost rezultate din operații de JOIN, denormalizând schema OLTP astfel:

Tabela *Sensor* stochează datele cu privire la senzor *id*, *name*, *description*, dar regăsește și informațiile *Gateway*-ului cu care este asociat. Așadar, aceasta are 3 atribute suplimentare *machine_id*, *gateway_friendly_name*, *type*, aduse prin relația dintre *Sensor* și *Gateway* din OLTP.

Tabela *Address* menține informații cu privire la oraș și țară, având atributele *id*, *city*, *country*, fiind o asociere între tabelele *City* și *Country* din OLTP.

Tabela *Home* stochează date cu privire la locuință, incluzând aici atât *home_name* adus din tabela *Home*, cât și *room_name* adus din tabela *Room* din OLTP.

Tabela dimensiune *Time* este adusă în această schemă pentru a descrie temporal informația, menținând ca atribute *reading_id*, *full_date*, *year*, *month*, *day*, *quarter*.

Microserviciul de *Data Warehouse* este o aplicație Spring Boot, utilizând pentru partea de acces la baza de date Spring Data JPA. Aceasta a fost folosită pentru a avea mai multă flexibilitate în vederea creării cererilor de informații ce stau la baza generării analizelor. Schema bazei de date a fost generată folosind Hibernate, urmând a fi creat fișierul SQL al acesteia la pornirea serviciului. Acest microserviciu respectă aceleași caracteristici ca cel OLTP, permițând *tracing*-ul prin configurarea Zipkin, rezolvarea fișierelor de configurare la momentul inițializării prin Spring Cloud Config și înregistrarea acestuia prin clientul Eureka pentru a putea fi găsit de către Gateway. De asemenea, cererile acestuia sunt autentificate și autorizate folosind JSON Web Tokens prin *starter*-ul OAuth 2.0 Resource Server, iar pentru a-i oferi o performanță sporită s-a adăugat implementarea Caffeine.

Astfel, aplicația conține cereri pentru 3 analize: valoarea medie, valoarea minimă și valoarea maximă în funcție de id-ul senzorului și id-ul *gateway*-ului la care acesta trimite valorile înregistrate, id-ul senzorului fiind primit ca parametru în *path*-ul *request*-ului aferent.

Următoarea secvență de cod definește *repository*-ul destinat entității *Record*.

```
@Repository
public interface RecordRepository extends JpaRepository<Record, Long> {

    @Query(value =
        "select avg(r.value) from record r inner join sensor s on s.id
        = r.sensor_id where s.id = ?1 and s.machine_id = ?2 and r.record_date >=
        SYSDATE - 1",
        nativeQuery = true)
```

```

double getAverageValue(long sensorId, String pairedTo);

@Query(value =
    "select min(r.value) from record r inner join sensor s on s.id
= r.sensor_id where s.id = ?1 and s.machine_id = ?2 and r.record_date >=
SYSDATE - 1",
    nativeQuery = true)
double getMinValue(long sensorId, String pairedTo);

@Query(value =
    "select max(r.value) from record r inner join sensor s on s.id
= r.sensor_id where s.id = ?1 and s.machine_id = ?2 and r.record_date >=
SYSDATE - 1",
    nativeQuery = true)
double getMaxValue(long sensorId, String pairedTo);

```

Datele din *Data Warehouse* sunt colectate din OLTP folosind proceduri stocate, migrarea acestora făcându-se zilnic printr-un *job* automat, ce se rulează la ora 22:00. Procedura *migrate_data()* inserează valorile aferente pentru a popula tabelele *Address*, *Home*, *Sensor* și *Record* aplicând multiple operații de *join* pentru aceasta din urmă pentru a colecta toate valorile atributelor necesare. Procedura *insert_time()* inserează valorile coloanelor din tabela *Time*.

Procedurile stocate sunt chemate în metodele din *repository*-ul *RecordRepository*, care sunt la rândul lor apelate în metodele din *RecordService*, iar mai apoi în cele din *RecordController*, mapând fiecare apel de analiză într-un serviciu de tip *GET* și fiecare apel de inserare într-un serviciu de tip *PUT*.

Următoarea secvență de cod definește *endpoint*-urile disponibile din microserviciul *Data Warehouse*.

```

@GetMapping("/{sensorId}/avg")
public ResponseEntity<Double> getAvgValue(@PathVariable("sensorId") long
sensorId,
                                         @AuthenticationPrincipal Jwt
jwt){
    return ResponseEntity.ok().body(service.getAvgValue(sensorId, jwt));
}

@GetMapping("/{sensorId}/min")
public ResponseEntity<Double> getMinValue(@PathVariable("sensorId") long
sensorId,
                                         @AuthenticationPrincipal Jwt
jwt){
    return ResponseEntity.ok().body(service.getMinValue(sensorId, jwt));
}

@GetMapping("/{sensorId}/max")
public ResponseEntity<Double> getMaxValue(@PathVariable("sensorId") long
sensorId,
                                         @AuthenticationPrincipal Jwt jwt)

```

```

{
    return ResponseEntity.ok().body(service.getMaxValue(sensorId, jwt));
}

@PutMapping("/triggerDataMigration")
public ResponseEntity<String> callDataMigration() {
    service.migrateData();

    return ResponseEntity.ok().body("Data has been migrated
successfully.");
}

@PutMapping("/insertTime")
public ResponseEntity<String> callInsertTime() {
    service.insertTime();

    return ResponseEntity.ok().body("Time has been inserted
successfully.");
}

```

III.5.6. Microserviciul ML

Pe lângă serviciile de OLTP și *Data Warehouse*, mai sunt prezente și cele de ML și inferență care sunt într-o strânsă legătură.

Serviciul ML este dezvoltat folosind Spring Boot și este disponibil la calea *nn*, fiind de asemenea și client de Eureka ce îi permite Cloud Gateway-ului să ruteze cererea prin rezolvare la *run-time* a URL-ului. Alte aspecte notabile sunt folosirea *starter*-ului OAuth 2 Resource Server pentru a valida JWT-urile cu scopul autorizării și autentificării cererilor, rezolvarea proprietăților la *run-time* prin Spring Cloud Config. De asemenea, stratul de abstractizare a *cache*-ului împreună cu implementarea Caffeine sunt folosite pentru a îmbunătăți performanța. Includerea Spring Data este folosită pentru a facilita accesul la baza de date. Persistența este oferită de o instanță de Oracle Express Edition 21c, partajată cu serviciul *Data Warehouse* pentru a accesa datele formate conform diagramei stea. Serviciul expune un singur *endpoint*, *predict*, primind ca variabilă a căii ID-ul senzorului pentru care să fie făcută inferența. La nivelul serviciului, acesta verifică existența unui senzor cu acel ID, apoi selectează din baza de date înregistrări aferente senzorului pe ultimele 24 de ore. Acestea sunt apoi formate ca matrice cu două coloane, oră și valoare și apoi trimise către serviciul de Inferență, prin intermediul unui client Feign. Acest client permite un stil declarativ în folosirea de client REST, abstractizând detaliile ce țin de implementare.

Următoarea secvență de cod definește clientul *Feign* ce abstractizează comunicarea cu microserviciul *Flask*.

```
@FeignClient(name = "nn-client", url="${nn.uri}")
public interface NnClient {

    @RequestMapping(method = RequestMethod.POST, value = "/predict")
    List<List<Double>> predict(@RequestBody List<List<Double>> values);
}
```

Serviciul de Inferență este dezvoltat în Python folosind *framework*-ul Flask pentru a crea un server HTTPS, iar partea de *machine learning* este construită folosind Tensorflow, Keras și Sci-Kit Learn. Acest serviciu nu necesită autentificare, fiind disponibil doar în Intranet și apelat de către serviciul ML, ne-existând rută a *Gateway*-ului care să aibă legătura directă cu acest microserviciu. Există un singur *endpoint*, denumit *predict*, apelat folosind metoda HTTP POST, având ca încărcătură 24 înregistrări reprezentând valori din trecut. Ca răspuns va trimite rezultatul inferenței folosind un model și un *scaler* pre-antrenat, la care sunt folosite date prelucrate. Din matricea originală ce este caracterizată de două coloane, *ora* și *valoare*, mai sunt introduse coloanele *ora_cos*, cosinusul orei ajustate, și *ora_sin*, rezultatul funcției sinus aplicată orei ajustate, urmând a fi îndepărtată coloana *ora*.

Modelul și *scaler*-ul au fost pre-antrenate și exportate de către altă aplicație Python, fiind definit un model secvențial Keras ce reprezintă o rețea neuronală recurentă de tipul *Long Short Term Memory* (LSTM) cu 40 de unități. Aceasta este optimizată folosind Adam, bazată pe metrica de cost Mean Squared Error (MSE). Rețeaua neuronală recurentă este indicată pentru regresii efectuate pe serii de timp, permițând informațiilor să fie persistate pe parcursul buclei de dimensiune *unități*. Adicional, folosirea de LSTM evită apariția problemei de învățare în cazul dependențelor pe termen lung. După compilare, acesta a fost exportat folosind ponderi inițializate aleator folosind o distribuție normală.

Următoarea secvență de cod reprezintă structura rețelei neuronale folosite.

```
model = Sequential()
model.add(layers.Input(shape=(no_lags, no_features)))
model.add(layers.LSTM(no_layers))
model.add(layers.Dense(no_output, activation="linear"))

model.compile(optimizer="adam",
              loss="mse")
```

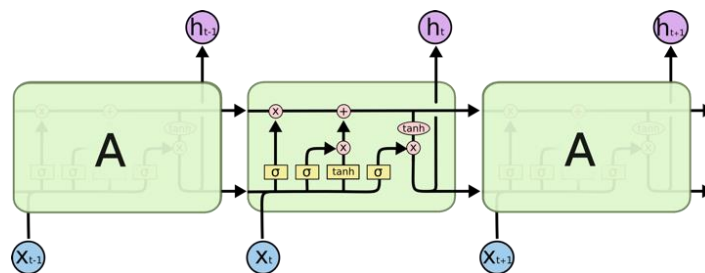


Figura 3.12. Diagrama unui strat LSTM

Modelul a fost antrenat pe un *dataset* constituit din 500 de înregistrări istorice provenite de la senzor, fiind încărcate ca *Dataframe Pandas* dintr-un fișier CSV. Acestea au fost prelucrate analog cu procesul efectuat de către serviciul de Inferență, adăugând coloanele *ora_cos* și *ora_sin* și eliminând coloana *ora*, urmând a fi ulterior separate în *dataset* destinat antrenării și *dataset* destinat testării pentru a preveni *overfitting*-ul modelului.

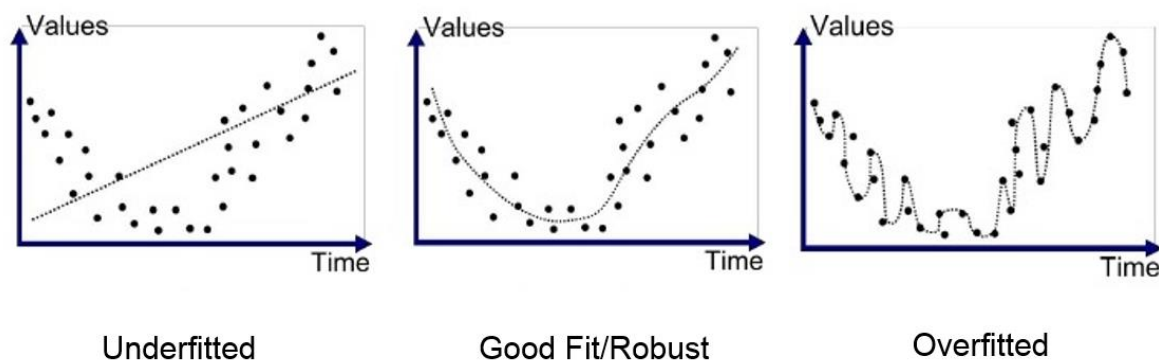


Figura 3.13. Posibile rezultate în urma antrenării

Ulterior, *scaler*-ul din Sci-Kit Learn *MinMaxScaler* este folosit pentru a plasa valorile pe o scară de la 0 la 1, acesta fiind antrenat pe datele de antrenament și aplicat atât pe setul de antrenament cât și cel test. După prelucrarea datelor, modelul exportat anterior a fost instanțiat folosind *load_model* din Keras, iar datele au fost redimensionate corespunzător. După mai multe încercări, cele mai mici valori ale funcției de cost au fost înregistrate folosind 35 de epoci cu un split de 10% din date pentru validare. În urma efectuării antrenamentului, modelul a fost evaluat pe datele de test cu rezultate bune ce sugerează un *fit* robust. Având ponderile setate și *scaler*-ul antrenat, acestea au fost exportate pentru uz ulterior.

```

Retal device set to: Apple M1
systemMemory: 8.00 GB
maxCacheSize: 2.67 GB

Epoch 1/35
11/11 [=====] - 5s 83ms/step - loss: 0.2134 - val_loss: 0.1294
Epoch 2/35
11/11 [=====] - 0s 11ms/step - loss: 0.1238 - val_loss: 0.1214
Epoch 3/35
11/11 [=====] - 0s 12ms/step - loss: 0.1121 - val_loss: 0.1175
Epoch 4/35
11/11 [=====] - 0s 11ms/step - loss: 0.1184 - val_loss: 0.1198
Epoch 5/35
11/11 [=====] - 0s 12ms/step - loss: 0.1102 - val_loss: 0.1187
Epoch 6/35
11/11 [=====] - 0s 11ms/step - loss: 0.1096 - val_loss: 0.1178
Epoch 7/35
11/11 [=====] - 0s 12ms/step - loss: 0.1095 - val_loss: 0.1176
Epoch 8/35
11/11 [=====] - 0s 11ms/step - loss: 0.1095 - val_loss: 0.1178

```

Figura 3.14. Antrenarea rețelei neuronale

```

Epoch 31/35
11/11 [=====] - 0s 12ms/step - loss: 0.1087 - val_loss: 0.1161
Epoch 32/35
11/11 [=====] - 0s 11ms/step - loss: 0.1087 - val_loss: 0.1161
Epoch 33/35
11/11 [=====] - 0s 12ms/step - loss: 0.1088 - val_loss: 0.1162
Epoch 34/35
11/11 [=====] - 0s 11ms/step - loss: 0.1092 - val_loss: 0.1163
Epoch 35/35
11/11 [=====] - 0s 12ms/step - loss: 0.1087 - val_loss: 0.1162
History: {'loss': [0.21336592733860016, 0.12381288409233093, 0.11205174028873444, 0.116
WARNING:absl:Found untraced functions such as lstm_cell_layer_call_fn, lstm_cell_layer_c
3/3 [=====] - 0s 53ms/step - loss: 0.1130
Evaluation: 0.11295798420906067

```

Figura 3.15. Evaluarea rețelei neuronale

Serviciul de Inferență folosește atât modelul cât și *scaler*-ul, pe care le încarcă la pornirea server-ului Flask. Înainte de a efectua predicția, datele sunt prelucrate conform mențiunilor anterioare, transformate de către *scaler* și redimensionate corespunzător. Inferența este apoi efectuată, iar rezultatul acesteia este readus la scară normală prin metoda *inverse_transform* a *scaler*-ului. Acest rezultat este convertit în format JSON și întors serviciului ML, care îl întoarce apoi către utilizator.

III.5.7. Microserviciul de interfață grafică

Pentru a facilita utilizarea acestei soluții, a fost dezvoltată o interfață grafică folosind *framework*-ul Next.js. Componentele folosite în crearea *view*-urilor provin din librăria Material UI. Next.js a fost ales în detrimentul unei soluții pur React pentru nivelul înalt de calitate a experienței utilizatorului oferit, pentru performanța ridicată pe care o oferă prin afișarea statică a paginilor, cât și pentru posibilitățile de optimizare disponibile pentru motorul de căutare [12].

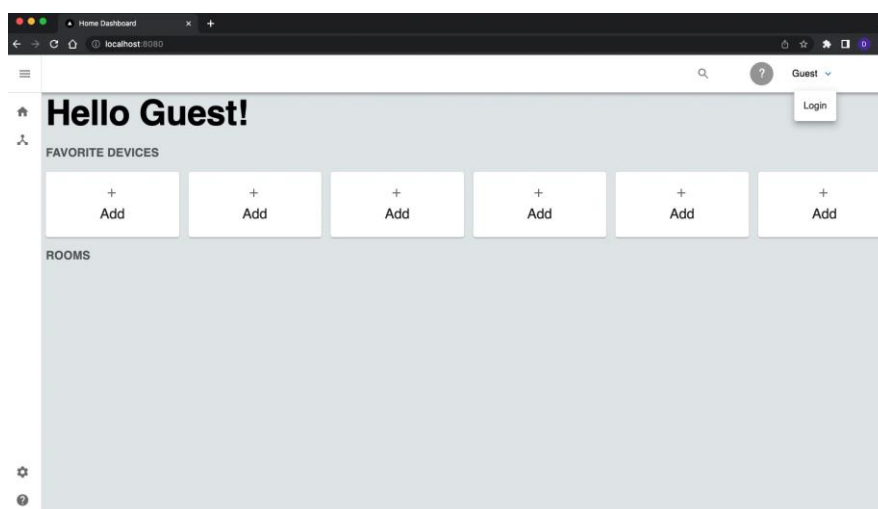
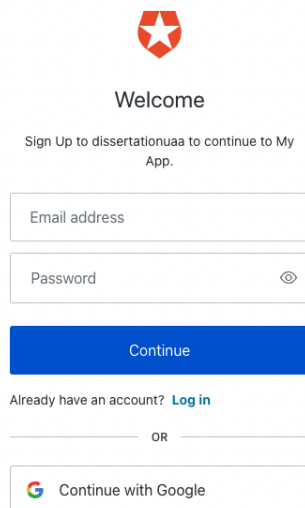


Figura 3.16. Pagina de utilizator ne-autentificat

Pagina de *dashboard* este configurată să suporte atât un utilizator ne-autentificat, cât și un utilizator autentificat. Un utilizator ne-autentificat se va putea conecta în aplicație selectând

funcția de *Login*, fiind astfel redirecționat către platforma Auth0, precum în figura 3.17, unde va putea avea două opțiuni. Fie va crea un cont în cazul în care utilizatorul este nou, fie se va putea *loga* în cazul unui utilizator existent.



The image shows the Auth0 'Welcome' screen. At the top is the Auth0 logo (a red shield with a white star). Below it is the word 'Welcome'. Then, a line of text says 'Sign Up to dissertationuaa to continue to My App.' There are two input fields: 'Email address' and 'Password' (with an eye icon for toggling visibility). Below these is a blue 'Continue' button. Under the button, it says 'Already have an account? [Log in](#)'. At the bottom, there is a horizontal line with 'OR' in the center, and below that is a button with the Google logo and the text 'Continue with Google'.

Figura 3.17. Redirecționare către Platforma Auth0



Figura 3.18. Opțiuni utilizator autentificat

Odată ce utilizatorul s-a autentificat, revenind în pagina *dashboard* aceasta va prelua denumirea utilizatorului (figura 3.18). De asemenea, vor fi disponibile variantele de vizualizare a profilului și de *delogare* a utilizatorului.

Pagina de *settings* ilustrată în figura 3.19 permite utilizatorului autentificat să își modifice setările cu privire la casă și camere. Apăsând *grid*-ul *Home* se va deschide *grid*-ul ce îi va permite utilizatorului să creeze o casă, iar în cazul în care casa este deja creată, aceasta se poate redenumi.

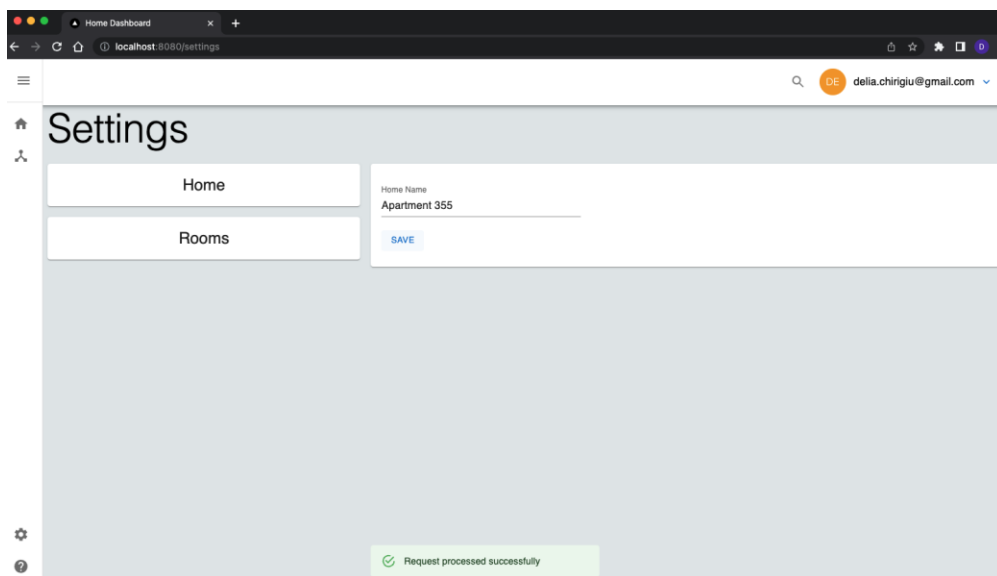


Figura 3.19. Pagina Settings

În același fel, apăsând *grid-ul Rooms*, *grid-ul* ce permite modificarea locuinței se va închide și se va deschide *grid-ul* ce permite gestionarea camerelor. Acesta vine preîncărcat cu camera *Default*, aceasta fiind necesară asocierii unui dispozitiv înainte de a-i alege camera. În continuare, însă, se pot adăuga camere noi (figura 3.20) și se pot șterge, respectiv modifica (redenumi) camere existente (figura 3.21).

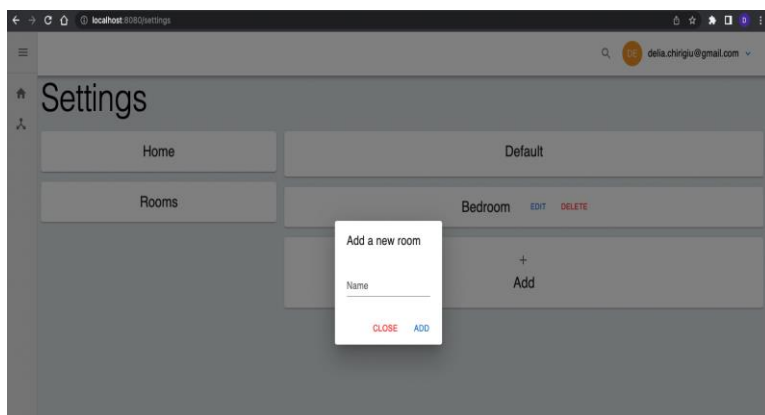


Figura 3.20. Adăugarea unei noi camere

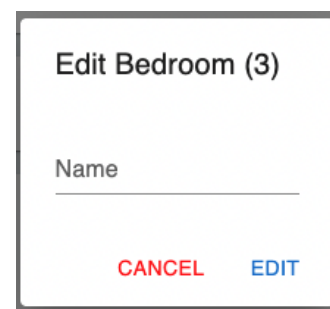


Figura 3.21. Modificarea camerei

Odată introdusă o cameră, aceasta va putea fi vizualizată atât în pagina de setări cât și în *dashboard* (figura 3.22), specificând de asemenea și numele locuinței din care aceasta face parte.

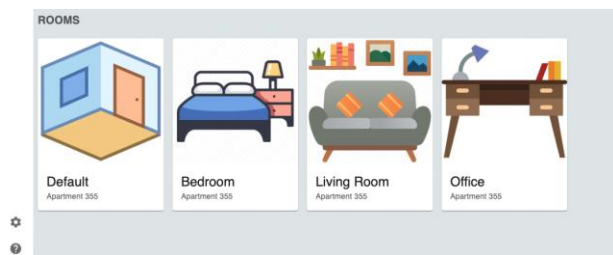


Figura 3.22. Vizualizarea camerelor asociate casei

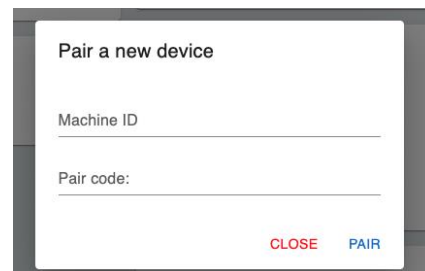


Figura 3.23. Împerecherea cu un device

Pagina de *devices* permite utilizatorului să își adauge un *device* IoT folosind un *machine* ID, acesta fiind id-ul *gateway*-ului și codul de împerechere cu acesta (figura 3.23). Tot aici se pot redenumi *device*-urile și se vor afișa informațiile utile precum temperatura și umiditatea actuale ce sunt preluate din schema OLTP, cea medie în urma ultimelor 24 ore, cea maximă și cea minimă, ultimele trei fiind preluate folosind serviciile aflate în aplicația bazei de date depozit, precum în figura 3.24.

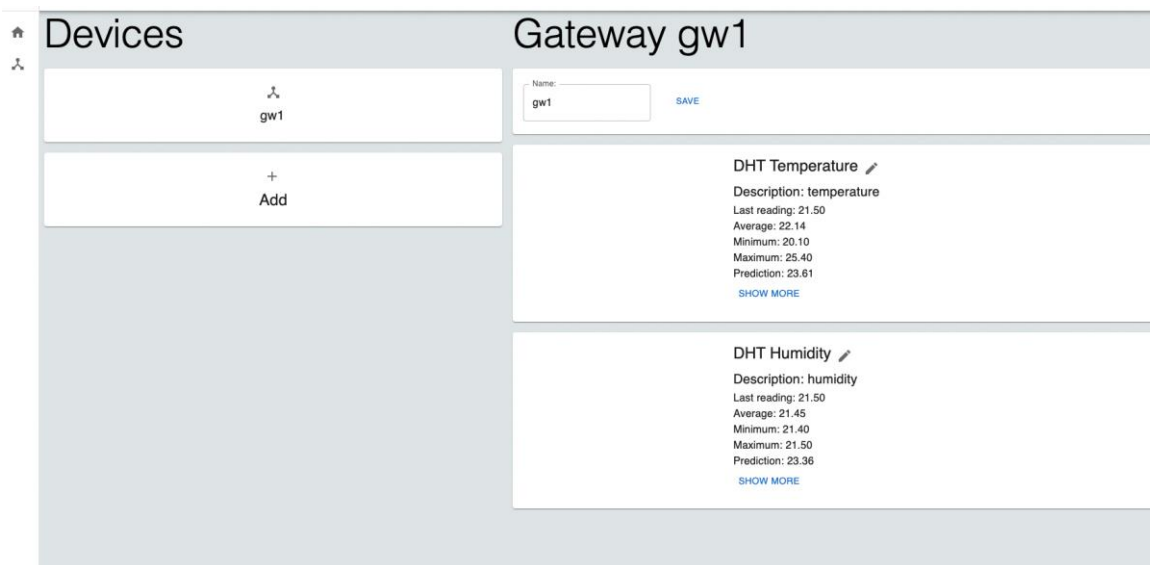


Figura 3.24. Pagina de urmărire date

Odată adăugat un *device*, în dreapta vor apărea *grid*-urile cu privire la senzorii conectați la acesta (figura 3.24). Pe lângă cererile din depozitul de date și OLTP, este prezentă și valoarea prezisă pentru următoarea oră în ceea ce privește capacitatea senzorului. Aceasta este oferită prin apelarea *endpoint*-ului din microserviciul de ML.

Tot în această pagină, apăsând butonul *Show More*, pentru fiecare senzor se va deschide un grafic cu privire la datele istorice ale acestuia, prezentând valoarea pe axa Y și ora la care aceasta a fost citită pe axa X. Librăria *react-chartjs-2* a fost folosită pentru crearea acestor

grafice, sursa datelor fiind baza de date OLTP, iar cererile întorcând date istorice recente (figurile 3.25 și 3.26).

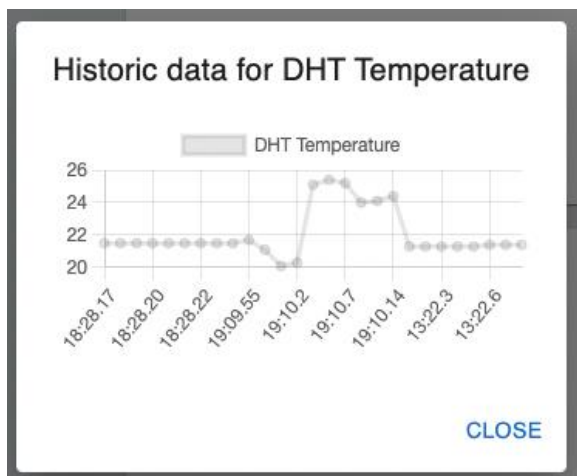


Figura 3.25. Grafic înregistrări temperatură

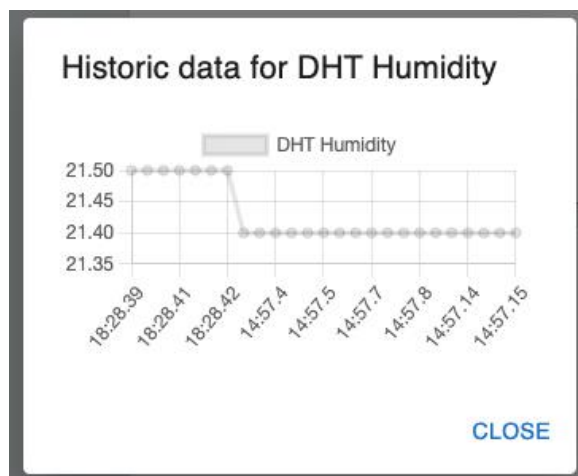


Figura 3.26. Grafic înregistrări umiditate

În pagina principală *dashboard*, utilizatorul își poate adăuga dispozitivele împerecheate în lista de dispozitive preferate. Aceasta este o listă persistată local. Utilizatorul are opțiunea de a adăuga maximum 6 device-uri în lista de favorite. Apăsând butonul de *Add* se va deschide dialogul în care se poate selecta dispozitivul dorit (figura 3.27). În urma procesului de adăugare, acesta va apărea în *grid*-ul ales, specificând denumirea acestuia precum în figura 3.28.

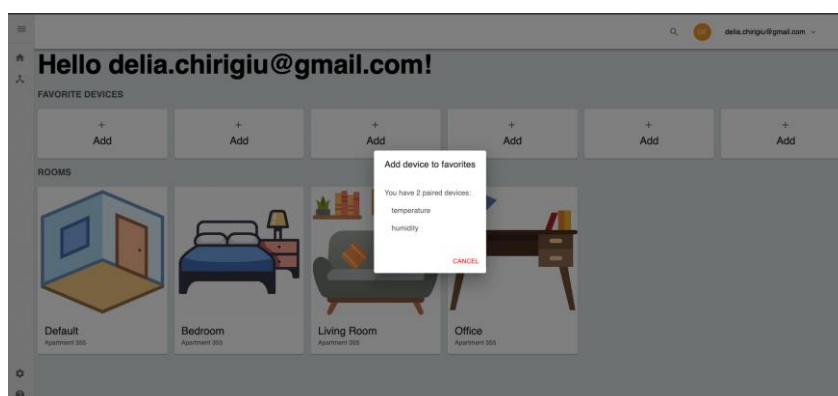


Figura 3.27. Adăugarea unui device în lista de favorite

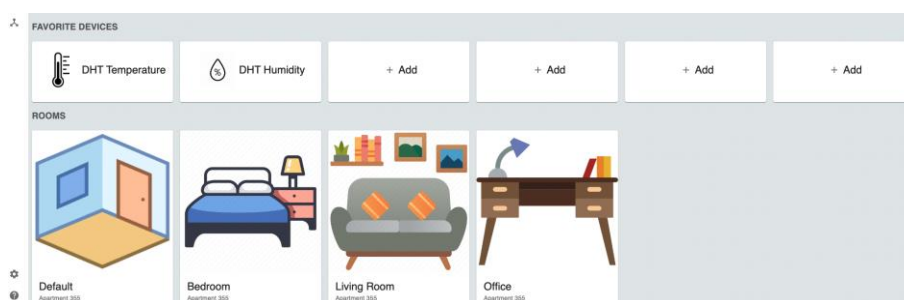


Figura 3.28. Vizualizarea dispozitivelor favorite

Concluzii

În concluzie, proiectarea unei soluții IoT modulare și extensibile este posibilă folosind tehnologii disponibile comercial pentru un preț redus, fapt confirmat și prin expansiunea rapidă a acestui segment.

Din punct de vedere al contribuției personale, aceasta se regăsește atât în dezvoltarea arhitecturii soluției, cât și în dezvoltarea aplicațiilor. Astfel, protocolul MQTT a fost folosit pentru a eficientiza distribuirea datelor până la centralizarea lor, continuând apoi cu distribuirea lor folosind HTTPS pentru a avea o arhitectură de tip RESTful. De asemenea, din considerente de securitate s-a ales utilizarea în cloud a unui *gateway* prin care informația va trece și va fi apoi distribuită către restul microserviciilor. În ceea ce privește volumul mare de informații, scopul microserviciului de *Data Warehouse* este de a stabiliza performanța cererilor necesare rapoartelor existente sau ulterioare. Un alt aspect de menționat este reprezentat de modularitatea și scalabilitatea soluției datorate arhitecturii acesteia.

Soluția finală este constituită din mai multe noduri IoT reprezentate de plăci de dezvoltare ESP8266 având senzori atașați care transmit date prin intermediul protocolului MQTT către *gateway*-ul IoT pentru a fi centralizate și trimise în *cloud*. *Gateway*-ul IoT este constituit de un Raspberry Pi 4 ce găzduiește un *broker* Mosquitto, dar și o aplicație NodeJS ce colectează datele publicate pe topicul *data* și apoi le trimite în *cloud* prin intermediul HTTPS POST. Găzduirea *broker*-ului local a fost aleasă atât din considerente de securitate, nemaifiind necesară securizarea încărcăturii nici la nivel transport, nici la nivel aplicație, dar și de consum de energie, protocolul MQTT consumând mai puține resurse decât cel HTTP.

Multiple microservicii au fost dezvoltate, găzduite într-o instanță Oracle Cloud cu arhitectura ARM, fiind expus doar serviciul Cloud Gateway, restul serviciilor de infrastructură, dar și cel OLTP, *Data Warehouse* sau *Machine Learning* fiind accesibile doar din interiorul rețelei. Serviciile dezvoltate sunt *cloud-native*, dar și destinate rulării în containere Docker, permițând lansarea pe noi *host*-uri în doar câteva minute. Această soluție include și componenta de *machine learning* ce permite predicția de noi valori pe baza unei serii de timp, inferența fiind posibilă prin apelarea serviciului Flask. Toate cele 3 servicii funcționale Spring Boot sunt legate la o bază de date Oracle Express Edition 21c, găzduită de asemenea în *Cloud*, însă pe altă instanță Oracle cu arhitectura x86-64, fiind singura platformă suportată de către imaginea Docker oficială.

Accesul la funcționalitățile acestei soluții este facilitat de o interfață grafică Next.js servită de asemenea prin intermediul lui Cloud Gateway, ce permite crearea de locuințe, camere, împerecherea de *gateway*-uri IoT, vizualizarea de date istorice, afișarea de statistici din serviciul *Data Warehouse*, dar și predicții oferite de către serviciul *Machine Learning*.

Posibila extensie a acestei soluții poate fi reprezentată de adăugarea de noi modele de *Machine Learning*, dar și de dezvoltarea mai multor *endpoint*-uri în serviciul de *Data Warehouse* pentru a genera diverse rapoarte și statistici. De asemenea, soluția poate fi ușor extinsă prin adăugarea de noi senzori meniți să capteze alte tipuri de date, pentru care s-ar putea, de asemenea, extinde restul modulelor pentru generarea de noi rapoarte și analize, dar și pentru simpla evidențiere de informații.

Bibliografie

- [1] Cisco, „Cisco Annual Internet Report (2018–2023) White Paper,” 9 Martie 2020. [Interactiv]. Disponibil: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>. [Accesat 10 Martie 2022].
- [2] A. Domingus, „Medium,” 02 Mai 2020. [Interactiv]. Disponibil: <https://adriennedomingus.medium.com/distributed-systems-an-introduction-to-publish-subscribe-pub-sub-6bc72812a995>. [Accesat 07 Aprilie 2022].
- [3] MQTT, „MQTT,” [Interactiv]. Disponibil: <https://mqtt.org/>. [Accesat 07 Aprilie 2022].
- [4] Docker, „Docker Docs,” [Interactiv]. Disponibil: <https://docs.docker.com/get-started/overview/>. [Accesat 22 Aprilie 2022].
- [5] A. Geron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, O'Reilly, 2017.
- [6] Colah, „Understanding LSTM Networks,” 27 August 2015. [Interactiv]. Disponibil: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>. [Accesat 27 Aprilie 2022].
- [7] A. Parajuli, „The IoT Projects,” 18 Aprilie 2020. [Interactiv]. Disponibil: <https://theiotprojects.com/esp8266-dht11-dht22-temperature-humidity-with-local-web-server>. [Accesat 18 Martie 2022].
- [8] Espressif, „ESP8266 Configuration,” [Interactiv]. Disponibil: <https://arduino-esp8266.readthedocs.io/en/latest/ideoptions.html#note-about-platformio>. [Accesat 15 Martie 2022].
- [9] auth0, „auth0,” [Interactiv]. Disponibil: <https://auth0.com/docs/>. [Accesat 20 Aprilie 2022].
- [10] baeldung, „Baeldung,” [Interactiv]. Disponibil: <https://www.baeldung.com/spring-cloud-netflix-eureka>. [Accesat 01 Aprilie 2022].

[11] Oracle, „Data Warehousing Guide,” [Interactiv]. Disponibil: https://docs.oracle.com/cd/A87860_01/doc/server.817/a76994/schemas.htm. [Accesat 29 Mai 2022].

[12] Next.js, „Next.js,” [Interactiv]. Disponibil: <https://nextjs.org/docs/getting-started>. [Accesat 10 Mai 2022].