



Universitatea Politehnica București
Facultatea de Automatică și Calculatoare
Departamentul de Automatică și Ingineria Sistemelor

Sistem de achiziție, procesare și distribuție a datelor

Absolvent
Petrișor-Ștefan Lăcătuș

Coordonator
Conf. Dr. Ing. Andreea Udrea

Septembrie 2015

Cuprins

1	Introducere	1
2	Arhitectura soluției	3
2.1	Prezentare generală	3
2.1.1	Baza de date	5
2.1.2	Aplicația Java	6
2.2	Entități	6
2.2.1	Punctul de date	6
2.2.2	Canalul de date	7
2.2.3	Blocul de intrare	8
2.2.4	Blocul de procesare	9
2.2.5	Diagrama funcție bloc (FBD)	11
2.3	Primirea datelor	12
3	Interfața aplicației	15
4	Implementarea aplicației de achiziție, procesare și distribuție a datelor (Ale-	20
	gria)	
4.1	Interfața Web	20
4.1.1	Securitate	21
4.1.2	Management	23
4.1.2.1	Managementul blocurilor de intrare	24
4.1.2.2	Managementul blocurilor de procesare	25
4.1.2.3	Managementul diagramelor	26
4.1.2.4	Administrare tag-uri	27
4.1.3	Monitorizare	29
4.2	API-ul aplicației	30
4.3	Executarea unui bloc de procesare	31
4.4	Executarea unei diagrame funcționale	33
4.4.1	Ordinea execuției blocurilor	33

4.4.2	Generarea rezultatului	35
4.4.3	Implementarea execuției	36
4.5	Serviciul de date	37
4.6	Baza de date	38
4.6.1	Stocarea entităților	38
5	Exemplu de utilizare: Smart Home	41
6	Concluzii și Dezvoltării ulterioare	44
6.1	Concluzii	44
6.2	Dezvoltării ulterioare	44
	Bibliografie	45

Capitolul 1

Introducere

Într-o lume emergentă, a dispozitivelor inteligente și cu puternice capacități de conectare, problema folosirii în mod corect și eficient a datelor devine din ce în ce mai importantă. Mai mulți analiști în domeniu estimează creșteri amețitoare ale numărului de dispozitive: $5 \cdot 10^9$ până în 2020 și 10^{12} până în 2035 [6]. Deși aceste estimări sunt foarte apropiate de certitudini, ce este însă neclar, este modul în care aceste dispozitive vor fi folosite pentru a realiza aplicații conectate, care înglobează date din sisteme complet diferite.

Problema poate fi generalizată în următoarele trei concepte:

- **Colectarea de date:** Tot mai multe dispozitive folosesc volume mari de date în formate variate. Aceste date trebuie stocate în mod eficient folosind instrumente de stocare scalabile.
- **Prelucrarea datelor:** Date din surse variate trebuie combinate pentru a obține informații. Aici sunt incluse atât datele stocate local, cât și datele disponibile din alte sisteme și servicii ce trebuie interogate. Din aceste date se urmărește generarea de date noi, cu valoare și semnificație mai mare.
- **Conectarea:** Datele procesate trebuie făcute accesibile pentru sisteme exterioare.

Problema aceasta se întâlnește atât în mediile industriale, cât și în aplicațiile de mici dimensiuni. În ultimii 5 ani, a avut loc o nouă revoluție industrială [18]. Paradigma industrială clasică este înlocuită la o paradigmă nouă în care procesele de fabricație sunt digitalizate complet, toate componentele procesului de fabricație fiind conectate central [10]. Tradițional, această interconectare se face prin intermediul automatelor programabile, conectate la un sistem de tip SCADA. Această structură are totuși dezavantajul că este statică și nu valorifică în totalitate existența unor senzori inteligenți, iar fiecare automat trebuie programat individual pentru un singur scop, datele achiziționate fiind greu de centralizat.

Pe de altă parte, pentru aplicațiile din mediul privat, sistemele care centralizează și procesează datele sunt fie foarte scumpe, restricționând utilizatorul la senzori și dispozitive produse de un singur producător, fie sunt greu de configurat, necesitând multe echipamente separate de procesare. Spre exemplu, o automatizare a casei, folosind echipamente Siemens, costa aproximativ 15000€, dispozitivele de procesare reprezentând jumătate din acest preț. Pe de altă parte, pachete software open-source, precum OpenHAB [16] rezolvă un subset al problemei, ușurând integrarea echipamentelor compatibile, însă nu sunt destul de generice, având aplicabilitate într-un singur domeniu.

Soluția propusă dorește să răspundă la problema aplicațiilor de dimensiuni mici și mijlocii, în special din mediul privat care, propunând o alternativă la achiziționarea de PLC-uri sau alte dispozitive complicate pentru stocarea și procesarea datelor. Aceasta unifică achiziția, procesarea, stocarea și distribuția datelor într-o singură aplicație ce poate fi instalată atât local, cât și pe o instanță cloud, permițând accesul din orice locație. Un alt avantaj al unificării tuturor acestor funcționalități într-o singură aplicație este prezentarea unui interfață standard pentru implementarea proceselor de procesare. Astfel, se evită unul din cele mai mari avantaje ale automatelor programabile, la care, fiecare producător are propriile standarde de implementare a aplicațiilor.

Una din cerințele inițiale pe care aplicația a urmărit să le respecte este aspectul generic al datelor. Deși majoritatea aplicațiilor vor fi în procesarea numerică, acesta este doar unul din mai multe tipuri de date. Acest aspect generic permite folosirea soluției ca o magistrală de procesare a datelor în medii de lucru, care transformă date complexe produse de un sistem în date ce pot fi acceptate de alt sistem, menținând un istoric în tot acest timp.

Următoarele funcționalități sunt încuse în aplicație:

- Colectarea datelor de la dispozitive și servicii.
- Implementarea funcțiilor de procesare folosind limbaje de nivel înalt.
- Realizarea procesării folosind diagrame funcție bloc.
- Posibilitatea de integrare în alte sisteme.
- Monitorizarea în timp real.

Lucrare urmărește prezentarea implementării acestei soluții. Capitolul 2 prezintă arhitectura generală, descriind entitățile folosite și modul în care acestea interacționează. Capitolul 3 ilustrează interfața vizuală pe care aplicația o pune la dispoziția utilizatorului, iar capitolul 4 descrie în detaliu implementarea. Ultimele două capitole, 5 și 6, prezintă un caz de utilizare al aplicației respectiv câteva concluzii și dezvoltări ce ar putea îmbunătăți soluția propusă.

Capitolul 2

Arhitectura soluției

2.1 Prezentare generală

Sistemul de achiziție, procesare și distribuție a datelor a fost conceput ca o platformă de dezvoltare rapidă, bazat pe modele. Utilizatorul poate să își concentreze resursele asupra soluției finale, abstractizând detaliile implementării prin folosirea unor metode de programare vizuală, cu blocuri refolosibile în mai multe aplicații diferite. Aplicația a fost construită pe baza unei arhitecturi modulare; modulele fiind cât mai puțin interconectate, permițând modificări rapide și testarea modulelor individual. Urmărind această gândire modulară s-au identificat 4 componente esențiale:

- **Baza de date** Asigură stocarea datelor și a entităților existente;
- **Interfața web pentru management** O interfață ușor de utilizat ce permite utilizatorului să manipuleze canale, diagrame, blocuri de procesare, dar și alți utilizatori
- **API-ul pentru date** Un API specializat în achiziția de date și care permite informarea altor dispozitive cu privire la apariția unor date noi.
- **Elemente de procesare** Asigură procesarea datelor, atât în cadrul blocurilor de procesare, cât și în cadrul diagramelor funcționale.

În vederea implementării sistemului s-au identificat următoarele elemente componente esențiale, componente ce reprezintă elementele constructive a sistemului. Acestea au reprezentare atât în baza de date, ca entități, cât și în aplicația Java, ca clase. Identificarea acestor elemente s-a făcut pe baza analizei cazurilor de utilizare ale produsului în care s-au investigat metodele prin actorii interacționează cu sistemul.

- **Canalul de date** Reprezintă elementul de bază a sistemului ce asigură recepția, persistența și emiterea de date. La crearea canalului, datele dintr-un canal, trebuie

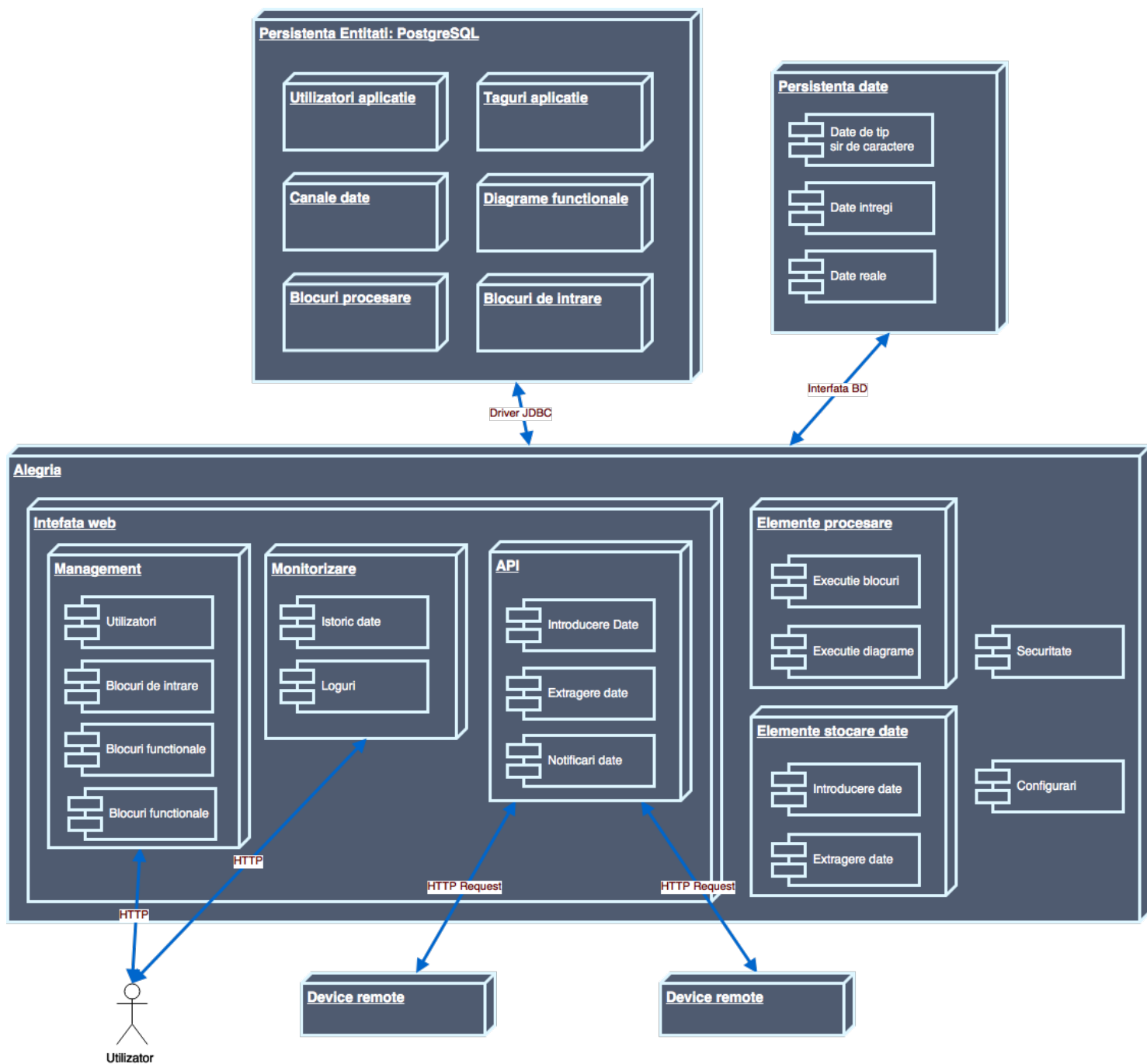


Figura 2.1: Arhitectura generală a aplicației

să respecte un format prestabilit. Pentru transformarea datelor în formatul stabilit se poate introduce un bloc de pre-procesare care transformă datele din formatul brut în formatul standard;

- **Blocul de intrare** Elementul ce mapează un element din lumea reală în interiorul platformei. Blocurile de intrare permit gruparea mai multor canale de date într-o structură unică;
- **Blocul de procesare** Elementul dinamic al aplicației ce aplică transformări asupra datelor. Un bloc de procesare primește ca intrări mai multe canale de date și are la ieșire un alt canal de date. Utilizatorul poate folosi blocuri standard, existente în sistem, sau poate implementa blocuri noi direct în interfața programului;
- **Diagrama funcție bloc(FBD)** Folosește blocuri de intrare, canale de date și blocuri de procesare pentru a descrie o funcție complexă cu 0-n intrări și o ieșire. Aceste diagrame folosesc datele - aflate pe canalele de date - ce sunt trimise către blocurile de procesare și la final se obține un singur rezultat care este salvat pe un canal de date.

2.1.1 Baza de date

Fiind vorba despre o aplicație puternic bazată pe date, aceasta are nevoie de un nivel de persistență de înaltă performanță. Urmărind arhitectura propusă din figura 2.1, putem identifica două cazuri de utilizare pentru baza de date:

- Stocarea modelului entităților: fiecare entitate descrisă în lista de mai sus trebuie stocată în baza de date, într-o structură relațională. Entitățile sunt puternic interconectate și de aceea este recomandată o bază de date relațională, de tip SQL. Din punct de vedere al dimensiunii setului de date, chiar și în aplicațiile de mare complexitate, este vorba despre câteva milioane de înregistrări. Factorul care face acest număr să crească este conectarea a tot mai multor dispozitive, ce duce la din ce în ce mai multe canale de date. Astfel, stocarea entităților nu va aduce probleme de performanță.
- Stocarea datelor: în baza de date vor fi stocate atât datele primite pe fiecare canal asociat unui bloc de intrare, cât și datele procesate de diagrame. Aceste date au un puternic caracter istoric, reprezentând o serie de timp în care se reține valoarea fiecărui punct de date la un anumit moment. Problema stocării acestor date este una mai complicată datorită necesității unei puternice scalări a bazei de date. Această problemă reprezintă un caz de utilizare pentru o baza de date NoSQL sau chiar o bază de date specializată în stocarea seriilor de timp [28].

2.1.2 Aplicația Java

Legătura dintre baza de date și utilizatorii finali se face prin intermediul unei aplicații Java complexe, care este obiectul acestui proiect. Aplicația conține toată logica platformei, de la operații asupra entităților din baza de date, la adăugarea, extragerea și procesarea datelor. Separarea modulelor s-a făcut pe baza scopului acestora astfel:

- **Administrare:** pentru administrarea entităților din baza de date. Aceste module permit atât operații de căutare și afișare, cât și de creare, editare și ștergere a utilizatorilor, a blocurilor de intrare și funcționale, și a diagramelor. Fiecare dispune de o interfață HTML5;
- **Monitorizare:** permite monitorizarea execuției aplicației de la vizualizat loguri pentru a diagnostica probleme până la realizarea de grafice folosind datele de pe anumite canale. Tot aici este disponibilă și funcția de a exporta date în formate uzuale (CSV, fișiere Microsoft Excel, etc);
- **API:** aplicația dispune și de un API pentru a fi folosit programatic de către alte aplicații externe. Acesta este format din două componente: servicii pentru administrarea entităților și servicii pentru adăugarea și extragerea datelor;
- **Elemente de procesare:** sunt împărțite în două subcategorii: cele pentru procesarea blocurilor de intrare și de procesare, și cele pentru procesarea diagramelor;
- **Elemente stocare date:** permit interfațarea cu sursele de date. Prin intermediul unei interfațe abstractă, acestea asigură servicii de introducere și extragere a datelor care nu țin cont de modul în care baza de date este implementată;
- **Alte module:** asigură, printre altele securitatea aplicației.

2.2 Entități

2.2.1 Punctul de date

Punctul de date reprezintă elementul constructiv al sistemul reprezentând obiectul procesării, stocării și distribuției.

Sistemul acceptă intern date în următoarele formate, ilustrate și în figura 2.2:

- **Long:** numere în intervalul $(-2^{63}, 2^{63} - 1)$, fără virgulă; folosește Long;

- **Real:** numere cu virgulă, având dublă precizie, reprezentate pe 64-bit conform standardului [12] IEEE 754; folosește Double pentru reprezentare internă;
- **Sir de caractere:** Un șir de caractere fără lungime limită ce trebuie formatat conform [26].
- **Obiect:** Un obiect Java serializat în text. Intern, asemănător cu tipul de date String.

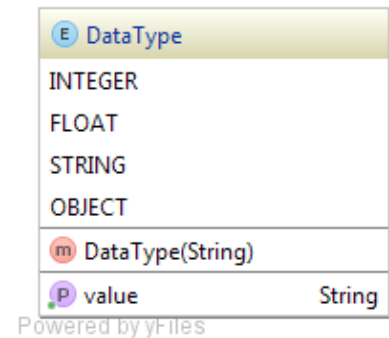


Figura 2.2: Tipurile de date acceptate în sistem

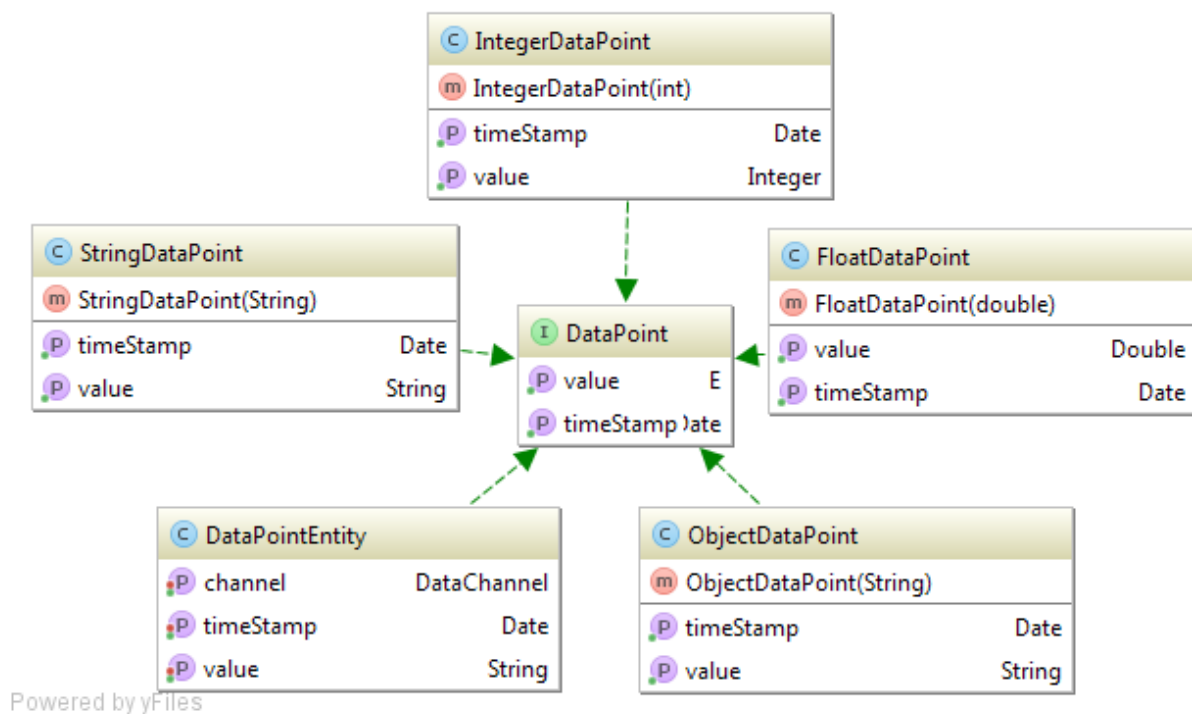


Figura 2.3: Clasele care implementează interfața DataPoint

2.2.2 Canalul de date

Canalul de date este entitatea care asigură "curgerea" datelor prin sistem. Orice punct de date din sistem aparține unui canal, acest lucru fiind realizat drept constrângere atât la nivelul aplicației, cât și la nivelul bazei de date.

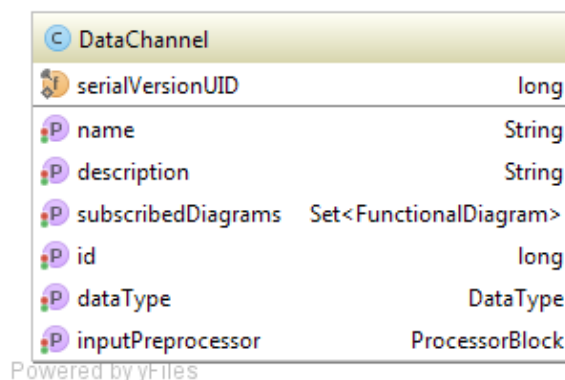


Figura 2.4: Clasa DataChannel

Canalul de date este și mijlocul prin care utilizatorul interacționează cu punctele de date. Când un dispozitiv adaugă date noi în sistem, acestea sunt atașate unui canal care poate fi folosit ca una dintre intrările unei diagrame de blocuri funcționale. De asemenea, datele procesate de o diagramă sunt atașate unui canal permițând apoi accesul pentru extragerea datelor deja existente și pentru a primi notificări de fiecare dată când pe un canal apar informații noi.

În plus, un canal de date poate avea și un bloc de preprocesare atașat. Acesta este executat de fiecare dată când date noi încearcă să fie introduse în canal, permițând validarea și transformarea datelor brute în date ce respectă tipul de date al canalului.

2.2.3 Blocul de intrare

Blocurile de intrare modelează elemente reale din aplicație, grupând mai multe canale de date și expunându-le pentru a permite introducerea datelor din exterior. Acestea descriu modul în care datele sunt legate de un element real.

[iulia] Spre exemplu, o sursă inteligentă și conectată în alimentare neîntreruptibilă din figura 2.6 poate fi privită ca un bloc de intrare, unde fiecare senzor este reprezentat de un canal de date. [iulia] Dispozitivul inteligent devine astfel conectat la platforma și poate trimite date către aceasta. Pentru că datele pot avea perioade de eșantionare diferite, dispozitivul trimite date de la unul sau mai mulți senzori odată, fiecare canal de date fiind tratat individual.

Un alt mod în care blocurile de intrare pot fi privite este ca obiecte, sau "Things" în cadrul Internetului Tuturor Lucrurilor (Internet of Things - IoT). Astfel, putem privi blocurile de intrare ca dispozitive ce monitorizează bătăile inimii sau activitatea celebrară, ca automobile cu rețele complexe de senzori, sau chiar aplicații de lucru ce generează date în timp real. Prin acest mijloc, dispozitive inteligente se pot conecta în platformă, fie

trimitând direct date, fie prin intermediul unui agent care se află pe dispozitiv, agent ce funcționează ca un *gateway*, implementând protocoale de comunicație proprietare.

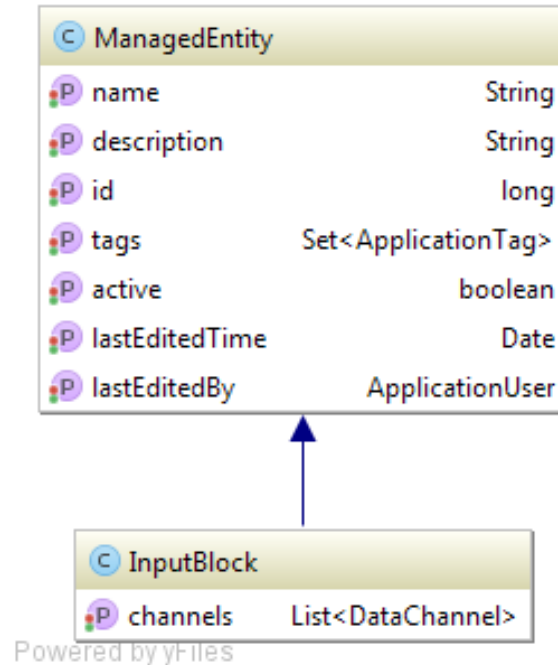


Figura 2.5: Clasa **InputBlock**

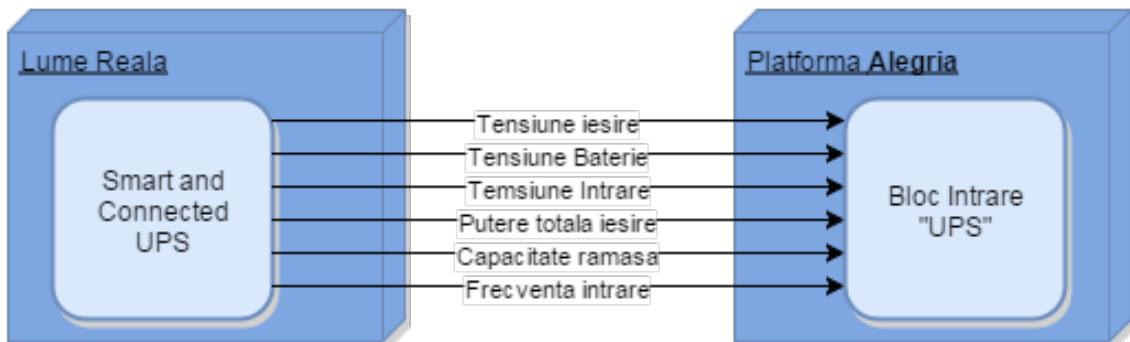


Figura 2.6: Modelarea unui UPS inteligent în Alegria

2.2.4 Blocul de procesare

Blocurile de procesare realizează operațiile de transformare a datelor. Din punct de vedere funcțional, acestea au la intrare ultimele date introduse de pe mai multe canale și returnează un singur punct de date, comportându-se ca un element de tip "black-box"[7]. Pentru implementare, utilizatorul folosește limbaje dinamice, precum JavaScript sau Ruby, scriind cod direct în interfața web. Acest cod este rulat de către server. Conceptul de blo-

curi de procesare reprezintă o implementare a blocurilor de operații definite în standardul IEC61131-3 [20, Apendix C] .

Blocuri pre-implementate, existente în orice instanță a platformei, permit rezolvarea de probleme complexe cu cunoștințe minime de programare. Un alt aspect important al blocurilor de procesare, este faptul ca sunt independente de sursa de date, ele oferind doar mijloace de prelucrare a unor date abstracte. Această abstractizare permite refolosirea lor în mai multe proiecte. Spre exemplu, un bloc care face media tuturor intrărilor nu ține cont de originea datelor.

Blocurile **nu au memorie statică**. Ele pot folosi informații exterioare, cum ar fi timpul curent al zilei sau alte informații din sistem, sau pot chiar accesa servicii web, însă ele nu au stare interioară. Astfel, se poate considera că blocurile reprezintă un element combinațional și nu unul secvențial.

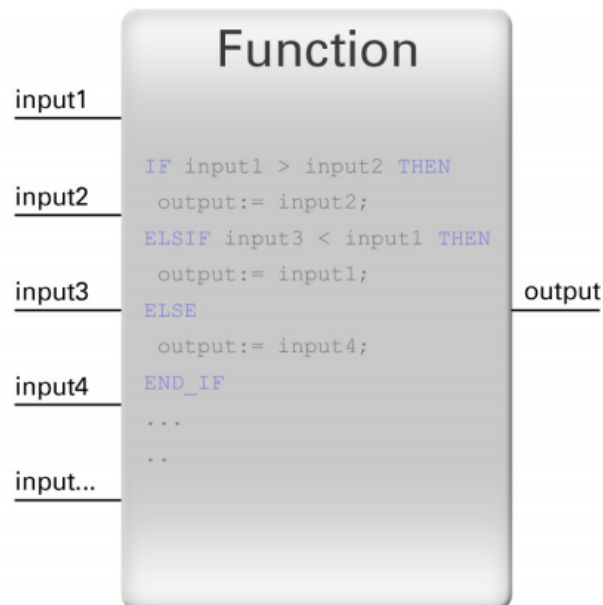


Figura 2.7: Exemplu bloc de procesare

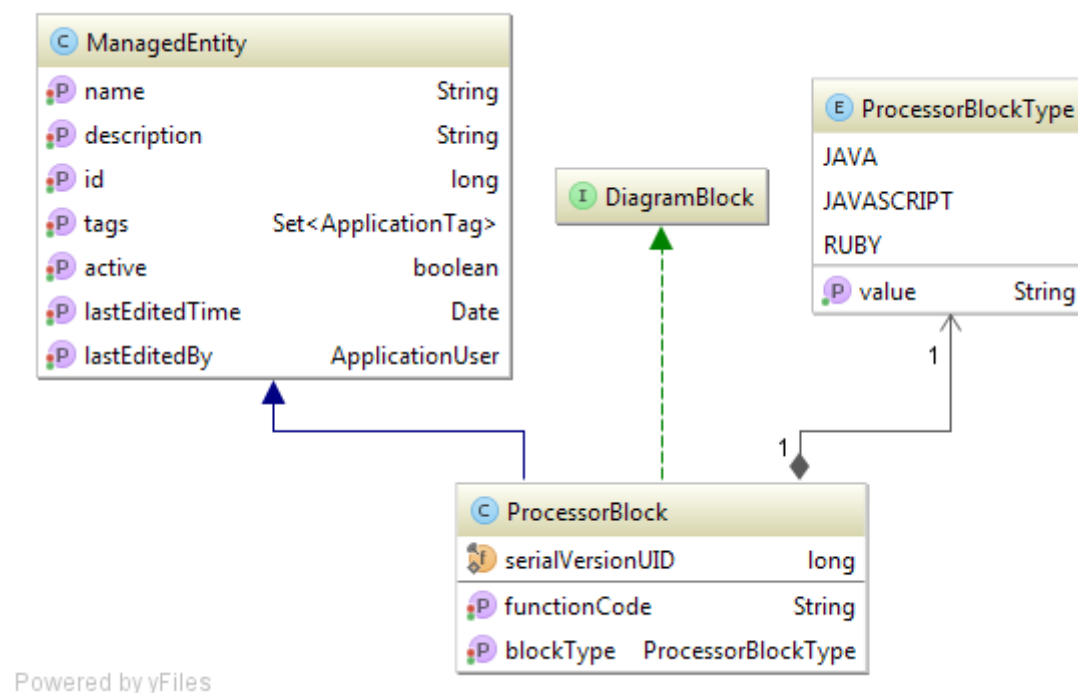


Figura 2.8: Clasa **ProcessorBlock**

2.2.5 Diagrama funcție bloc (FBD)

Diagramele funcție bloc reprezintă o implementare a unuia din cele patru limbaje de programare pentru automate programabile, specificate de standardul IEC 61131-3:2013 [20, pp. 128-140]. FBD reprezintă un limbaj de control al procesului, iar, în mod normal, toate blocurile de procesare dintr-o diagrama sunt executate. În cel mai simplu caz de utilizare, o FBD realizează următoarele operații:

- Acceptă date de intrare de la unul sau mai multe canale;
- Realizează o operație de transformare asupra acelor date folosind un bloc de procesare;
- Salvează rezultatul pe un canal de ieșire.

Legăturile dintre blocuri sunt unidirecționale. Un bloc de procesare poate trimite rezultatul său la unul sau mai multe blocuri, iar o intrare poate fi conectată la o singură ieșire. Transmiterea de date se face fără întârzieri. Diagramele sunt executate în ordinea definită de ordonarea topologică a nodurilor din graful ce definește diagrama [13, pp. 20-50].

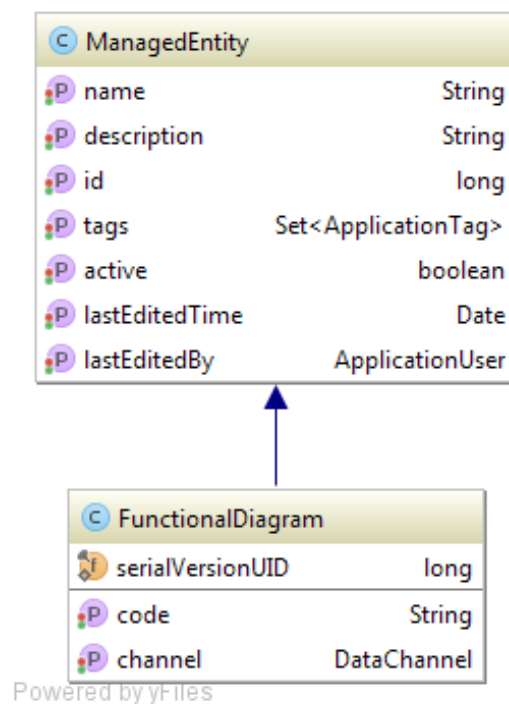


Figura 2.9: Clasa **FunctionalDiagram**

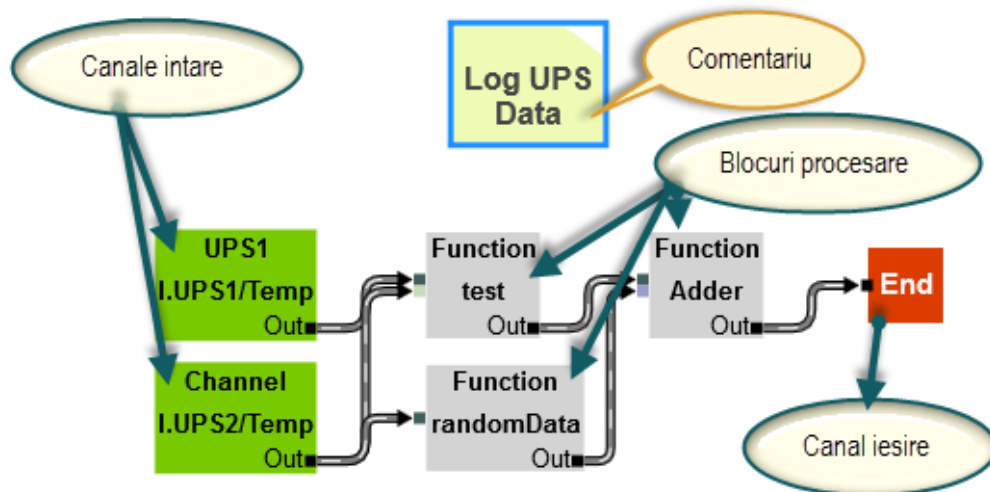


Figura 2.10: Exemplu diagrama funcțională

2.3 Primirea datelor

Primirea datelor se face prin intermediul unei interfețe de transfer a stării (REST). Pentru integrarea mai ușoară cu sisteme deja existente sunt acceptate mai multe formate. Ast-

fel, au fost implementate mai multe procesoare care primesc date atât într-un format nou, caracteristic aplicației, cât și în formate standard din industrie. În această versiune, modalități de trimitere a datelor au fost implementate:

- Trimitere către un singur canal, un singur punct odată, pe baza serviciului */api/put/inputId/channelId/data*. Acest serviciu adaugă un singur punct în baza de date, la momentul curent și este folosit pentru sisteme care trimit date rar și nu trebuie să se țină cont de data locală de pe device-ul care a trimis punctul de date.
- În formatul standard folosit de openTDSB, cu următoarele modificări care păstrează totuși compatibilitatea: metricile reprezintă numele canalului, iar tag-urile sunt opționale. Se acceptă atât formatul în care într-o cerere se află un singur punct, cât și formatul cu o listă de puncte. Canalele dintr-o cerere multidimensională nu trebuie să facă parte din același bloc de intrare. Acest mod de introducere a datelor este sugerat pentru sistemele care folosesc mai multe canale de date și care trimit seturi de date mai mari printr-o singură cerere. Spre exemplu, un dispozitiv poate trimite date de pe mai mulți senzori și poate stoca local mai multe măsurători pe același senzor pentru a trimite toate datele odată.

Odată primite, noile puncte de date trec prin procesul descris în figura 2.11:

1. Se interoghează baza de date pentru detalii privind canalul ce tocmai a primit date.
2. Dacă un preprocesor există pe canalul specificat, atunci el este încărcat.
3. Se execută preprocesorul cu punctul de date primit.
4. Se salvează rezultatul în baza de date.
5. Asincron, se lansează toate diagramele care trebuie să se execute atunci când se primesc date noi pe acest canal.
6. Asincron, se informează "ascultătorii" de primirea unor date noi de către canal.

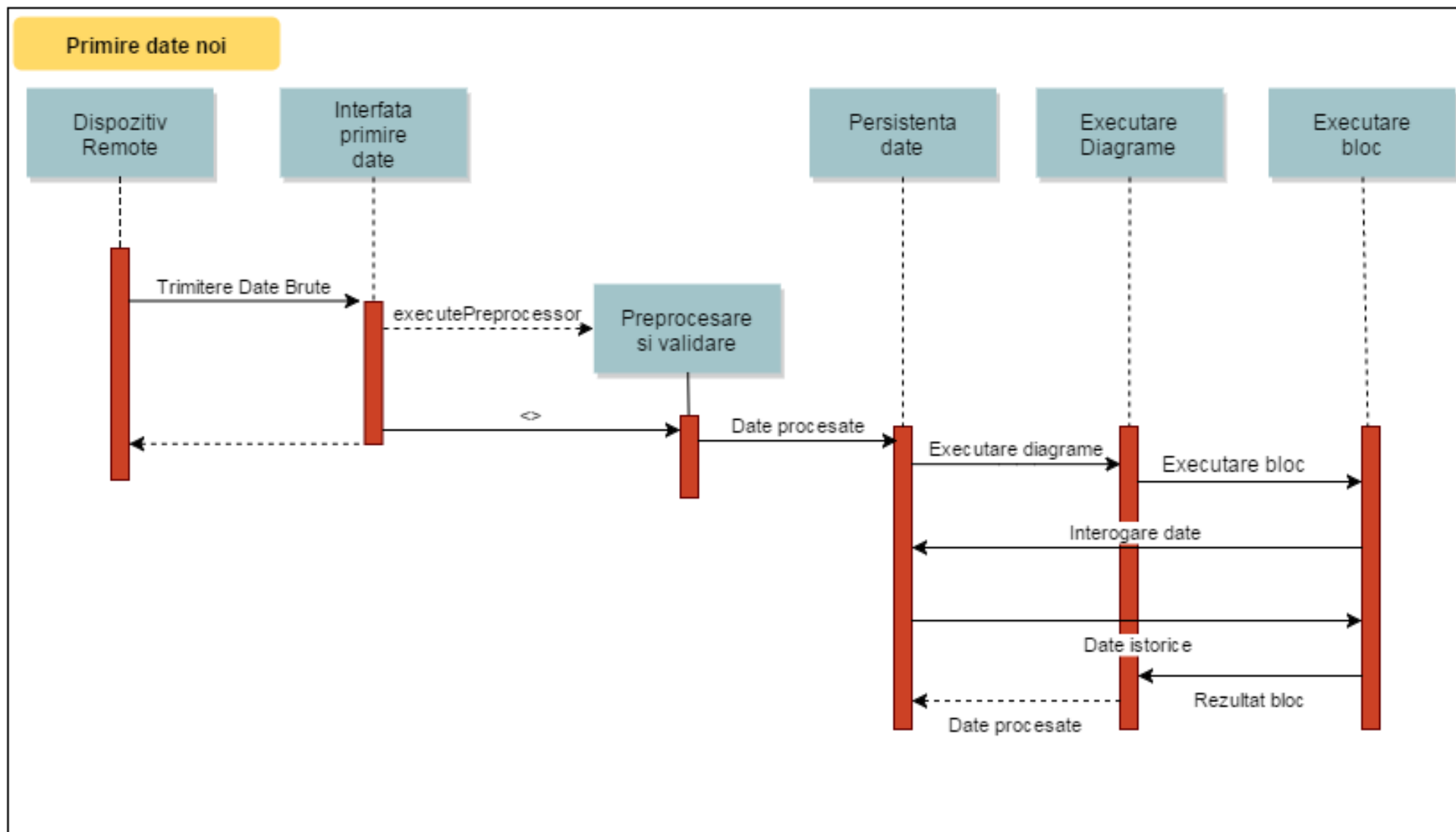


Figura 2.11: Diagrama de secvențe pentru introducerea de noi date și execuția diagramelor

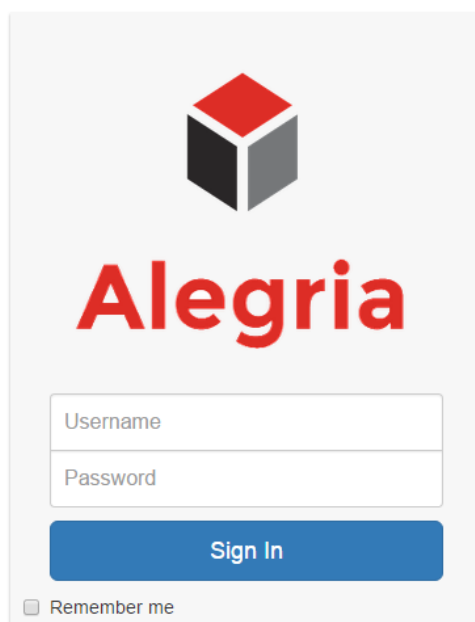
Capitolul 3

Interfața aplicației

Aplicația dispune de o interfață modernă și a fost implementată folosind standardul HTML5, cu pagini dinamice care păstrează starea în client, prin încărcarea asincronă a elementelor. Întreaga interfață este de tip responsive, scalându-se automat pentru afișare pe dispozitive cu rezoluții diferite (monitor, tabletă sau telefoane).

La accesarea aplicației, utilizatorul este întâmpinat de interfață de autentificare din figura 3.1.

Sign in to continue to Alegria



The login form is centered on a light gray background. At the top, it features the text 'Sign in to continue to Alegria' in a small, dark gray font. Below this is the Alegria logo, which consists of a 3D cube with a red top face and black side faces. Underneath the logo, the word 'Alegria' is written in a large, bold, red sans-serif font. The form contains two input fields: 'Username' and 'Password', both with light gray borders and placeholder text. Below these fields is a prominent blue button with the text 'Sign In' in white. At the bottom left of the form, there is a small checkbox followed by the text 'Remember me'.

Figura 3.1: Autentificarea în aplicație

Login-ul în aplicație permite accesul la interfața de management și monitorizare. Na-

vigația se face printr-un meniu aflat în antetul paginii, vizibil în figura 3.2

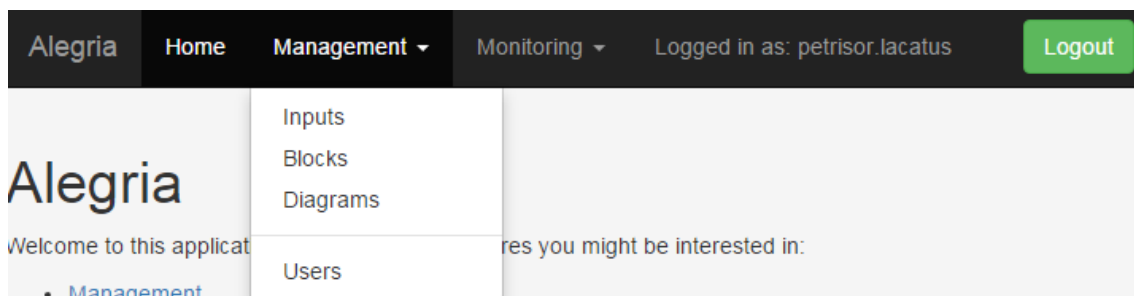


Figura 3.2: Navigarea prin funcțiile aplicației

Dacă utilizatorul este administrator, atunci el poate accesa și administrarea utilizatorilor. Aici el poate adăuga noi utilizatori, modifica utilizatorii existenți sau poate dezactiva accesul unora din aplicație.

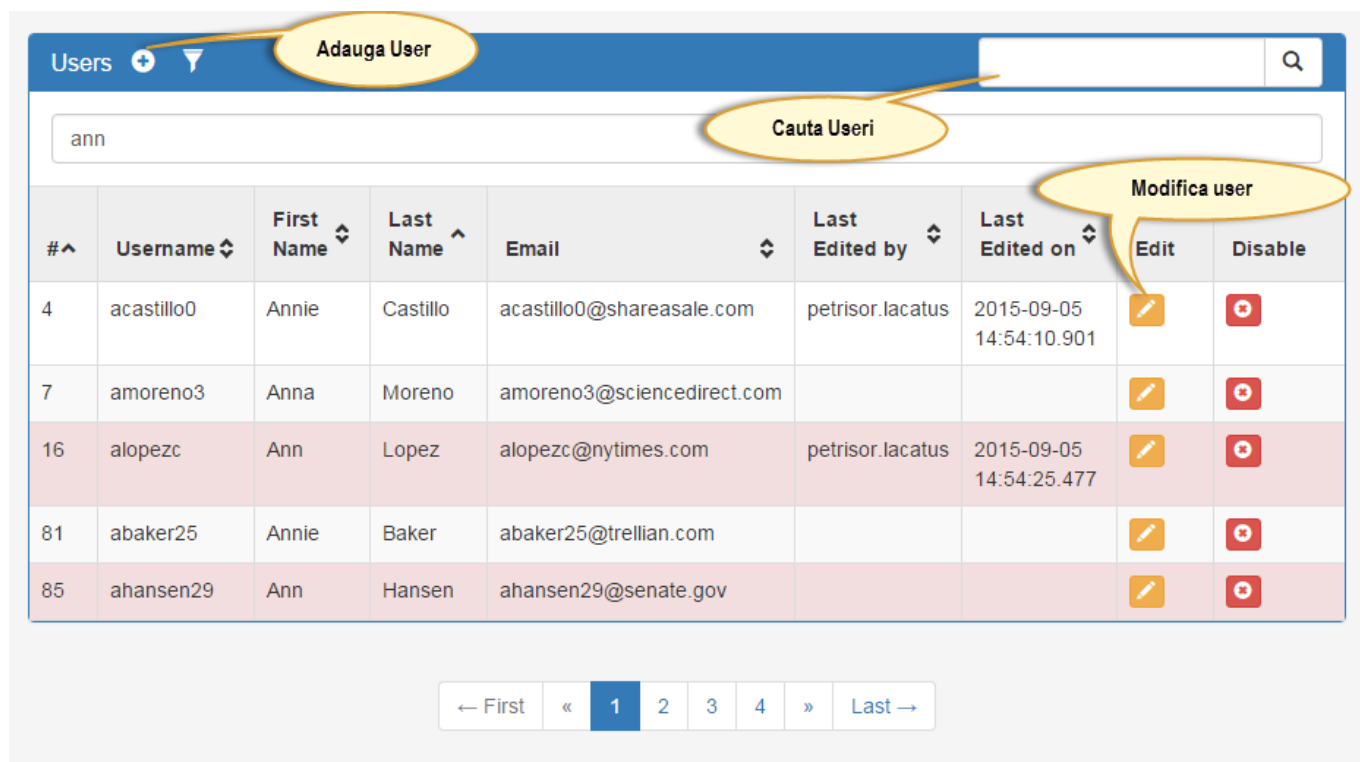


Figura 3.3: Managementul utilizatorilor din aplicație

Toate entitățile dispun de o interfață similară cu cea din figura 3.3, în care utilizatorul are acces facil la entitățile existente în sistem. Tabelele în care datele sunt afișate permit sortarea și filtrarea, ușurând astfel găsirea entității ce trebuie modificată. În plus, căutarea se poate face nu doar după numele entității, ci și după taguri sau descriere. Conținutul este paginat astfel încât timpul de încărcare a paginii să fie cât mai mic.

În cazul administrării blocurilor de intrare, un panou modal, încărcat dinamic, este disponibil pentru adăugare și editare. În acest panou, utilizatorul poate seta taguri și administra canalele de pe acel bloc.

Edit input

Input Name

UPS1

Tags

nxdata x

ups x

Description

UPS din nxdata, rack2

Active

☒

Input channels

+Add new Channel

Temp	Float	test	—
Input Voltage	Float	Input Preprocessor	—
Output Voltage	Float	Input Preprocessor	—
Capacity	Integer	Input Preprocessor	—

Cancel

Save

Figura 3.4: Editarea unui bloc de intrare

Pentru management-ul blocurile de procesare, utilizatorul are la dispoziție și un editor inteligent care afișează codul colorat în funcție de limbajul selectat. Astfel, utilizatorii pot depista erorile de sintaxa în timpul editării codului.

Edit block ✕

Name

Adder

Description

Adds all the input data

Tags

test x rxdata x

Block Type

Javascript ▼

Code

```
1 var parseInput = function(data) {
2   var sum = 0;
3   for each (var el in data) {
4     if(el.value > 0) {
5       sum = sum + el.value;
6     }
7   }
8   return sum;
9 };
```

Active

☒

Cancel

Save

Figura 3.5: Modificarea unui bloc de procesare și editarea codului

Editarea diagramelor funcționale se face vizual, într-o interfață asemănătoare cu cea din SimuLink sau LOGO! Soft Comfort. Aceasta interfață permite adăugarea vizuală a blocurilor dintr-o paletă, căutarea automată a instanțelor de blocuri și exportarea schemei diagramei pentru salvare într-un mediu extern.

Blocurile pot fi modificate atât prin schimbarea proprietăților acestora din meniul ce apare în partea de jos a diagramei când blocul este selectat, dar și direct prin acțiunea de click stânga sau dreapta pe acesta. Pentru blocurile de procesare, utilizatorul are opțiunea să vizualizeze și să modifice codul direct din aceasta pagină, fără a mai naviga spre pagina de administrare a blocurilor de procesare. Alături de câmpul de descriere, comentariile adăugate direct peste diagrama, ajută la documentarea implementării și funcționalității.

Edit diagram

Name

Description

Channel

Tags

Active ☒

Channel **Function** **Comment**

```

graph LR
    I1[I1 I.UPS1/Temp Out] --> P1[P1 test Out]
    I1 --> P2[P2 randomData Out]
    I2[I2 I.UPS2/Temp Out] --> P1
    I2 --> P2
    P1 --> P5[P5 Adder Out]
    P2 --> P5
    P5 --> End[End]
    
```

Execution Ordering

[Delete](#) [Render SVG](#) [Save](#)

Figura 3.6: Realizarea unei diagrame funcționale

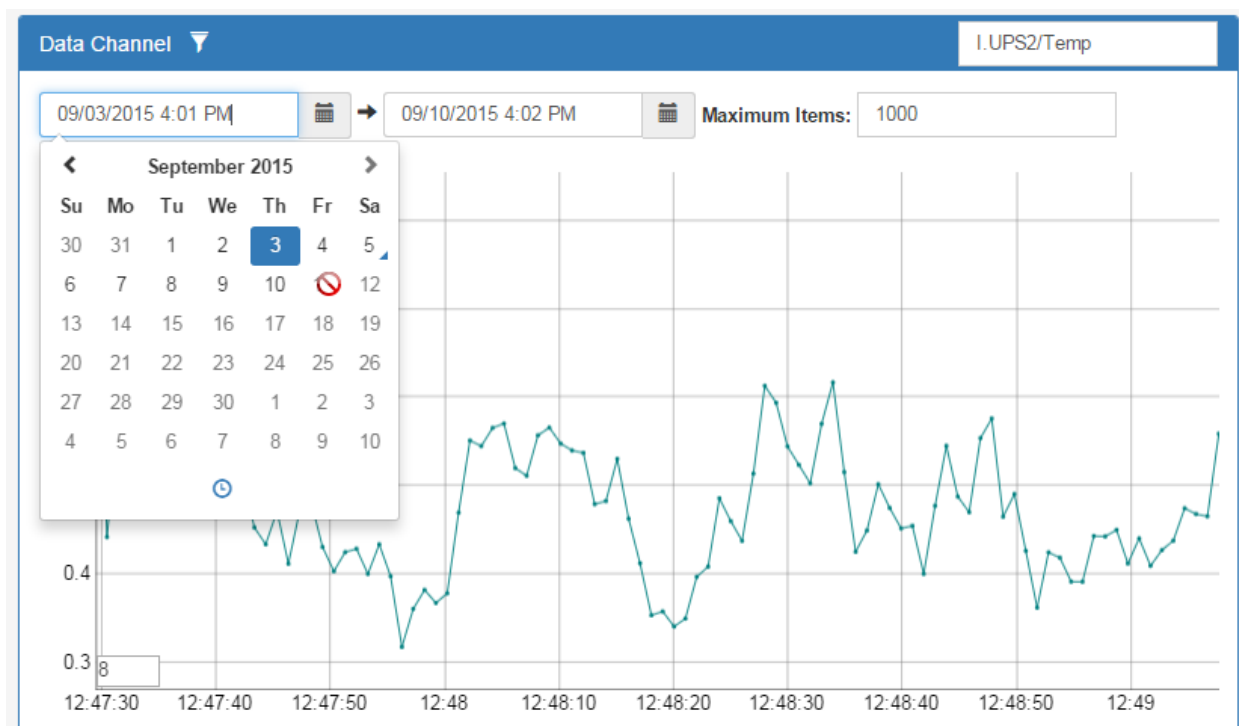


Figura 3.7: Monitorizarea datelor de pe un canal

Capitolul 4

Implementarea aplicației de achiziție, procesare și distribuție a datelor (Alegria)

4.1 Interfața Web

Aplicația de achiziție, procesare și distribuție a datelor (Alegria) a fost implementată cu ajutorul platformei **Spring Boot** [22]. Platforma a fost aleasă pentru stabilitatea ei excepțională, folosind Spring Framework care stă la baza unora din cele mai mari aplicații existente [23], dar și pentru ușurința prin care o aplicație poate fi compusă din elemente funcționale, abstractizând peste nivelele de jos a programului, permițând alocarea timpului pe logica aplicației și nu pe implementarea platformei pe care aplicația rulează. Un alt motiv pentru care a fost aleasă platforma Spring, este faptul că suportă programarea orientată pe aspecte și injecția dependențelor, permițând scrierea de cod curat și ușor de testat.

Cum aplicația este bazată pe arhitectura MVC (model-view-controller), aceasta a fost structurată în trei elemente separate:

- **Interfața vizuală:** Realizată în **HTML5**, folosind motorul de templating Thymeleaf pe server și Bootstrap cu JQuery în client pentru afișarea paginilor. Această combinație permite realizarea de pagini de tip responsiv, compatibile cu toate dispozitivele existente pe piață indiferent de rezoluție (monitor cu display mare, tabletă,

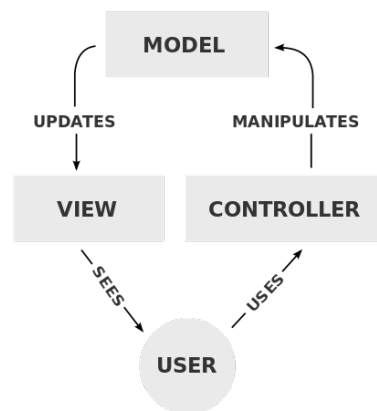


Figura 4.1: Colaborarea între componentele MVC

telefon mobil, etc).

- **Modele:** O reprezentare a entităților din baza de date în sistemul de achiziție, procesare și distribuție a datelor (Alegria).
- **Controller-e:** Realizează legătura dintre partea vizuală a aplicației și entitățile din baza de date, asigurând atât metodele care "completează" template-urile cu date, cât și implementarea interfeței API care introduce și extrage date.

4.1.1 Securitate

Securitatea este un aspect foarte important ce trebuie considerat, atât pentru accesul la date, cât și la entități. Astfel, în implementare s-a folosit framework-ul **Spring Security** care ușurează management-ul securității, fiind puternic integrat și cu restul platformei Spring.

Pentru autentificare în vederea utilizării unei resurse protejate, utilizatorul va folosi unul din două mecanisme:

- Autentificare securizată prin username și parolă, aceste detalii fiind stocate în tabelă *application_user*, parola fiind stocată abia după ce a fost trecută printr-o funcție criptografică de hashing. Această metodă este folosită pentru autentificarea utilizatorilor în interfața de management și monitorizare. Odată ce procesul a reușit, un token unic va fi generat, iar request-urile următoare vor fi verificate pe baza procesului descris mai jos.
- Autentificare pe baza de token, folosită pentru securizarea API-ului, dar și în cazul în care un utilizator s-a autentificat deja cu username și parolă. Fiecare request trebuie să aibă un token, fie într-un cookie, fie ca parametru în url.

Tot în scopuri de securitate, fiecare entitate ce poate fi modificată menține un istoric al tuturor modificărilor, împreună cu utilizatorul care le-a efectuat, iar, pentru o dezvoltare ulterioară, accesul unui utilizator poate fi limitat doar la obiectele care au aceleași tag-uri ca și utilizatorul.

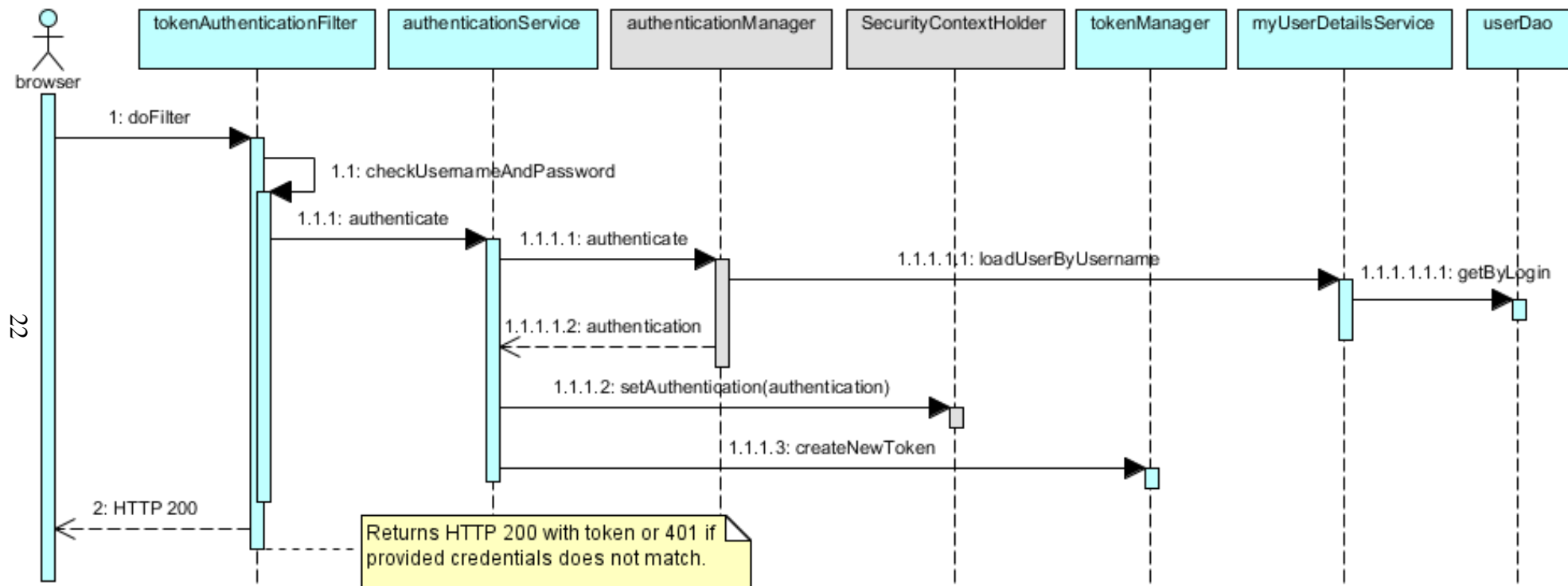


Figura 4.2: Procesul de autentificare în aplicație

4.1.2 Management

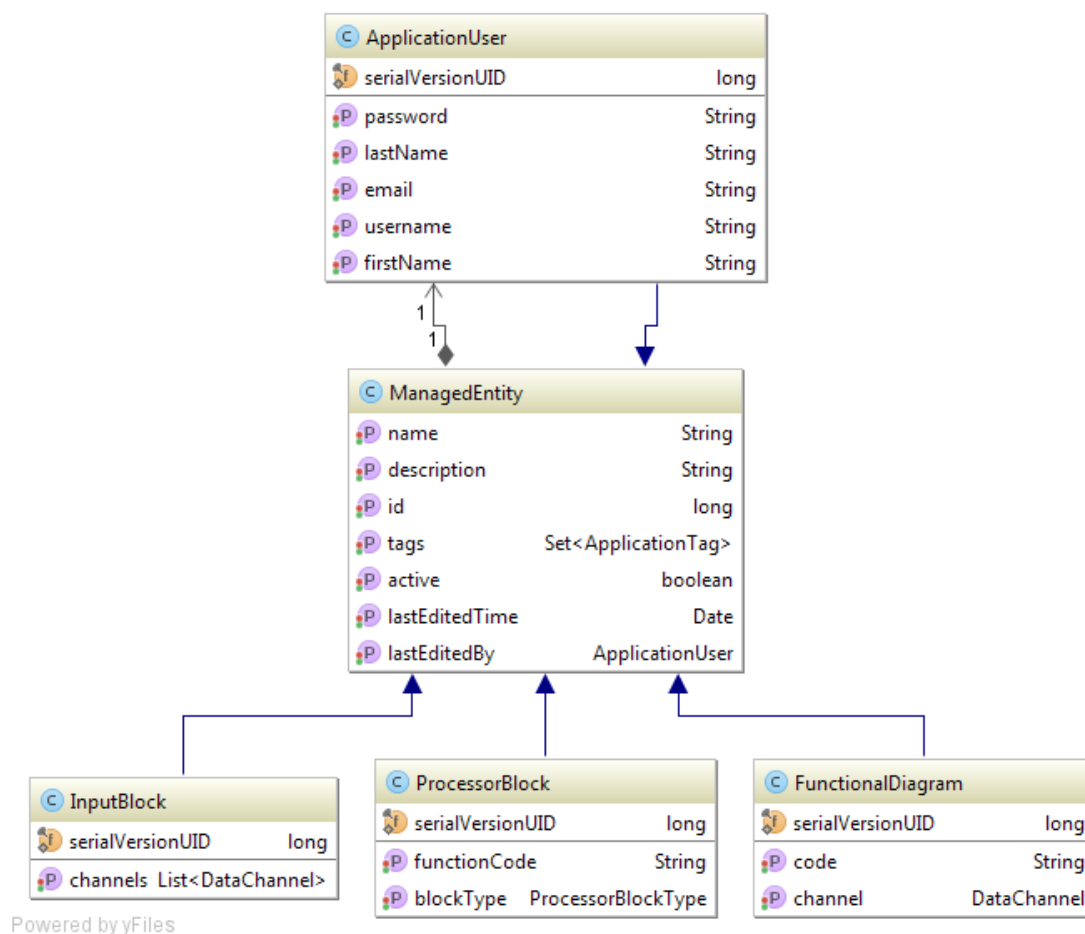


Figura 4.3: Diagrama claselor care sunt administrate de către utilizator și implementează ManagedEntity

Toate entitățile care implementează ManagedEntity permit apoi operații de adăugare, modificare și ștergere. Acest proces de administrare vizuală folosește următoarele resurse:

- Un **repository**, care extinde JpaRepository din framework-ul Spring Data. Acesta asigură operații de căutare, creare, citire, modificare și ștergere a entităților. Un avantaj al folosirii acestui repository, care implementează paradigma Data Access Object (DAO) este că interacțiunea cu baza de date se face într-un mod consistent și sigur, incompatibilitățile de tip fiind detectate la compilare, și nu la rulare. În Alegria, toate repository-urile folosite, împreună cu implementările lor, se află în package-ul `ro.pub.acse.sapd.repository`.
- O **vizualizare**, template Thyeleaf, care, într-o singură pagină HTML expune către utilizator toate operațiile suportate de repository. Această pagină este dinamică,

ce folosește dialoguri modale încărcate prin AJAX pentru a edita entități, fără a fi necesar ca utilizatorul să fie redirecționat. Entitățile sunt afișate sub forma tabelară, dinamică, care permite sortarea și filtrarea după diverse condiții. Dialogul modal de editare este specific entității modificate. Vizualizările pentru întreaga listă se află în `resources/templates/management` iar conținutul dialogului modal se află în subdirectorul `fragments`.

- Un **controller**, clasă cu adnotarea `@Controller`, leagă repository-ul de vizualizare și specifică către Spring care sunt endpoint-urile (căile pe care acest controller le tratează) prin adnotarea `@RequestMapping`. Controllere-le pentru managementul entităților se află în `ro.pub.acse.sapd.controller.web.management`. Un exemplu de metode care sunt tratate într-un asemenea controller se găsesc în figura 4.4.

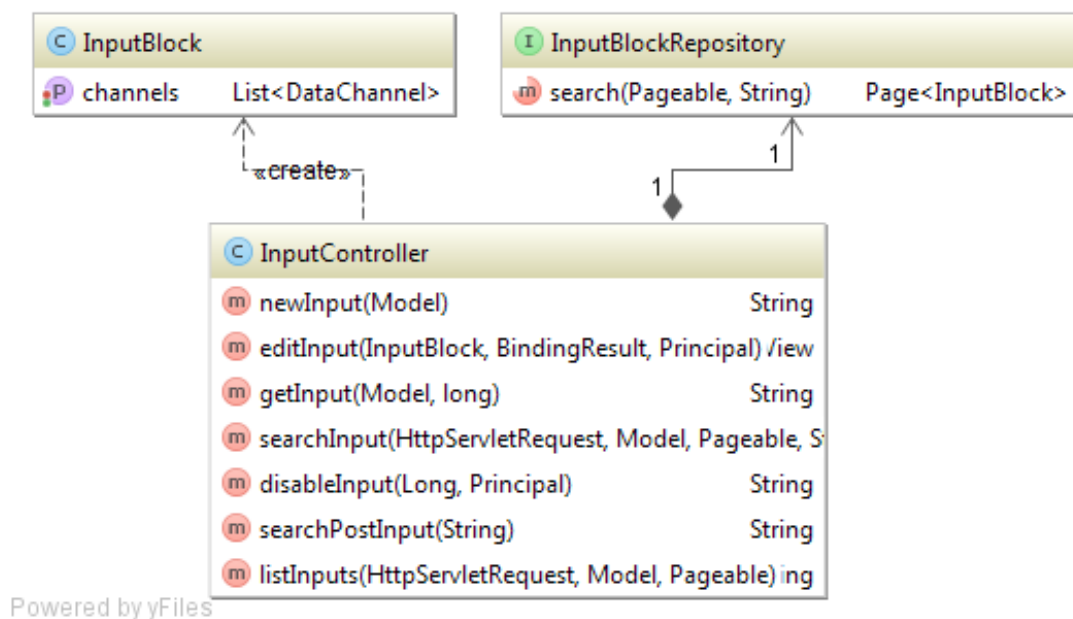


Figura 4.4: Interacțiunea dintre repository, controller și entitate

4.1.2.1 Managementul blocurilor de intrare

Pe lângă elementele descrise mai sus, la management-ul blocurilor de intrare trebuie ca utilizatorul să poată vizualiza și modifica lista de canale ale unui bloc. În vederea implementării acestei particularități, în dialogul modal pentru adăugare și modificare, a fost realizat un formular dinamic cu câte o linie pentru fiecare canal de date. Pentru blocurile de intrare care au deja un canal atașat, acest formular este generat de către server,

în template-ul `thymeleaf input.html`, iar, dinamismul formularului este implementat cu ajutorul unor funcții JavaScript care manipulează structura documentului.

+Add new Channel			
Temp	Float	transformToC	-
Voltage	Float	Input Preprocessor	-
Battery Capacity	Float	Input Preprocessor	-
Input Voltage	Float	Input Preprocessor	-
onAC	Integer	toBoolean	-

Figura 4.5: Formular HTML dinamic pentru editarea canalelor unui bloc de intrare

Pentru selectarea blocului de preprocesare a fost folosit endpoint-ul din API de la adresa `/blocks/getBlocks` care returnează lista tuturor blocurilor de procesare în format JSON. Această listă este folosită pentru permite completarea automata a câmpului pentru blocul de preprocesare a datelor ale unui canal, folosind librăria JavaScript **Bootstrap 3 Typeahead** [1].

4.1.2.2 Managementul blocurilor de procesare

O particularitate a editării blocurilor de procesare este folosirea librăriei JavaScript **CodeMirror** [3] pentru afișarea codului. Cuvintele cheie ale limbajului dat de tipul blocului fiind evidențiate, acest lucru făcând dezvoltarea codului mult mai ușoară. Un alt avantaj al acestei librări este posibilitatea găsirii erorilor de sintaxa mult mai rapid, fără a testa blocul. Aceasta funcționalitate este implementata cu ajutorul unei funcții care se execută de fiecare dată când se modifică valoarea selectată în input-ul "Block Type".

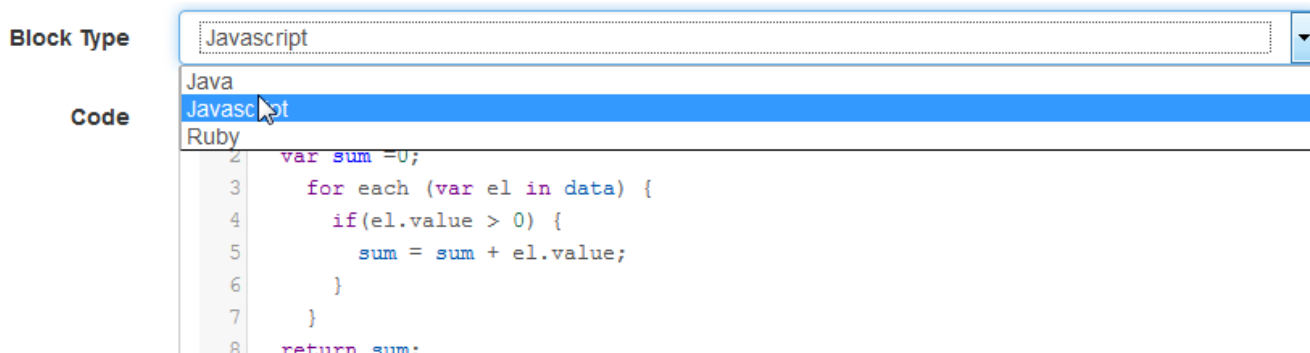


Figura 4.6: Modificarea limbajului din editorul CodeMirror în funcție de tipul blocului

4.1.2.3 Managementul diagramelor

Pentru implementarea funcțiilor de design vizual al diagramelor funcționale s-a ales librăria JavaScript **GoJS** [9]. Aceasta permite implementarea de diagrame interactive, fiind compatibilă cu toate browsere-le și cu dispozitivele mobile moderne. Librăria reprezintă un punct foarte bun de plecare deoarece asigură suport atât pentru funcționalități precum drag-and-drop, copiere și lipire, undo și redo, și multe alte funcționalități. Un alt avantaj al acestei librării este ca suportă adăugarea de condiții asupra diagramei chiar la construcția acesteia. Din acest motiv, librăria a fost configurată să nu permită decât legături de la o ieșire la o intrare.

În vederea realizării acestor configurări a fost scris fișierul JavaScript `diagram.js`. Aici sunt configurate tipurile de blocuri:

- **Canale de intrare:** acestea sunt configurate să nu poată avea intrări, ci doar o singură ieșire. Atunci când un canal de intrare este selectat, utilizatorul poate să îi atribuie un nume și să selecteze care este canalul ce indică de acel bloc. Aceasta selecție se face cu ajutorul unui input cu auto completare.
- **Blocuri de procesare:** deoarece numărul de intrări este variabil a fost implementată funcția `addPort`. Aceasta este apelată atunci când utilizatorul dă click pe opțiunea "Add input" din meniul contextual al blocului. Pentru operațiunea de ștergere a portului a fost implementată funcția `removePort(port)`. Ordinea acestor porturi determină și ordinea elementelor din lista cu care este apelat blocul de procesare. Când un bloc este selectat utilizatorul poate să îi atribuie un nume și să indice blocul de procesare la care se referă. Această selecție se face cu ajutorul unui input cu auto completare. Astfel, același bloc de procesare poate fi folosit de mai multe ori chiar și în cadrul aceleiași diagrame. Pentru ușurință dezvoltării, linkuri către detaliile blocului de procesare sunt disponibile chiar în interiorul diagramei;

- **Stop:** bloc ce semnifică canalul de ieșire. Configurat cu un singur port de intrare, el nu poate fi legat decât la o singură ieșire;
- **Comentariu:** nu se poate lega în diagramă și nu ia parte la execuția acesteia.

Toate aceste blocuri sunt adăugate și într-o paletă pentru adăugare ușoară în diagramă. Deoarece blocul "Stop" nu poate avea decât o singură instanță, acesta nu este disponibil în paletă.

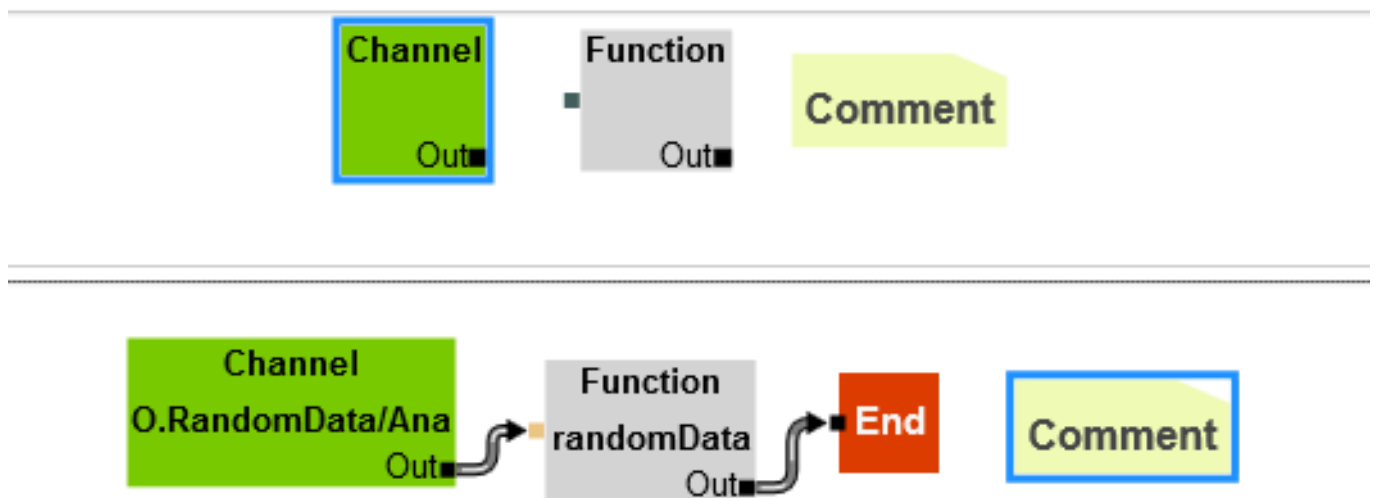


Figura 4.7: Paleta de blocuri și exemplu de instanțe ale acestora

Când utilizatorul salvează o diagramă, este generat un model JSON al acesteia, model ce este apoi salvat în baza de date. În acest model sunt salvate proprietățile diagramei, urmate de lista nodurilor și de legăturile dintre porturile nodurilor.

4.1.2.4 Administrare tag-uri

După cum s-a discutat în secțiunea despre securitate, fiecare entitate care este administrată de utilizator poate avea mai multe taguri. Acestea reprezintă o lista de cuvinte care specifică un concept, spre exemplu, toate entitățile care sunt folosite într-o diagrama pot avea același tag. Tagurile permit astfel gruparea entităților, fiind folosite în căutare.

Input Name: UPS1

Tags: ups x nx

Description: UPS din nxdata, funcie

nxdata

Figura 4.8: Editarea tag-urilor unui bloc de intrare

Pentru implementare, pe partea de server au fost creata clasa `ApplicationTag`, cu doua câmpuri: `nume`, și `Id`. Toate instancele acestei clase sunt salvate în baza de date, în tabela `application_tag`, iar endpoint-ul `/tags/getTags` întoarce toate tag-urile din acea tabelă. Acest serviciu API este folosit de librăria `JavaScript Bootstrap Tags Input` [2], care, împreună cu librăria de autocompletare [1], permite utilizatorului să selecteze taguri deja existente. Atunci când utilizatorul introduce totuși un tag care nu există deja în baza de date, acesta este creat automat, fiind setat direct pe entitatea modificată. Fiecare `ManagedEntity` are un set de taguri, permițând salvarea tagurilor în baza de date, într-un tabel separat, care implementeaza relația de tip `Many-to-Many` dintre entitate și `ApplicationTag`.

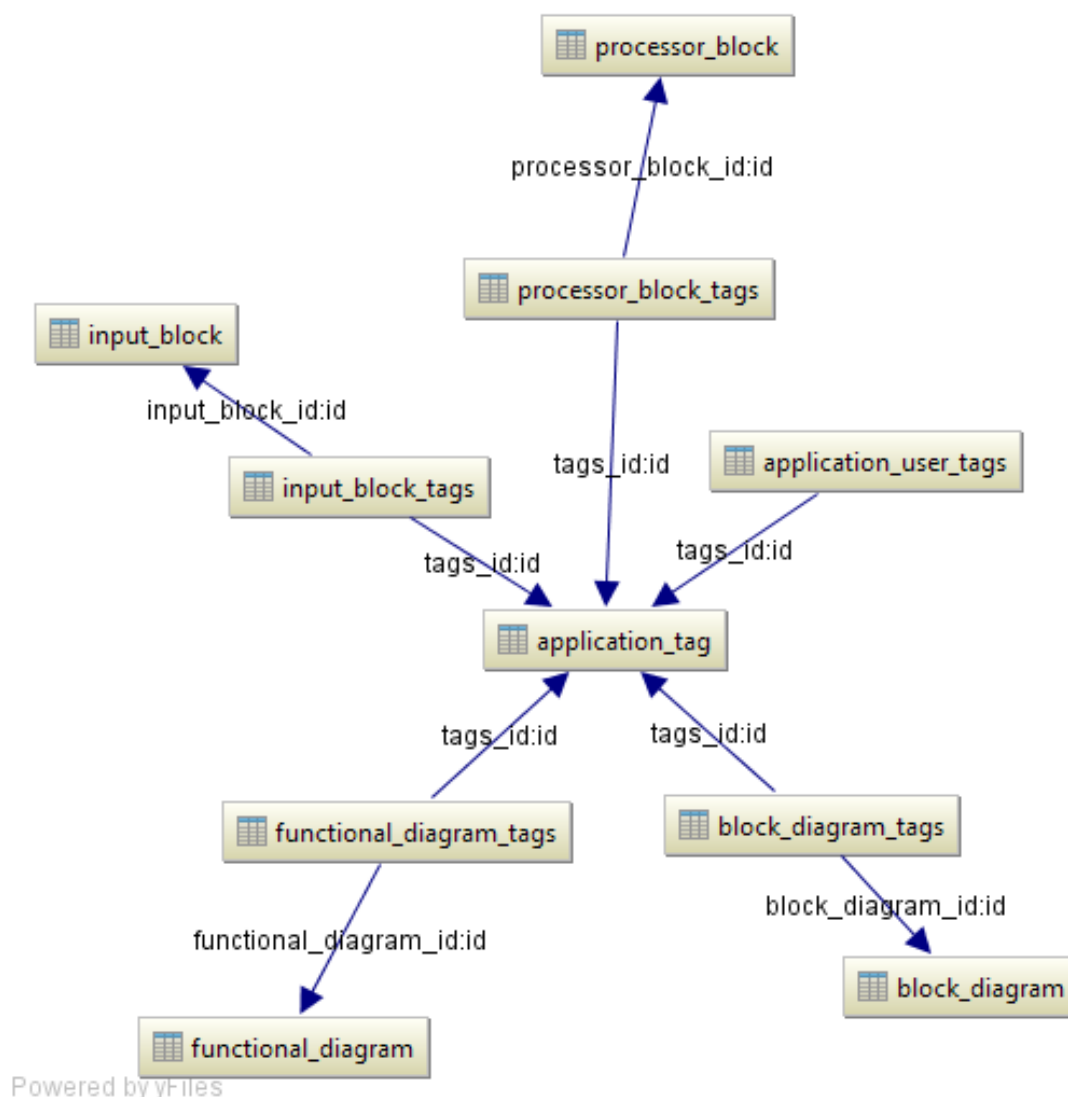


Figura 4.9: Relațiile dintre tabelă tags și celelalte entități

4.1.3 Monitorizare

O chestiune de mare importanță este vizualizarea datelor ce au intrat sau au fost procesate de aplicație. În acest scop, a fost implementată vizualizarea datelor în timp real. Pentru extragerea informațiilor de pe un canal se folosesc endpoint-urile din API de la `/api/fetch/channelId`, care întorc punctele de pe un canal dintr-un anumit interval de timp, în format JSON.

Deoarece datele procesate de aplicație sunt de mai multe tipuri, apare totuși problema modului de afișare. În funcție de datele întoarse prin API se selectează unul din două moduri. Pentru datele numerice se folosește o reprezentare grafică, folosind librăria `dygraphs` [5], iar pentru datele care nu pot fi reprezentate numeric, se folosește sunt afișarea

sub formă de tabel, ca șiruri de caractere.

Pentru ambele moduri de reprezentare, utilizatorul are opțiunea să obțină doar datele dintr-o anumită perioadă de timp sau ultimele înregistrări pe acel canal. Acest mod de selecție este ilustrat în figura 4.10

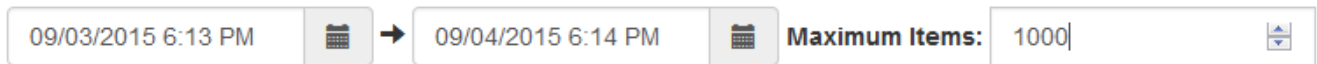


Figura 4.10: Filtrarea datelor obținute

4.2 API-ul aplicației

Pentru a permite interfațarea aplicației cu alte servicii, dar și pentru a facilita dezvoltarea unor interfețe web dinamice, aplicația dispune de un API REST. Acesta a fost configurat să folosească date în format JSON.

În următoarea lista sunt descrise toate endpoint-urile API-ului, împreună cu detalii despre folosirea acestora. În afară de câteva excepții menționate, toate aceste servicii folosesc metoda HTTP GET.

- `/blocks/getBlocks`: Întoarce toate blocurile de procesare din baza de date;
- `/blocks/getChannels`: Întoarce toate canalele declarate în baza de date. Pentru fiecare canal returnat se specifică blocul de intrare de care aparține sau dacă este ieșirea unui diagrame funcționale;
- `/tags/getTags`: Întoarce toate tagurile existente în aplicație;
- `/api/put/{inputId}/{channelId}/{data}`: adaugă punctul data pe canalul specificat prin `channelId`. Datele primite sunt în format `String` și respectă standardul specificat în RFC3986 [30]. Folosește metoda PUT;
- `/api/put/openTDSB`: Adaugă date în formatul openTDSB [28]. În corpul requestului trebuie să se afle un JSON care respectă standardul impus. Folosește metoda PUT.
- `/api/fetch/{channelId}`: Întoarce date de pe canalul `channelId`. Următorii parametrii pot fi folosiți pentru a extrage doar anumite date:
 - `from`: Data de început a istoricului, în format ISO 8601.
 - `to`: Data de final a istoricului, în format ISO 8601.

- `maxItems`: Numărul maxim de înregistrări ce sunt returnate. Acest parametru are 500 ca valoare implicită.
- `/api/fetch/last/{channelId}`: Întoarce date de pe canalul `channelId`. Următorii parametrii pot fi folosiți pentru a extrage doar anumite date:
 - `maxItems`: Numărul maxim de înregistrări ce sunt returnate. Acest parametru are 500 ca valoare implicită.

Pe lângă aceste metode, aplicația mai pune la dispoziție și API-ul generat automat cu ajutorul Spring Data. Acesta permite adăugarea, modificarea, ștergerea și căutarea entităților folosite în aplicație programatic.

4.3 Executarea unui bloc de procesare

Interacțiunea cu blocurile de procesare se face prin Spring Bean-ul `BlockExecutor`. Acesta este responsabil de inițializarea implementărilor interfeței `GenericBlockExecutor`. Pentru fiecare implementare există un câmp asociat în enumerația `ProcessorBlockType`, câmp care este folosit ca proprietate în entitatea `ProcessorBlock`. Acest Bean poate fi apoi injectat folosind adnotarea `@Autowired` în toate clasele care doresc să execute blocuri.

Interfața `GenericBlockExecutor` are o singură metodă `processData` cu doi parametri: unul de tip `String`, reprezentând codul funcției ce trebuie executat și unul de tip `List<DataPoint>` care conține toate punctele ce pot fi folosite în blocul respectiv. Clasa `DataPoint` are două câmpuri: valoare și instanță de timp. În sistem există următoarele implementări a interfeței `GenericBlockExecutor`:

- `JavaBlockExecutor`: Acest executor reprezintă un caz special deoarece folosește clase locale ce se află deja în classpath-ul JVM-ului pe care rulează aplicația. Aceste blocuri pot fi folosite pentru a extinde aplicația cu cod de înaltă performanță, acționând ca un mecanism de extensii ale aplicației, permițând interacțiunea cu alte sisteme. Aceste blocuri pot să reprezinte doar o interfață pentru apelul unor librării externe făcând posibilă, spre exemplu, implementarea unui bloc care să execute scripturi MATLAB. Acest bloc primește ca parametru doar numele canonic al clasei, iar la execuție încarcă clasa la care face referință folosind funcții din pachetul `java.lang.reflect`. Dacă clasa nu este găsită sau are loc o eroare la execuția clasei, este generată o excepție de tipul `BlockExecutionException`.

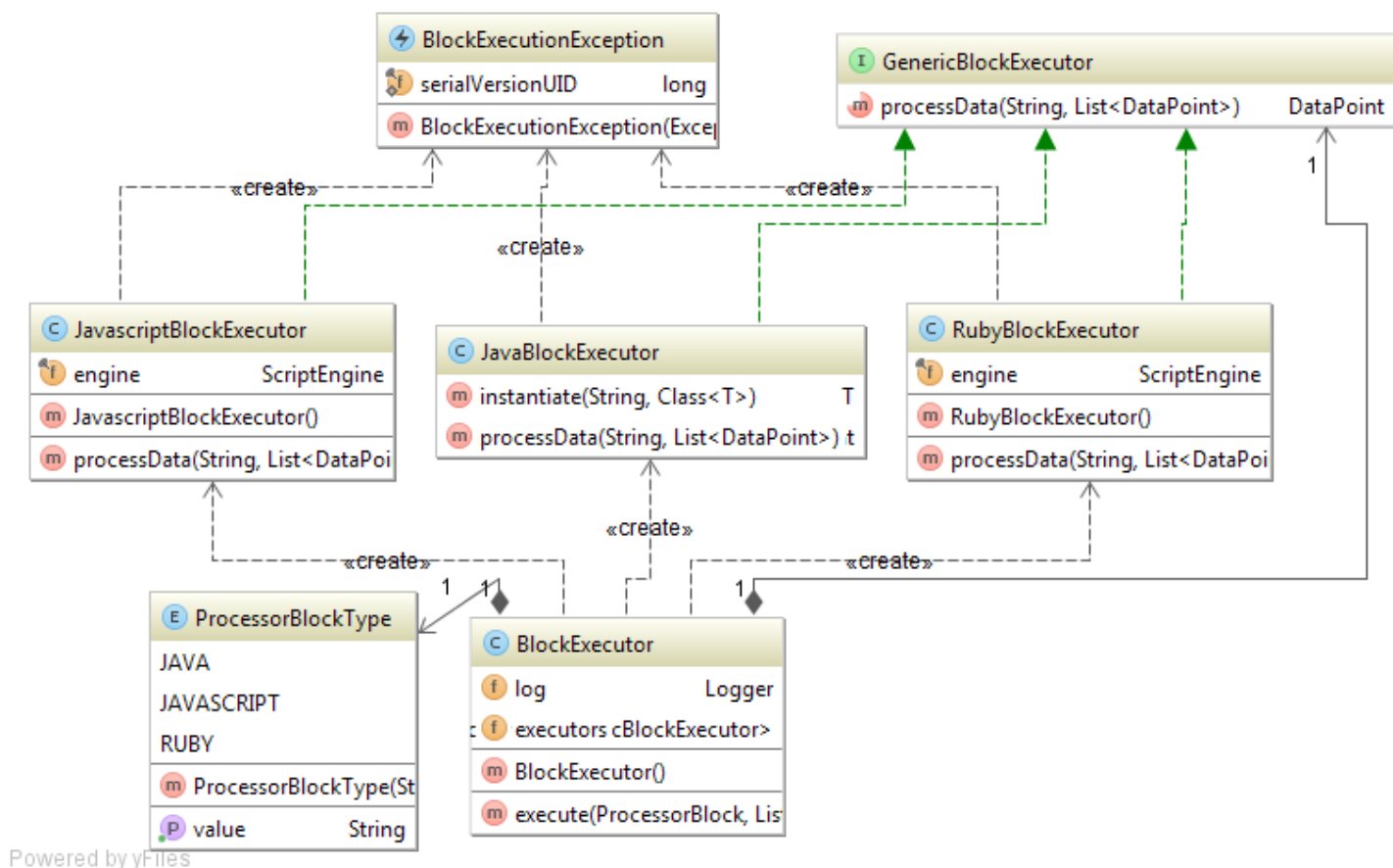


Figura 4.11: Diagrama claselor ce asigura executarea unei diagrame

- **JavascriptBlockExecutor**: Executor ce poate rula cod JavaScript pe server. Aceste blocuri trebuie să conțină o funcție `parseInput` care ia ca parametru o listă de `DataPoint`. Pentru execuția blocurilor de acest tip, standardul JSR 223 [21] vine în ajutor prin abstractizarea funcționalității interne necesare execuției de cod scris într-un limbaj dinamic, direct în mașina virtuală Java. În versiunea 8 a JVM-ului aceste funcții sunt executate folosind runtime-ul de înaltă performanță Nashorn care este accelerat de modificări recente, introduse în JSR 292 [25], ridicând performanța codului JavaScript la un nivel apropiat de funcțiile scrise în Java. Pentru versiunile precedente de JVM este folosit Mozilla Rhino. Dacă funcția `parseInput` nu este găsită sau la execuția script-ului are loc o eroare, este generată o excepție de tipul `BlockExecutionException`. Scriptul poate returna direct instanțe ale interfeței `DataPoint`, sau alte obiecte care sunt apoi reprezentate ca un `StringDataPoint`.
- **RubyBlockExecutor**: Executor ce rulează cod Ruby pe server. Din punct de vedere al implementării este similar cu `JavascriptBlockExecutor`, însă folosește librăria JRuby [27].

4.4 Executarea unei diagrame funcționale

Figura 2.11 prezintă și modul în care o diagrama este executată. Diagramele sunt lansate în execuție de fiecare dată când un canal folosit în ea primește informații noi. Dacă două canale primesc date în același timp atunci diagrama va fi lansată în execuție de două ori, câte o dată pentru fiecare punct de date primite.

4.4.1 Ordinea execuției blocurilor

Problema ordinii execuției unei FBD este intens dezbătută atât în literatură cât și în aplicațiile industriale. Cum standardul *IEC61131-3* [20] nu propune o soluție pentru ordinea de execuție, producătorii industriali folosesc metode proprii, de la separarea blocurilor într-un tabel și executarea de la stânga la dreapta, sus în jos [29, p. 11], la definirea manuală a ordinii execuției [15, p. 11] sau folosind algoritmi care determina automat ordinea de execuție [8, p. 5].

În implementarea din aplicația Alegria s-a ales proiectarea unei metode automate pentru depistarea ordinii în care diagrama trebuie executată. Deoarece aceasta reprezintă un graf aciclic orientat, prima etapă a execuției este transformarea într-un graf reprezentat prin lista de adiacență, unde fiecare bloc reprezintă un nod. Din această reprezentare se omit blocurile de comentarii. Odată ce transformarea a fost efectuată cu succes se încercă aplicarea unui algoritm de sortare topologică [4] a grafului obținut. Aceasta implică găsirea unei ordini astfel încât pentru orice arc orientat uv de la u la v , u este înaintea lui v . Matematic problema poate fi formulată astfel:

Definiție 1. *O ordine topologică, notată ord_D , au unui graf orientat aciclic $D = (V, E)$ atribuie fiecărui nod o valoare astfel încât $ord_D(x) < ord_D(y)$ pentru orice arc $x \rightarrow y \in E$.*

În literatura există mai mulți algoritmi pentru efectuarea unei asemenea sortări [17], însă, deoarece grafurile în discuție sunt statice, adăugându-se sau ștergându-se noduri, s-a ales un algoritm clasic, stabil din punct de vedere numeric descris de Kahn în 1962 [14].

Simplificat, algoritmul implementat urmărește următorii pași:

1. Identifică toate nodurile spre care nu vine nici un arc. Valoarea acestor noduri va fi 0. În cazul diagramelor, este vorba de toate canalele de intrare, și de blocurile de procesare care nu au nici o intrare. Dacă aceste noduri nu există, înseamnă că graful nu respecta condiția de graf aciclic, deci acesta nu va putea fi executat;
2. Se alege unul din nodurile găsite mai sus;
3. Se șterge acest nod de valoare zero, împreună cu toate arcele care ies din el;

4. Se repetă pașii 1 și 2 până când nu mai există noduri în graf.

Algoritmul descris mai sus rulează în $\mathcal{O}(V + E)$.

Algoritmul 1: Algoritmul lui Khan pentru sortare topologică

Data: Un graf orientat aciclic reprezentat prin lista de adiacență

Result: Lista nodurilor ordonate topologic

```

1  $L \leftarrow$  Lista goală ce va conține nodurile sortate ;
2  $S \leftarrow$  Lista tuturor nodurilor spre care nu există nici un arc ;
3 while  $S$  conține elemente do
4   șterge nodul  $n$  din  $S$  ;
5   introdu nodul  $n$  în  $L$  ;
6   foreach nod  $m$  care are un arc  $e$  de la  $n$  la  $m$  do
7     șterge arcul  $e$  din graf;
8     if  $m$  nu mai are arce spre el then
9       inserează  $m$  în  $S$ 
10 if mai există arce în graf then
11   return Eroare: Graful are cel puțin un ciclu
12 else
13   return  $L$  (graful sortat topologic)

```

Astfel, pentru diagrama din figura 2.10, o posibilă ordine de execuție este descrisă în figura 4.13.

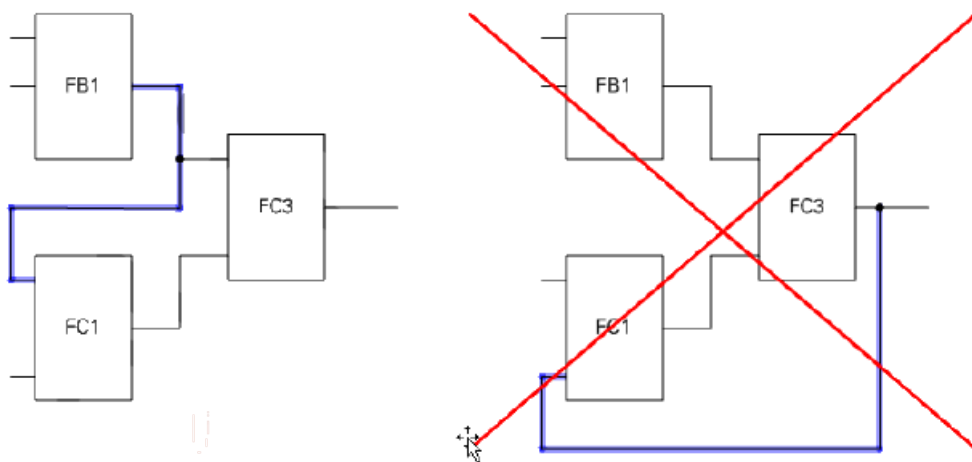


Figura 4.12: Diagramă ce conține cicluri

După cum se observă în alg. 1, acesta poate detecta grafuri care conțin cicluri și nu pot fi rezolvate. Această verificare permite detectarea erorilor, precum cea din diagrama 4.12 și informarea utilizatorului pentru ca acesta să rezolve problema.

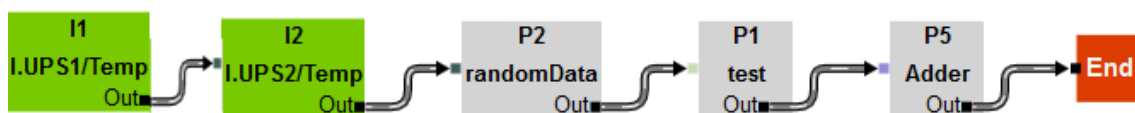


Figura 4.13: Ordinea execuției pentru diagrama din figura 2.10

Deși ciclurile la nivel de diagrama nu sunt permise, cele la nivel de canal sunt. Astfel, ieșirea unei diagrame poate fi setată ca intrare pentru aceeași diagramă, permițând executarea în buclă, deoarece odată ce o diagramă se termină de executat și salvează rezultatul pe canal, aceasta se relansează în execuție cu noi informații. Cu astfel de bucle se pot implementa diagrame ce interacționează între ele. Un alt aspect pozitiv al faptului ca ieșirile unor diagrame pot reprezenta intrări pentru alte diagrame este faptul că procese foarte complexe pot fi descompuse în părțile componente prin simpla înlănțuire a diagramelor.

4.4.2 Generarea rezultatului

Odată ce ordinea de execuție a fost calculată, procesul de calcul al rezultatului este destul de simplu, fiind ilustrat în alg. 2:

Algoritmul 2: Execuția unei diagrame FBD

Data: Lista nodurilor ordonate topologic

Result: Punctul de date ce trebuie adăugat pe canalul de ieșire a diagramei

```

1  rezultate ← Relație cheie-valoare între nod și rezultatul execuției lui;
2  L ← Lista ce conține nodurile sortate ;
3  foreach nod m din L do
4      intrari ← Lista goală de intrări pentru nodul m ;
5      foreach arc de la n către m do
6          if În rezultate exista rezultatul pentru blocul n then
7              Adaugă în intrari valoarea de la rezultate(n);
8          else
9              return Eroare: Graful nu poate fi executat;
10     if m este un bloc de procesare then
11         Executa blocul folosind intrările intrari;
12         Adaugă în rezultate valoarea calculată;
13     else
14         else if m este un canal de date then
15             Adaugă în rezultate ultima valoare de pe canal;
16 return Ultima valoare din rezultate
  
```

4.4.3 Implementarea execuției

Pentru execuția diagramelor este folosit Spring Bean-ul `DiagramExecutor`. Acesta este responsabil atât de parsarea unei diagrame cât și de execuția și extragerea rezultatelor. Acest Bean poate fi apoi injectat folosind adnotarea `@Autowired` în toate clasele care doresc să execute diagrame.

Parsarea este realizată cu ajutorul interfeței `DiagramParser`, aceasta permițând transformarea diagramei într-un graf reprezentat prin listă de adiacență. O implementare a acestei interfețe este clasa `GoJsDiagramParser` care parsează JSON-ul generat de librăria `GoJS`. Aceasta parsare se face cu ajutorul librăriei `Jackson`, folosind modelul de date din figura 4.15. Dacă parsarea nu este realizată cu succes, atunci o excepție de tipul `DiagramParseException` este aruncată. Dacă se dorește înlocuirea librăriei de front-end pentru realizarea diagramelor, sistemul poate fi adaptat doar prin scrierea unei noi clase ce implementează `DiagramParser`.

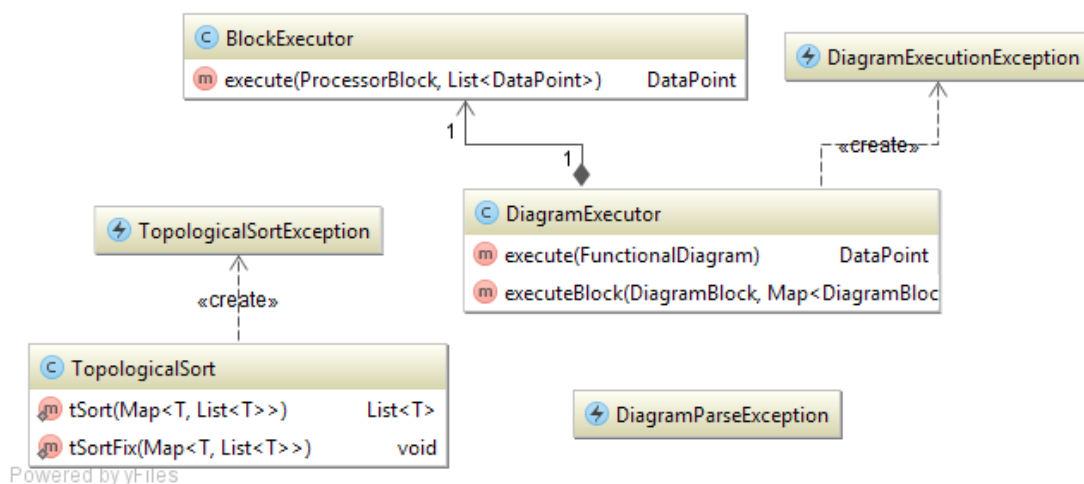


Figura 4.14: Diagrama claselor implicate în executarea unei diagrame

Odată ce parsarea a fost efectuată, execuția blocului poate continua: folosind clasa `TopologicalSort`, o implementare a alg. 1 cu tipuri generice și expresii lambda, ordinea de execuție este calculată. Apoi, alg. 2 este executat, și, cu ajutorul bean-ului `BlockExecutor` blocurile din diagrama sunt executate. Rezultatul final al metodei `execute` este un `DataPoint` care va fi salvat în baza de date, pe canalul de ieșire a diagramei.

O altă chestiune importantă este determinarea momentelor când o diagrama trebuie executată. În acest scop, pe fiecare canal se stochează o mulțime de diagrame care trebuie lansate atunci când se primesc informații noi. Această listă este actualizată de fiecare dată când diagrama se modifică prin interfața web. Metodele din `DiagramParser` sunt folosite pentru a obține lista de canale utilizate și pe fiecare dintre acestea, pe proprietatea

`Set<FunctionalDiagram>` `subscribedDiagrams` este adăugată acea diagrama. Pentru a nu se pierde consistența datelor, cât timp diagrama este modificată, ea nu se mai poate lansa în execuție.

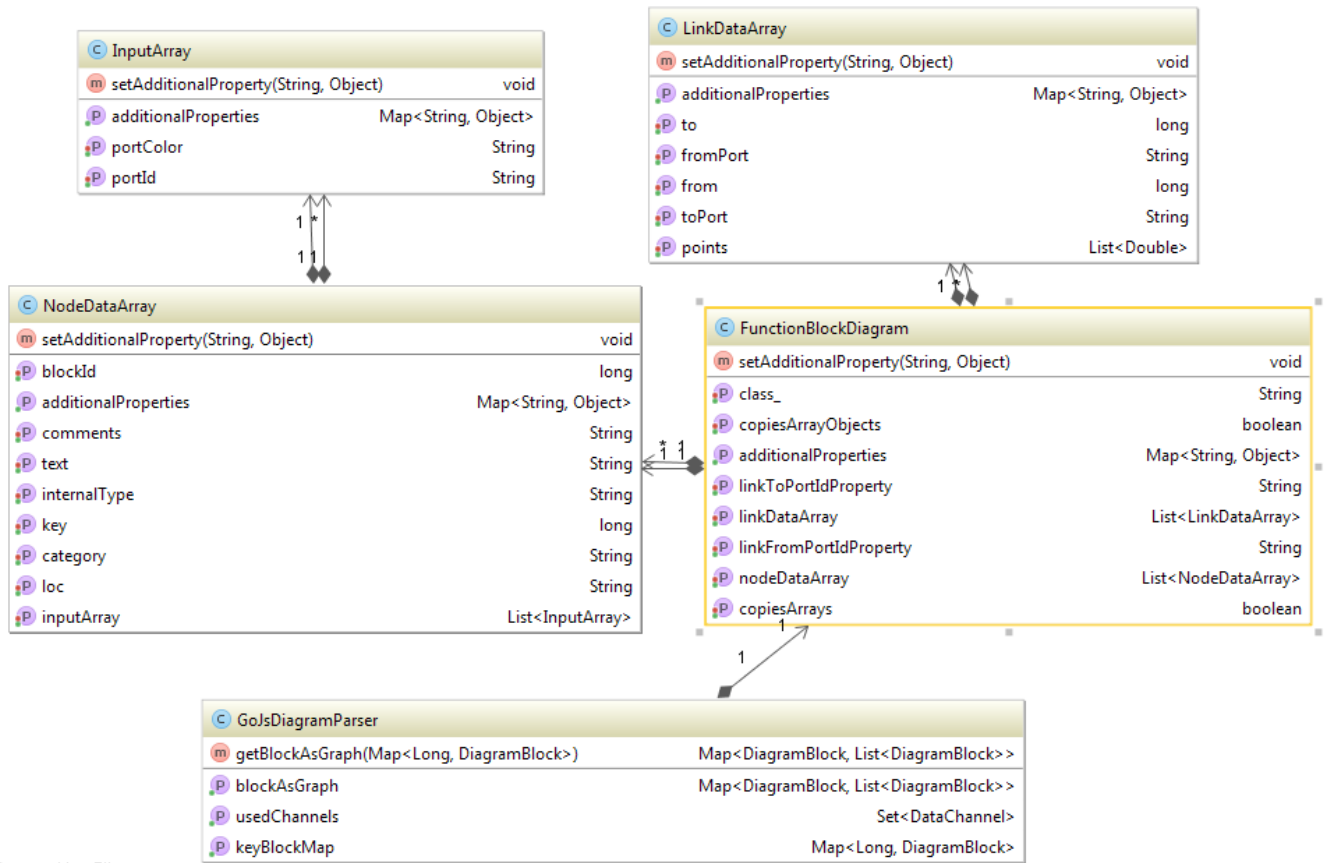


Figura 4.15: Diagrama claselor ce reprezintă modelul de date pentru diagramele GoJS

4.5 Serviciul de date

Serviciul de date, implementat prin Spring Bean-ul `DataService` permite abstractizarea modului în care `DataPoint`-urile sunt scrise în baza de date. În această implementare, datele sunt stocate în aceeași instanța de PostgreSQL ca și entitățile. Aceasta clasa folosește interogări SQL optimizate pentru a fi executate cât mai rapid, prin stocarea acestora direct pe serverul SQL.

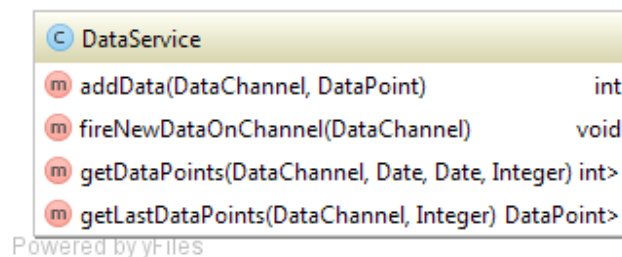


Figura 4.16: Interfața aplicației cu baza de date pentru informații

Această clasă asigură și execuția diagramelor din lista de `subscribedDiagrams` a canalului ce primește date. Această execuție se execută în mod **asincron**, pentru a nu bloca firul de execuție care primește date. Asincornicitatea este implementată printr-un pool de thread-uri care primesc task-uri de execuție, thread-ul apelant nefiind interesat de rezultatul execuției. Astfel, putem considera că implementarea pe mai multe fire de execuție este de tipul fire-and-forget.

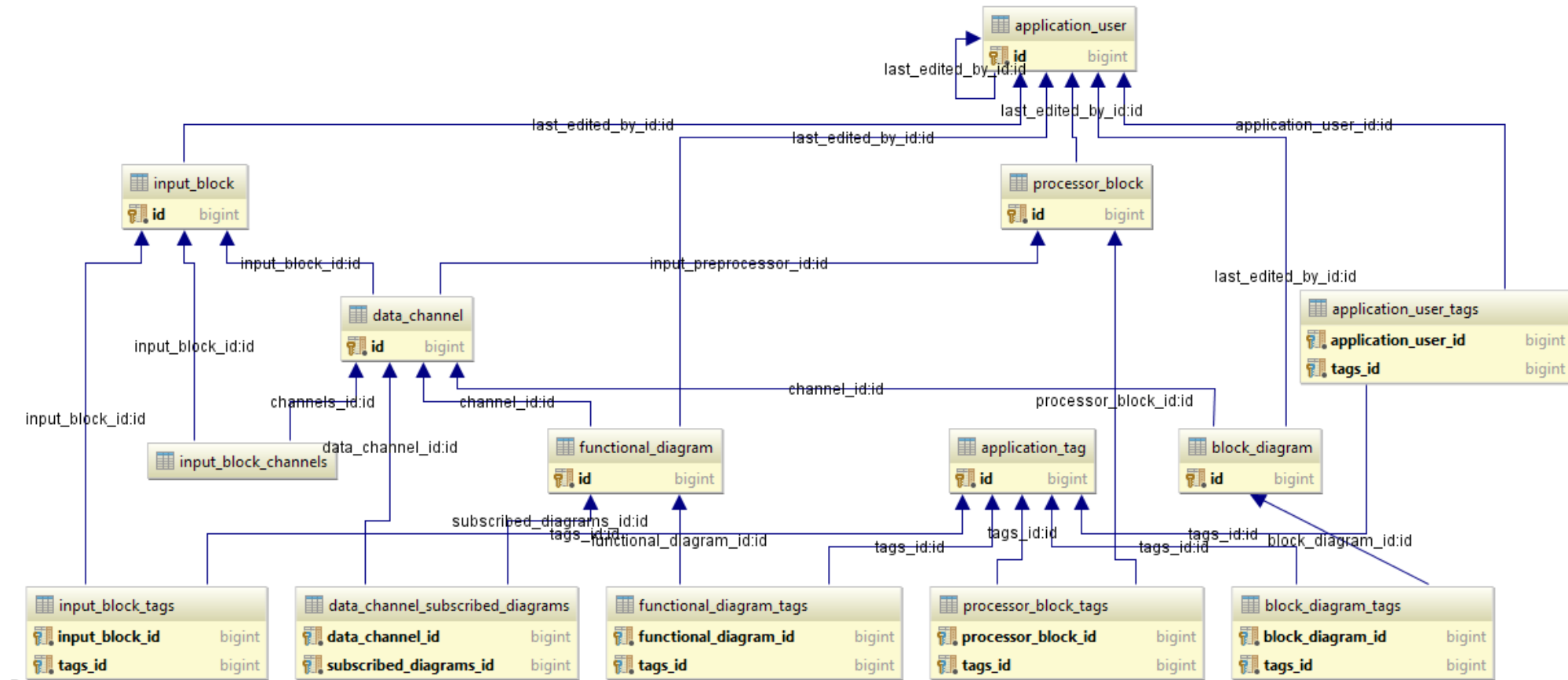
Pentru a garanta o metoda de management a firelor de execuție, s-au folosit capabilitățile native ale platformei Spring [24], prin adnotarea `@Async`.

4.6 Baza de date

După cum s-a discutat în capitolul despre arhitectura, aplicația are nevoie de mijloace de stocare pentru două tipuri de date: modelul entităților, și datele primite și procesate.

4.6.1 Stocarea entităților

Pentru stocarea entităților au fost investigate mai multe opțiuni de baze de date relaționale, PostgreSQL fiind aleasă pentru stabilitatea și posibilitatea de replicare volumelor mari de date, dar și pentru ușurința extensibilității[19]. Un alt avantaj al folosirii PostgreSQL este lipsa costului licenței, întrucât este o aplicație open-source cu o comunitate foarte dinamică.



Powered by yFiles

Figura 4.17: Modelul relațional al entităților din baza de date

Pentru generarea schemei bazei de date s-a folosit ORM-ul implicit din Spring Data, Hibernate [11] . Acesta permite generarea schema unei baze de date cu ajutorul programării orientate pe aspecte, prin adăugarea de adnotări pe clase Java. Această generare se realizează automat, de fiecare dată când modelul Java se modifica.

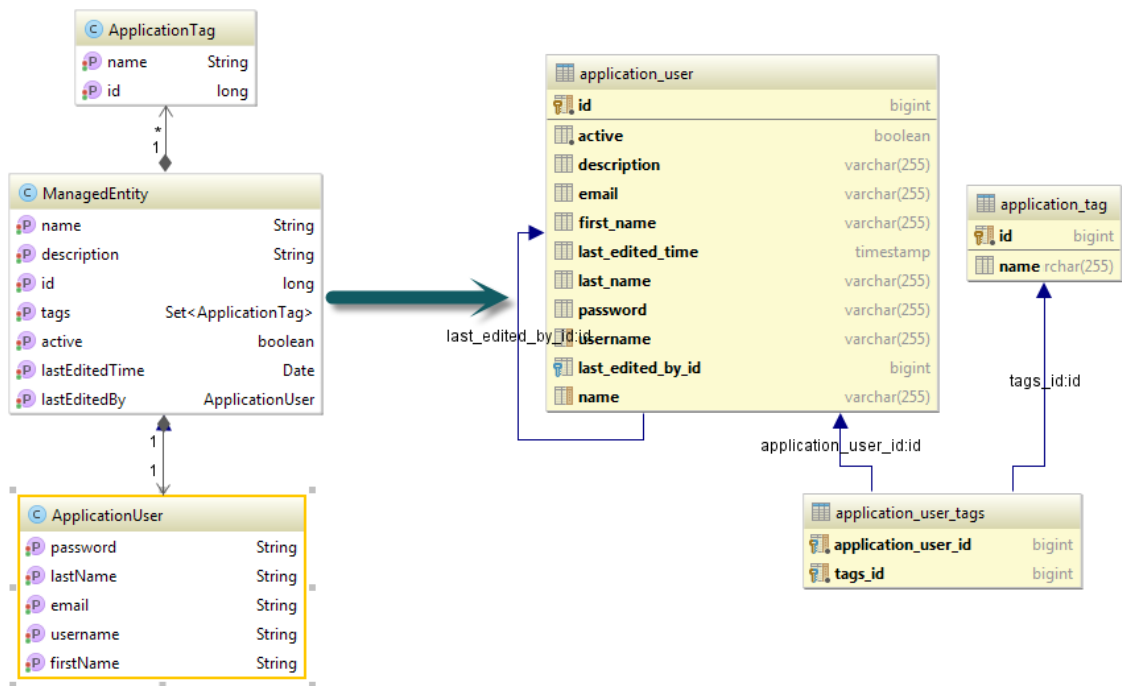


Figura 4.18: Transformarea claselor Java în relații din baza de date prin intermediul unui ORM

Capitolul 5

Exemplu de utilizare: Smart Home

Pentru a demonstra fiabilitatea platformei, acest capitol propune o soluție pentru problema automatizării unei case. O asemenea soluție ieftină și ușor de implementat lipsește încă de pe piață, sistemele profesionale fiind foarte scumpe, iar cele open-source sunt greu de instalat și configurat.

Urmărim deci rezolvarea automatizării unei case cu un dormitor și o sufragerie, în care au fost instalate următoarele dispozitive:

- Rețea de senzori inteligenți, distribuiți în toate camerele, capabili să ofere informații cu privire la numărul de oameni din casă.
- Sistem de iluminare inteligent, controlabil de la distanță.
- Un dispozitiv automat de închidere a ușii de la intrare.

Automatizarea funcționează astfel:

- Atunci când nu mai există oameni în casă, ușa trebuie încuiată;
- La un minut după ce camera este liberă lumina din acea cameră este stinsă;
- Dacă toți oamenii se află în dormitor, ușa se încuie automat la ora 00:00.

Au fost create următoarele blocuri de intrare, pentru fiecare dispozitiv:

- **Prezență Sufragerie:** indică dacă există persoane în sufragerie. Are un singur canal pe care trimite date de tip întreg despre numărul de oameni din cameră.
- **Prezență Dormitor:** senzorul din dormitor, analog cu Prezență Sufragerie.

Inputs + ▼								Q
# ^	Name ⇅	Description ⇅	Number Of Channels ^	Last edited by ⇅	Last Edited on ⇅	Edit	Disable	
104	Prezenta Sufragerie		1	petrisor.lacatus	2015-09-06 21:51:45.499			
105	Prezenta Dormitor		1	petrisor.lacatus	2015-09-06 21:52:06.657			

Figura 5.1: Intrările in aplicație

Următoarele diagrame au fost implementate:

- **Închidere ușă noaptea:** Comandă închiderea ușii după ora 00:00 dacă toți oamenii sunt în dormitor;
- **Securizare casa:** Comandă închiderea ușii când senzorii de prezență indică faptul că nu mai sunt persoane 5.3;
- **Oprire lumina dormitor:** Comandă oprirea luminii când nu mai e nimeni în dormitor;
- **Oprire lumina sufragerie:** Analog, pentru sufragerie;

Diagrams + ▼								S	Q
# ^	Name ⇅	Description ⇅	Last Edited By ^	Last Edited On ⇅	Edit	Disable			
10	Inchidere usa Noaptea		petrisor.lacatus	2015-09-06 21:53:34.002					
11	Securizare casa		petrisor.lacatus	2015-09-06 21:54:32.732					
12	Oprire lumina dormitor		petrisor.lacatus	2015-09-06 21:55:07.172					
13	Oprire lumina sufragerie		petrisor.lacatus	2015-09-06 21:55:35.588					

Figura 5.2: Diagramele implementate

Spre exemplu, în figura 5.3 este prezentată diagrama care realizează încuierea casei când nu mai exista persoane în interior. Aceasta folosește canalele celor doi senzori de prezență și trimite valorile acestora către blocul "**Adder**", care adună toate intrările numerice. Dacă suma acestor intrări este 0, blocul "**MaiMareCaZero**" va returna valoarea fals, care este citită de dispozitivul de pe ușă ce acționează mecanismul de închidere. În caz contrar, valoare returnată de blocul **MaiMareCaZero** este "adevarat", deci mecanismul nu va fi acționat. Putem astfel vedea cât de ușor este să implementezi o logica complexă folosind diagrame bloc.

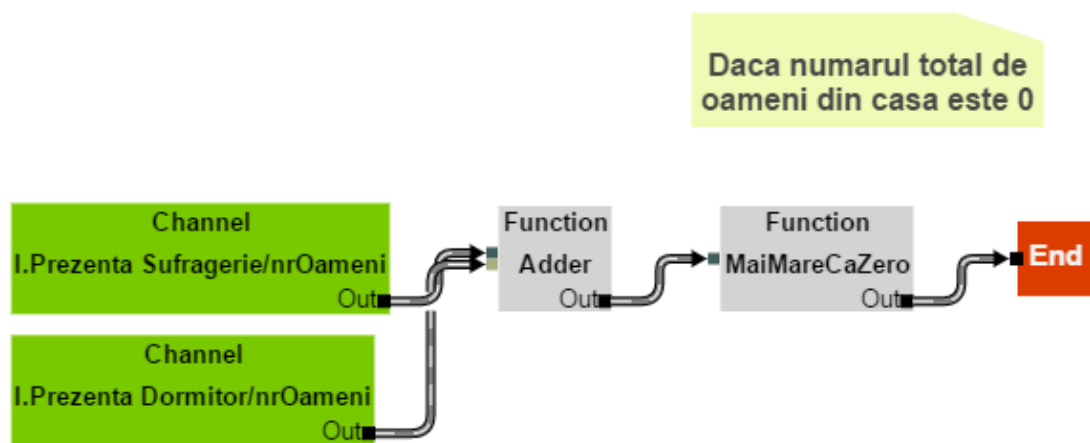


Figura 5.3: Diagrama pentru securizarea casei

Capitolul 6

Concluzii și Dezvoltării ulterioare

6.1 Concluzii

Această aplicație își propune să rezolve o problemă concretă, și anume permite interconectarea mai dispozitivelor inteligente, dispozitive ce se conectează la rândul lor la diverse surse de date. În această lume emergentă, serviciile sunt din ce în ce mai specializate, satisfăcând în mod complet o nișă. Orice persoană poate achiziționa câțiva senzori inteligenți, însă, pentru a obține valoare din acei senzori, datele trebuie colectate, procesate și stocate.

Soluția propusă răspunde astfel atât la problema colectării datelor din surse variate, cât și la problema transformării acelor date în date utilizabile de alte sisteme, generând informații din date neprelucrate.

6.2 Dezvoltării ulterioare

Un aspect foarte important ce poate face scopul unei dezvoltări ulterioare este scalarea platformei într-o instanță cloud, ce poate fi folosită cu o arhitectura de tipul platform as a service (PaaS). Astfel, un utilizator poate externaliza serviciul de procesare a datelor, singura lui grijă fiind colectarea datelor în sistem.

Împreună cu dezvoltarea de mai sus, o alta direcție este alinierea cât mai bună la conceptele din IoT (Internet of Things). Aceasta aliniere s-ar putea realiza prin transformarea blocului de intrare în "thing"-uri, cu proprietăți și servicii. Tot în această direcție de dezvoltare s-ar putea include dezvoltarea de agenți capabili să interfațeze între protocoale de date proprietare, și soluția propusă. Spre exemplu, implementarea unui client MQTT ar permite conectivitatea către o întreagă gama de dispozitive care respectă acest standard.

În vederea ușurării dezvoltării aplicațiilor pe aceasta platformă trebuie suplimentat numărul de blocuri de procesare implicite existente în sistem.

Bibliografie

- [1] *Bootstrap 3 Typeahead*. URL: <https://github.com/bassjobsen/Bootstrap-3-Typeahead> (visited on 09/02/2015).
- [2] *Bootstrap Tags Input*. URL: <http://timschlechter.github.io/bootstrap-tagsinput/examples/> (visited on 09/04/2015).
- [3] *CodeMirror*. URL: <https://codemirror.net/index.html> (visited on 09/02/2015).
- [4] *DAGs and Topological Ordering*. URL: <http://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf> (visited on 08/20/2015).
- [5] *Dygraphs is a fast, flexible open source JavaScript charting library*. URL: <http://dygraphs.com/> (visited on 09/03/2015).
- [6] *Facts and Forecasts: Billions of Things, Trillions of Dollars*. URL: <http://www.siemens.com/innovation/en/home/pictures-of-the-future/digitalization-and-software/internet-of-things-facts-and-forecasts.html> (visited on 09/05/2015).
- [7] *Function Blocks*. URL: <http://www.functionblocks.org/index.html> (visited on 08/20/2015).
- [8] *GE FANUC Function Block Diagram Lab*. URL: <http://geplc.com/downloads/Labs/GFS-384%20M03%20Function%20Block%20Diagram.pdf> (visited on 08/20/2015).
- [9] *GoJS Interactive Diagrams for JavaScript and HTML*. URL: <http://gojs.net/latest/index.html> (visited on 09/03/2015).
- [10] Deloitte Group. *Industry 4.0: Challenges and solutions for the digital transformation and use of exponential technologies*. 2014.
- [11] *Hibernate ORM: Idiomatic persistence for Java and relational databases*. URL: <http://hibernate.org/orm/> (visited on 09/05/2015).
- [12] *IEEE Standard for Floating-Point Arithmetic*. Aug. 2008, pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

- [13] *Industrial Use Cases of Distributed Intelligent Automation*. IEC 61499. TC 65/SC 65B - Measurement and control devices, Jan. 2011. URL: http://www.vyatkin.org/publ/IES_Mag_1499.pdf.
- [14] A. B. Kahn. *Topological Sorting of Large Networks*. Vol. 5. 11. New York, NY, USA: ACM, Nov. 1962, pp. 558–562. DOI: 10.1145/368996.369025. URL: <http://doi.acm.org/10.1145/368996.369025>.
- [15] *Logix5000 Controllers Function Block Diagram Programming Manual*. URL: http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm009_-en-p.pdf (visited on 08/20/2015).
- [16] *openHAB: empowering the smart home*. URL: <http://www.openhab.org/features/introduction.html> (visited on 09/05/2015).
- [17] David J. Pearce and Paul H. J. Kelly. *A Dynamic Topological Sort Algorithm for Directed Acyclic Graphs*. Vol. 11. New York, NY, USA: ACM, Feb. 2007. DOI: 10.1145/1187436.1210590. URL: <http://doi.acm.org/10.1145/1187436.1210590>.
- [18] Michael E. Porter and James E. Heppelmann. *How Smart, Connected Products Are Transforming Competition*. 11. Harvard, Nov. 2014, pp. 64–88. URL: <http://www.hbs.edu/faculty/Pages/item.aspx?num=48195>.
- [19] *PostgreSQL: Advantages*. URL: <http://www.postgresql.org/about/advantages/> (visited on 09/05/2015).
- [20] *Programmable controllers - Part 3: Programming languages*. IEC. TC 65/SC 65B - Measurement and control devices, July 2013. URL: http://www.dee.ufrj.br/control_e_automatico/cursos/IEC61131-3_Programming_Industrial_Automation_Systems.pdf.
- [21] *Scripting for the Java™Platform*. JCP 223. Sun Microsystems, Inc., Dec. 2006, pp. 1–140. URL: <https://www.jcp.org/en/jsr/detail?id=223>.
- [22] *Spring Boot*. URL: <http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/> (visited on 09/02/2015).
- [23] *Spring Framework Case Studies*. URL: <http://pivotal.io/resources/1/case-studies> (visited on 09/02/2015).
- [24] *Spring: Task Execution and Scheduling*. URL: <http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/scheduling.html> (visited on 09/05/2015).

- [25] *Supporting Dynamically Typed Languages on the JavaTM Platform*. JCP 292. Sun Microsystems, Inc., July 2011. URL: <https://www.jcp.org/en/jsr/detail?id=292>.
- [26] *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627. RFC Editor, July 2006, pp. 1–10. URL: <http://www.ietf.org/rfc/rfc4627.txt>.
- [27] *The Ruby Programming Language on the JVM*. URL: <http://jruby.org/> (visited on 09/04/2015).
- [28] *The Scalable Time Series Database*. URL: <http://opentsdb.net/index.html> (visited on 08/20/2015).
- [29] *TM241 Programming Manual*. URL: <http://www.kongzhi.net/files/download.php?id=8362> (visited on 08/20/2015).
- [30] *Uniform Resource Identifier (URI): Generic Syntax*. RFC 4627. RFC Editor, Jan. 2005, pp. 1–61. URL: <https://tools.ietf.org/html/rfc3986>.