Sistem de achiziţie, procesare si distribuite a datelor

Petrişor-Ştefan Lăcătuş

Septembrie 2015

Automatică şi Ingineria Sistemelor
Facultatea de Automatica si Calculatoare

Coordonator: Ş.L. Andreea Udrea

Cuprins

Capitolul 1

Introducere

Capitolul 2

Arhitectura soluţiei

2.1 Prezentare generală

Alegria a fost conceputa ca o platforma de dezvoltare rapida, bazata pe modele. Prin folosirea unor metode de programare vizuala, cu blocuri refolosibile in mai multe aplicaţii diferite, utilizatorul poate sa îşi concentreze resursele asupra soluţiei finale, abstractizând detaliile implementării. Aplicaţia a fost construita pe baza unei arhitecturi modulare, cu module cat mai puţin cuplate, care sa permită modificări rapide si testarea modulelor individual. Urmărind aceasta gândire modulare, au fost s-au identificat 4 componente esenţiale:

- **Baza de date** Asigură stocarea datelor, dar si a entităţilor existente in aplicaţie;
- **Interfaţa web pentru management** O interfaţă uşor de utilizat care sa permită utilizatorului să manipuleze canale, diagrame, blocuri de procesare, dar si alţi utilizatori
- **API-ul pentru date** Un API specializat pentru adăugare de date, achiziţionarea datelor, dar si sa permită altor dispozitive sa fie notificate de fiecare data când apar date noi.
- **Elemente de procesare** Asigură procesarea datelor, atât in cadrul blocurilor de procesare, dar si in cadrul diagramelor funcţionale.

In vederea implementării sistemului s-au identificat următoarele elemente componente esenţiale, componente ce reprezinta elementele constructive a sistemului. Acestea, au reprezentare atât in baza de date, ca entităţi, cat si in aplicaţia Java, ca clase. Identificarea acestor elemente s-a făcut pe baza analizei cazurilor de utilizare a produsului, in care s-au investigat metodele prin actorii interacţionează cu sistemul.

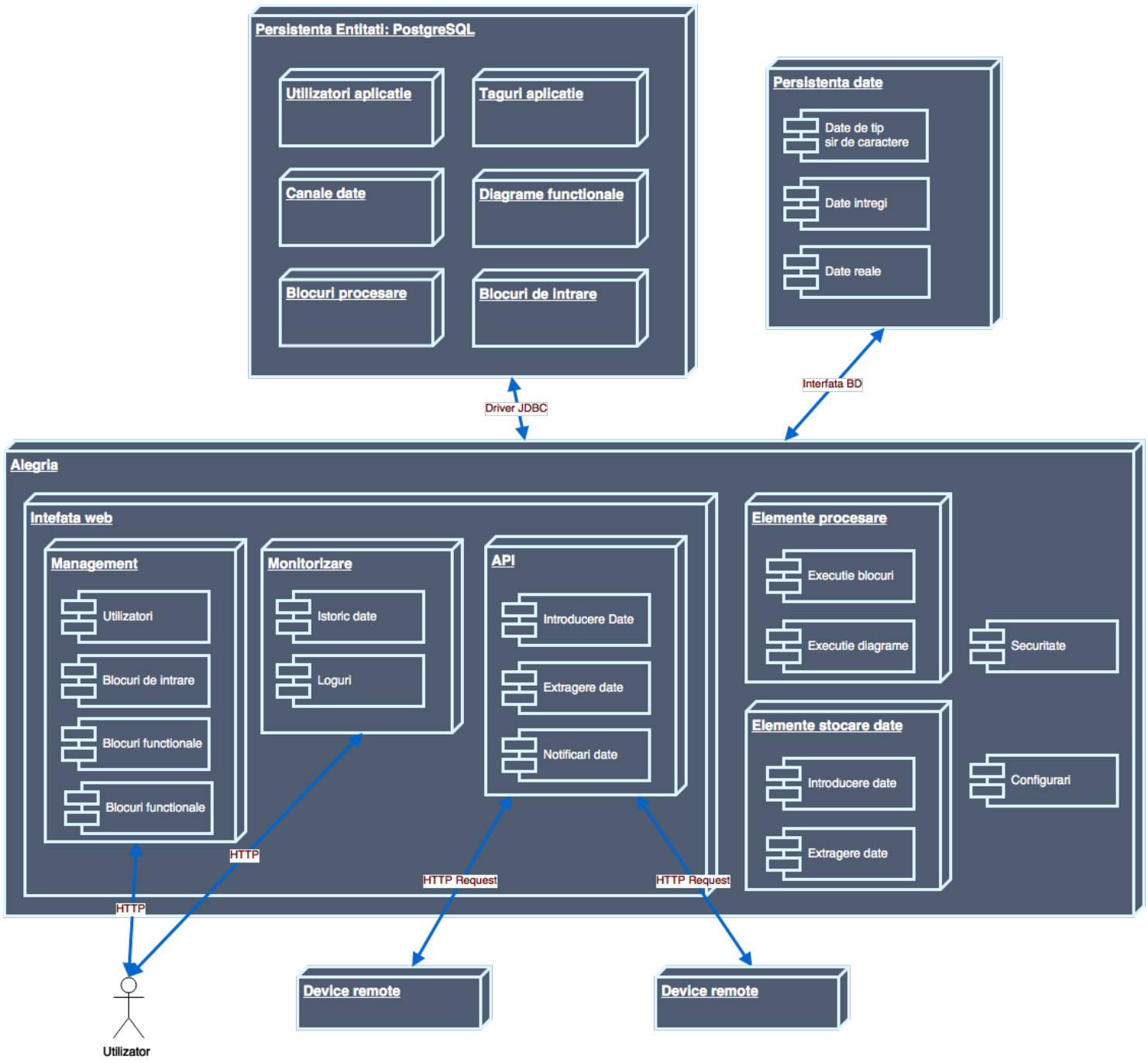

Figura 2.1: Arhitectura generala a aplicaţiei

- **Canalul de date:** reprezinta elementul de baza a sistemului, care asigura recepţia, persistenta si emiterea de date. Datele dintr-un canal trebuie sa respecte un format prestabilit la crearea canalului. Pentru transformarea datelor in formatul stabilit, se poate introduce un bloc de pre-procesarea care transforma datele din un format brut in formatul standard.

- **Blocul de intrare:** elementul de mapează un element din lumea real in interiorul platformei. Blocurile de intrare permit gruparea mai multor canale de date într-o structura unica.

- **Blocul de procesare:** elementul dinamic al aplicaţiei, ce aplica transformări asupra datelor. Un bloc de procesare primeşte ca intrări mai multe canale de date, si are la ieşire un alt canal de date. Utilizatorul poate folosi blocuri standard, existente in sistem, sau poate implementa blocuri noi direct in interfaţa programului.

- **Diagrama funcţie bloc(FBD):** foloseşte blocuri de intrare, canale de date si blocuri de procesare pentru a descrie o funcţie complexa intre intrări si o ieşire. Aceste diagrame folosesc la date aflate pe canale de date, care sunt trimise către blocuri de procesare si, la final se obţine un singur rezultat care este salvat pe un canal de date.

2.1.1 Baza de date

Fiind vorba despre o aplicaţie puternic bazata pe date, aceasta are nevoie de un nivel de persistenta de înaltă performaţă. Urmărind arhitectura propusă din figura 2.1, putem identifica doua cazuri de utilizare pentru baza de date:

- Stocarea modelului entităţilor: fiecare entitate descrisa in lista de mai sus trebuie stocata in baza de date într-o structura relaţională. Entităţile sunt puternic interconectate, iar o baza de date relaţională, de tip SQL este recomandata in acest caz. Din punct de vedere a dimensiunii setului de date, chiar si in aplicaţiile de mare complexitate, este vorba despre doar câteva milioane de înregistrări, factorul care face acest număr să crească fiind conectarea a tot mai mult dispozitive, ce duce la din ce in ce mai multe canale de date. Astfel, stocarea entităţilor nu va aduce probleme de performaţă.

- Stocarea datelor: in baza de date vor fi stocate atât datele primite pe fiecare canal asociat unui bloc de intrare, cat si datele procesate de diagrame. Aceste date au un puternic caracter istoric, reprezentând o serie de timp, in care se retine, pentru fiecare punct de date, valoarea la un anumit moment. Problema stocării acestor date este una mai complicata, datorita necesităţii unei puternice scalari a bazei de date.

Această problemă reprezinta un caz de utilizare pentru o baza de date NoSQL, sau chiar o baza de date specializata in stocarea seriilor de timp.[1]

2.1.2 Aplicaţia Java

Legătura dintre baza de date si utilizatorii finali se face prin intermediul unei aplicaţii Java complexe, care este obiectul acestui proiect. Aplicaţia conţine toată logica platformei, de la operaţii asupra entităţilor din baza de date, la adăugarea, si extragerea datelor, cat si pentru procesarea datelor. Separarea modulelor s-a făcut pe baza scopului acestora:

- **Administrare**: pentru administrarea entităţilor din baza de date. Aceste module permit operaţii de căutare si afişare, dar si de creere, editare si ştergere a utilizatorilor, a blocurilor de intrare si funcţionale precum si a diagramelor. Fiecare dispune de o interfaţă HTML5 in care moderna.
- **Monitorizare**: permit monitorizarea execuţiei aplicaţiei, de la vizualizat loguri pentru a diagnostica probleme, la realizarea de grafice a datelor pe anumite canale. Tot aici este disponibila si funcţia de a exporta date in formate uzuale, ca CSV sau fişiere Microsoft Excel.
- **API**: aplicaţia dispune si de un API pentru a fi folosita programatic de către alte aplicaţii externe. Acesta poate fi considerat ca fiind format din doua componente: serviciile pentru administrarea entităţilor, si cele pentru adăugarea si extragerea datelor.
- **Elemente de procesare**: împărţite in doua subcategorii: cele pentru procesarea blocurilor de intrare si de procesare, si cele pentru procesarea diagramelor.
- **Elemente stocare date**: permit interfaţarea cu sursele de date. Acestea asigura servicii de introducere si extragere a datelor, printr-o interfaţă abstracta, care nu tine cont de modul in care baza de date este implementata.
- **Alte module**: asigura, printre altele securitatea aplicaţiei.

2.2 Entităţi

2.2.1 Punctul de date

Punctul de date reprezinta elementul constructiv al sistemul, care este obiectul procesării, stocării si distribuţiei este punctul de date. Sistemul accepta intern date in formatele:

[1] *The Scalable Time Series Database*. URL: `http://opentsdb.net/index.html` (visited on 08/20/2015).

- **Întreg**: numere de la -2^{63} la $2^{63} - 1$, fără virgula, foloseşte *Long* pentru reprezentare interna;
- **Real**: numere cu virgula, având dubla precizie, reprezentate cu 64-bit conform standardului[2] IEEE 754 foloseşte *Double* pentru reprezentare interna;
- **Sir de caractere**: Un sir de fără limite a lungimii, care trebuie formatat conform.[3]
- **Obiect**: Un obiect Java serializat in text. Intern, asemănător cu tipul de date String.

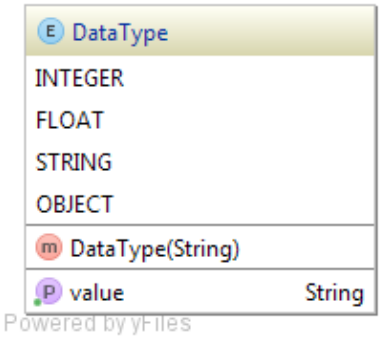

Figura 2.2: Tipurile de date acceptate in sistem

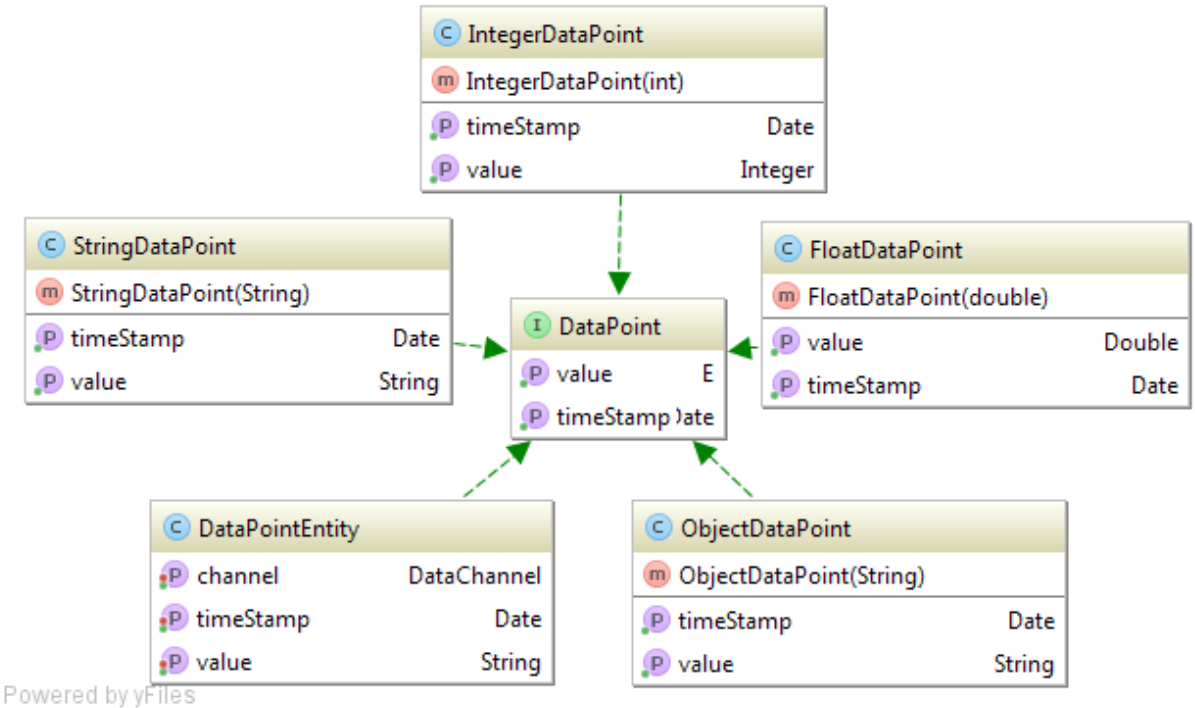

Figura 2.3: Clasele care implementează interfaţa DataPoint

[2] *IEEE Standard for Floating-Point Arithmetic*. Aug. 2008, pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[3] *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.

2.2.2 Canalul de date

Canalul de date este entitatea care asigura "curgerea" datelor prin sistem. Orice punct de date din sistem aparţine unui canal, acest lucru fiind realizat drept constrângere atât la nivelul aplicaţiei, cat si la nivelul bazei de date.

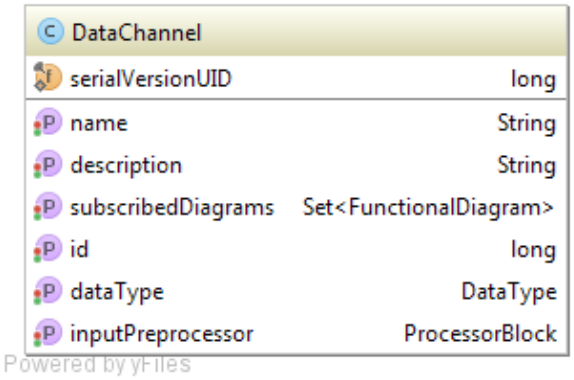

Figura 2.4: Clasa DataChannel

Canalul este si mijlocul prin care utilizatorul interacţionează cu punctele de date. Când un dispozitiv adaugă date noi in sistem, acestea sunt ataşate unui canal, care poate fi folosit ca una dintre intrările unei diagrame de blocuri funcţionale. De asemenea, datele procesate de o diagrama sunt ataşate unui canal, permiţând apoi accesul pentru extragerea de datelor deja existente, si pentru a primi notificări de fiecare data când pe un canal apar informaţii noi.

Un canal mai poate avea si un bloc de preprocesare ataşat. Acesta este executat de fiecare data când date noi încearcă sa fie introduse in canal, permiţând validarea si transformarea datelor brute in date ce respecta tipul de date al canalului.

2.2.3 Blocul de intrare

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

Spre exemplu, o sursă de alimentare neîntreruptibilă care este inteligenta si conectata din figura 2.6 poate fi privită ca un bloc de intrare, iar fiecare senzor de pe aceasta fiind canal de date. Dispozitivul inteligent devine astfel conectat la platforma si acesta poate sa trimită date către aceasta. Pentru ca datele pot sa aibă perioade de eşantionare diferite, dispozitivul trimite date către unul sau mai multe canale, fără sa fie forţat sa trimită date pentru toate canalele odată.

Un alt mod in care blocurile de intrare pot fi privite este ca obiecte, sau "Things" in cadrul Internetului Tuturor Lucrurilor (IoT). Astfel, putem privi blocurile de intrare ca dispozitive ce monitorizează bătăile inimii sau activitatea celebrară, ca automobile cu reţele complexe de senzori, sau chiar aplicaţii business ce generează date in timp real. Prin acest mijloc, dispozitive inteligente se pot conecta in platforma, fie trimiţând direct date, fie prin intermediul unui agent care se afla pe dispozitiv, agent ce funcţionează ca un gateway.

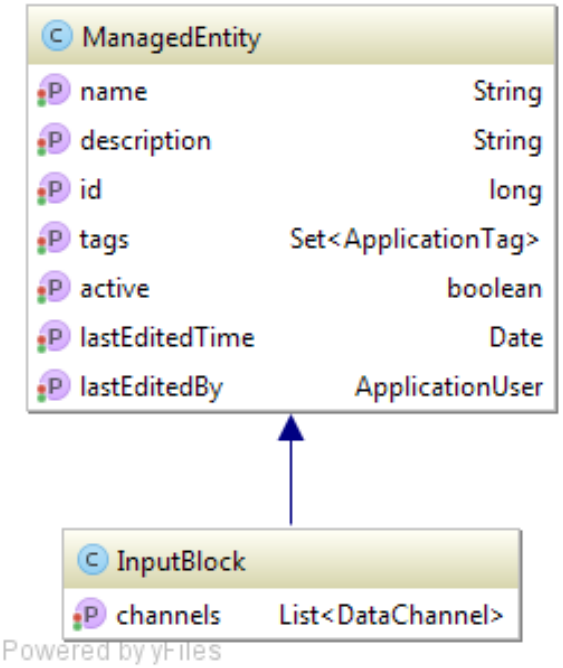

Figura 2.5: Clasa InputBlock

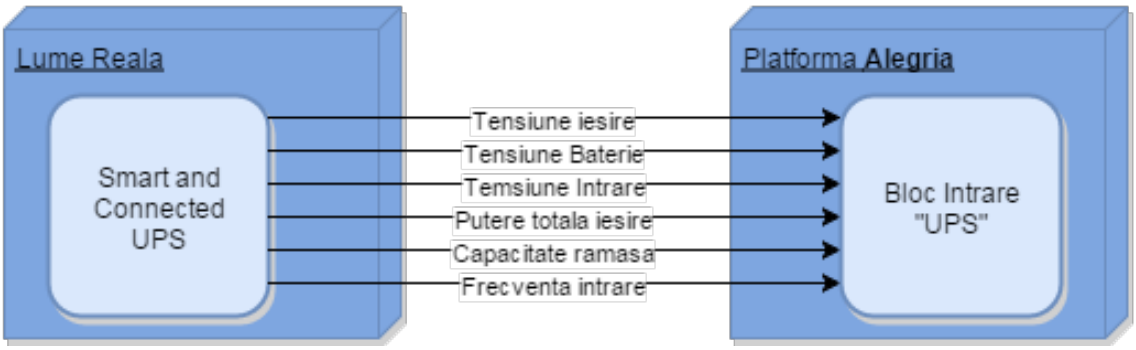

Figura 2.6: Modelarea unui UPS inteligent in Alegria

2.2.4 Blocul de procesare

Blocurile de procesare realizează operaţiile transformare a datelor. Din punct de vedere funcţional, acestea au la intrare ultimele date introduse de pe mai multe canale si returnează un singur punct de date, comportându-se ca un element de tip "back-box".[4] Pentru implementare, utilizatorul foloseşte limbaje dinamice moderne, ca JavaScript sau Ruby, scriind funcţiile direct in interfaţa web. Aceste funcţii sunt rulate de către server. Conceptul de blocuri de procesare reprezinta o implementare a blocurilor de operaţii definite in standardul.[5]

Blocuri pre-implementate exista in orice instanta a platformei care permit rezolvarea de probleme complexe cu cunoştinţe minime de programare. Un alt aspect important al blocurilor de procesare, este ca ele sunt complet independente, ele oferind doar mijloace de prelucrare a unor date abstracte. Aceasta abstractizare permite refolosirea lor in mai multe proiecte. Spre exemplu, un bloc care face media tuturor intrărilor nu tine cont de originea datelor.

Blocurile **nu au memorie statica**. Ele pot folosi informaţii exterioare, cum ar fi timpul curent al zilei, sau alte informaţii din sistem, însa ele nu au stare interioara. Astfel, se poate considera ca blocurile reprezinta un element combinaţional, si nu unul secvenţial.

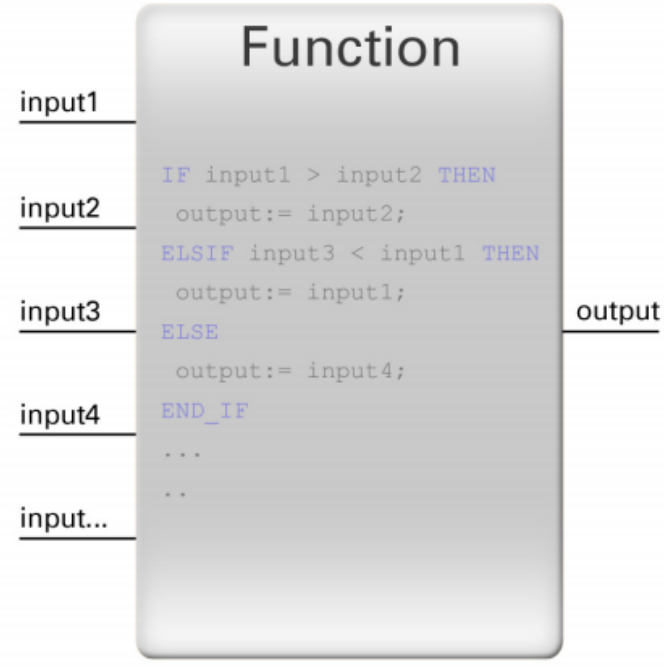

Figura 2.7: Exemplu bloc de procesare

[4] *Function Blocks*. URL: `http://www.functionblocks.org/index.html` (visited on 08/20/2015).

[5] *Programmable controllers - Part 3: Programming languages*. IEC. TC 65/SC 65B - Measurement and control devices, July 2013. URL: `http://www.dee.ufrj.br/controle_automatico/cursos/IEC61131-3_Programming_Industrial_Automation_Systems.pdf`, Apendix C.

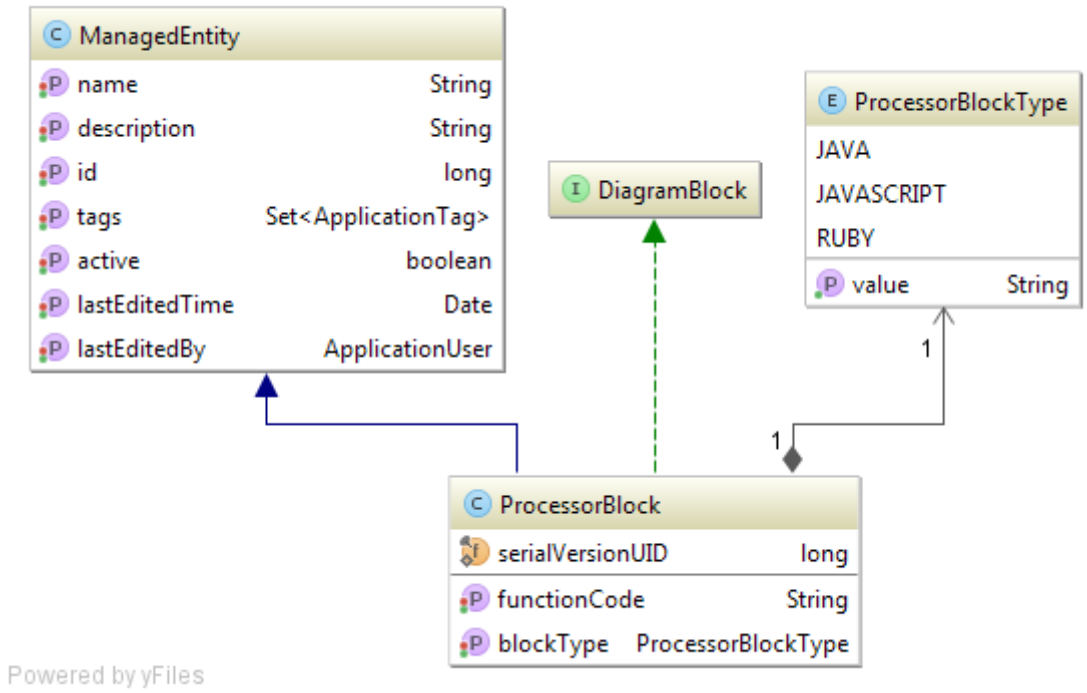

Figura 2.8: Clasa ProcessorBlock

2.2.5 Diagrama funcţie bloc(FBD)

Diagramele funcţie bloc reprezinta o implementare a unuia din cele patru limbare de programare pentru automate programabile, specificate de standardul IEC 61131-3:2013.[6] FBD-ul este un limbaj de control al procesului, in mod normal, toate blocurile de procesare dintr-o diagrama fiind executate. In cel mai simplu caz de utilizare, un FBD realizează următoarele operaţii:

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

Legăturile dintre blocurile de procesare sunt unidirecţionale. Un bloc de procesare poate trimite rezultatul al unul sau mai multe blocuri, iar o intrare poate fi conectata la o singura ieşire. Transmiterea de date se face fără întârzieri. Diagramele sunt executate in ordinea definita de ordonarea topologica a nodurilor din graful ce defineşte diagrama.[7]

[6] *Programmable controllers - Part 3: Programming languages*, pp. 128-140.

[7] *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`, pp. 20-50.

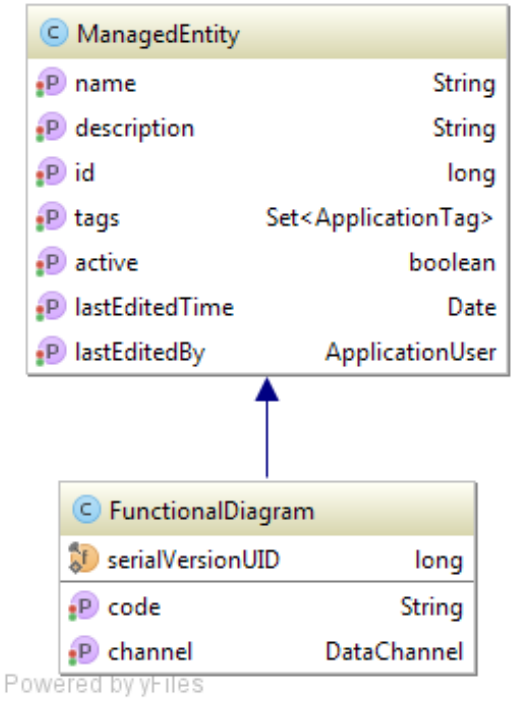

Figura 2.9: Clasa FunctionalDiagram

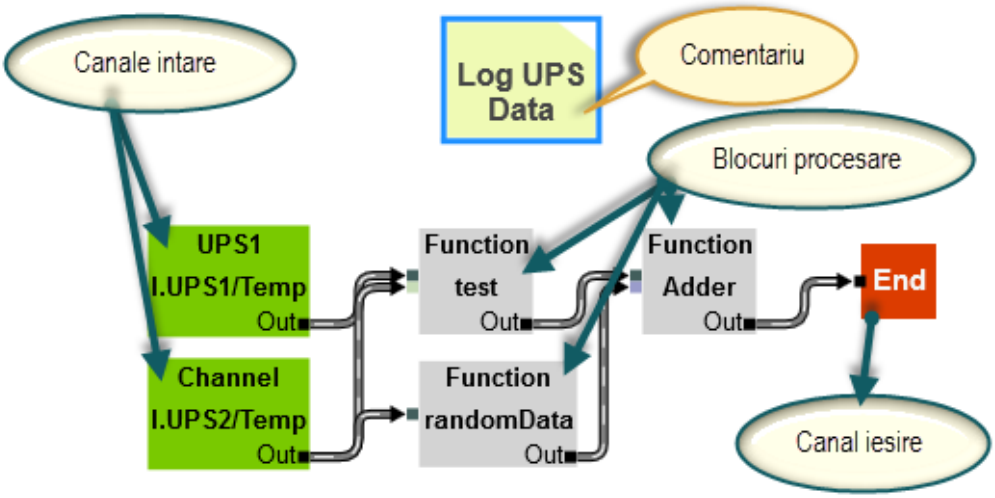

Figura 2.10: Exemplu diagrama funcţională

2.3 Primirea datelor

Primirea datelor se face prin intermediul unei interfeţe de transfer a stării (REST). Mai multe formate sunt incluse pentru integrarea mai uşoară cu sisteme deja existente. Astfel,

au fost implementate mai multe procesoare care primesc date atât într-un format special, cat si in formate standard in industrie. Astfel, doua modalităţi de trimitere a datelor exista in sistem:

- Trimitere către un singur canal, un singur punct odată: pe baza serviciului */api/put/inputId/channelId/data*. Acest serviciu adaugă un singur punct in baza de date, la momentul curent. Folosit pentru sisteme care trimit date rar, si nu trebuie sa se tina cont de data locala de pe device-ul care a trimis punctul de date.

- In formatul standard folosit de openTDSB in care au fost introduse următoarele modificări care păstrează totuşi compatibilitatea: metricile reprezinta numele canalului, iar tag-urile sunt opţionale. Se acceptat atât formatul in care într-o cerere se afla un singur punct, cat si formatul cu o lista de puncte. Canalele dintr-o cere multidimensionala nu trebuie sa facă parte din acelaşi bloc de intrare. Acest mod de introducere a datelor este sugerat pentru sistemele care folosesc mai multe canale de date si care trimit seturi de date mai mari printr-o singura cerere. Spre exemplu, un dispozitiv poate trimite date de pe mai multi senzori, si poate stoca local mai multe măsurători pe acelaşi senzor pentru a trimite toate datele odată.

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

1. Se interoghează baza de date pentru detalii privind canalul ce tocmai a primit date.
2. Dacă un preprocesor exista pe canalul specificat, atunci el este încărcat.
3. Se executa preprocesorul cu punctul de date primit
4. Se salvează rezultatul in baza de date.
5. Asincron, se lansează toate diagramele care trebuie sa se execute atunci când se primesc date noi pe acest canal.
6. Asincron, se informează ascultătorii ca canalul a primit date noi.

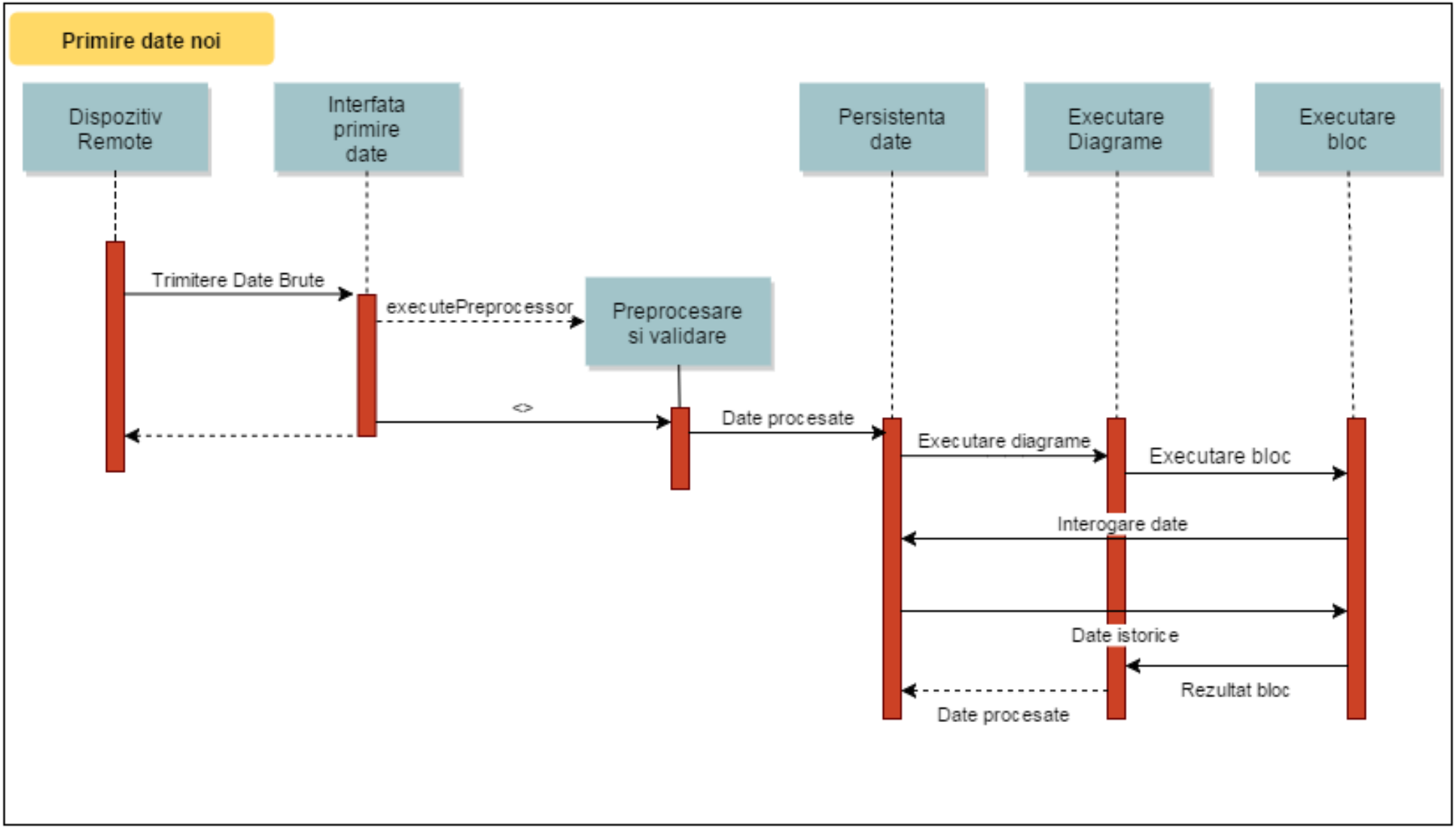

Figura 2.11: Diagrama de secvente pentru introducerea de noi date si execuţia diagramelor

2.4 Executarea unei diagrame

Figura 2.11 prezintă si modul in care o diagrama este executată. Diagramele sunt lansate in execuţie de fiecare dată când un canal folosit in ea primeşte informaţii noi. Dacă doua canale primesc date în acelaşi timp, atunci diagrama va fi lansata in execuţie tot de doua ori, pentru ambele puncte de date primite.

2.4.1 Ordinea execuţiei blocurilor

Problema ordinii execuţiei unei FBD este intens dezbătută atât in literatura, cat si în aplicaţiile industriale. Cum standardul *IEC61131-3*[8] nu propune o soluţie pentru ordinea de execuţie, producătorii industriali folosesc metode proprii, de la separarea blocurilor într-un tabel si executarea de la stânga la dreapta, sus in jos,[9] la definirea manuala a ordinii execuţiei[10] sau folosind algoritmi care determina automat ordinea de execuţie.[11]

In implementarea din Alegria s-a ales proiectarea unei metode automate pentru depistarea ordinii in care diagrama trebuie executata. Deoarece o diagrama reprezinta un graf aciclic orientat, prima etapa a execuţiei este transformarea într-un graf reprezentat prin lista de adiacenta, unde fiecare bloc de intrare, bloc si de procesare reprezinta un nod. Din aceasta reprezentare se omit blocurile de comentarii. Odată ce transformarea a fost efectuata cu succes se încerca aplicarea unui algoritm de sortare topologica[12] a grafului obţinut. Aceasta sortare implica găsirea unei ordini astfel încât pentru orice arc orientat uv de la u la v, u este înaintea lui v. Matematic problema poate fi formulata astfel:

Definiţie 1. *O **ordine topologica**, notata 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 $x \to y \in E$.*

Mai multi algoritmi pentru efectuarea unei asemenea sortări exista in literatura,[13] însa, deoarece grafurile in discuţie sunt statice, la care nu se adaugă sau se şterg noduri, s-a ales un algoritm clasic, stabil din punct de vedere numeric descris de Kahn in 1962.[14]

Simplificat, algoritmul implementat urmăreşte următorii paşi:

[8] *Programmable controllers - Part 3: Programming languages*.

[9] *TM241 Programming Manual*. URL: `http://www.kongzhi.net/files/download.php?id=8362` (visited on 08/20/2015), p. 11.

[10] *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), p. 11.

[11] *GE FANUC Function Block Diagram Lab*. URL: `http://geplc.com/downloads/Labs/GFS-384%20M03%20Function%20Block%20Diagram.pdf` (visited on 08/20/2015), p. 5.

[12] *DAGs and Topological Ordering*. URL: `http://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf` (visited on 08/20/2015).

[13] 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`.

[14] A. B. Kahn. *Topological Sorting of Large Networks*. Vol. 5. 11. New York, NY, USA: ACM, Nov.

1. Identifica toate nodurile spre care nu vine nici un arc. Valoarea acestor noduri va fi 0. In cazul diagramelor, este vorba de toate canalele de intrare, si de blocurile de procesare care nu au nici o intrare. Dacă aceste noduri nu exista, înseamnă ca graful nu respecta condiţia de graf aciclic, deci acesta nu va putea fi executat.

2. Se alege unul din din nodurile găsite mai sus.

3. Se şterge acest nod de valoare unu, împreună cu toate arcele care ies din el.

4. Se repeta paşii 1 si 2 pana când nu mai exista noduri in graf.

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

Algoritmul 1: Algoritmul lui Khan pentru sortare topologica

Data: Un graf orientat aciclic reprezentat prin lista de adiacenta
Result: Lista nodurilor ordonate topologic

```
1  L ← Lista goala ce va conţine nodurile sortate ;
2  S ← Lista tuturor nodurilor spre care nu exista nici un arc ;
3  while S conţine elemente do
4      şterge nodul n din S ;
5      introdu nodul n in L ;
6      foreach nod m care are un arc e de la n la m do
7          şterge arcul e din graf;
8          if m nu mai are arce spre el then
9              inserează m in S
10 if mai exista arce in graf then
11     return Eroare: Graful are cel puţin un ciclu
12 else
13     return L (graful sortat topologic)
```

Astfel, pentru diagrama din figura 2.10, o posibila ordine de execuţie este descrisa in figura 2.12.

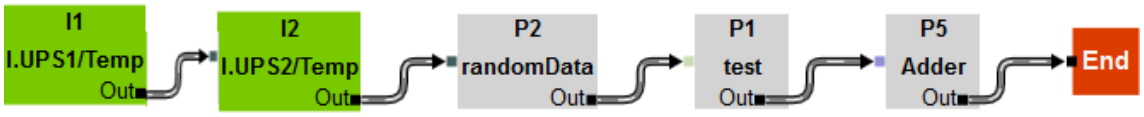

Figura 2.12: Ordinea execuţiei pentru diagrama din figura 2.10

1962, pp. 558–562. DOI: 10.1145/368996.369025. URL: http://doi.acm.org/10.1145/368996.369025.

După cum se observa si in alg. 1 acesta poate detecta grafuri care conţin cicluri si nu pot fi rezolvate. Aceasta verificare permite detectarea cazurilor, precum cel din diagrama 2.13, si permite informarea utilizatorului pentru ca acesta sa rezolve problema.

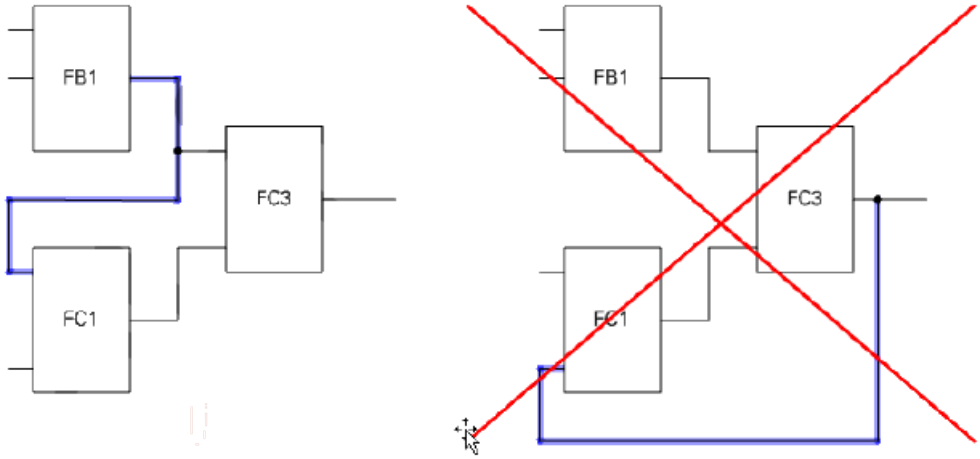

Figura 2.13: Diagrama ce conţine cicluri

Deşi ciclurile la nivel de diagrama nu sunt permise, cele la nivel de canal sunt. Astfel, ieşirea unei diagrame poate fi setata ca intrare pentru aceeaşi diagrama, permiţând executarea într-o bucla, deoarece odată ce o diagrama se termina de executat si salvează rezultatul pe canal, aceasta se relansează in execuţie cu noua informaţie. Cu astfel de bucle se poate implementa diagrame ce interacţionează intre ele. Un alt aspect pozitiv al faptului ca intrările unor diagrame pot fi ieşirile unor alte diagrame este faptul ca procese foarte complexe pot fi descompuse in părţile componente prin simpla înlănţuire a diagramelor.

2.4.2 Generarea rezultatului

Odată ce ordinea de execuţie a fost calculată, procesul de calcul al rezultatului este destul de simplu:

Algoritmul 2: Execuţia unei diagrame FBD

Data: Lista nodurilor ordonate topologic

Result: Punctul de date ce trebuie adaugat pe canalul de iesire a diagramei

```
1  rezultate ← Relaţie cheie-valoare intre nod si rezultatul execuţiei lui;
2  L ← Lista ce conţine nodurile sortate ;
3  foreach nod m din L do
4      intrari ← Lista goala de intrări pentru nodul m ;
5      foreach arc de la n către m do
6          if In rezultate exista rezultatul pentru blocul n then
7              Adaugă in 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 in rezultate valoarea calculata;
13     else
14         else if m este un canal de date then
15             Adaugă in rezultate ultima valoare de pe canal;
16 return Ultima valoare din rezultate
```

Capitolul 3

Interfaţa aplicaţiei

Capitolul 4

Implementarea Alegria

4.1 Interfaţa Web

Alegria a fost implementata cu ajutorul platformei **Spring Boot**.[1] Platforma a fost aleasa pentru stabilitatea ei excepţională, fiind bazata pe Spring Framework care sta la baza unora din cele mai mari aplicaţii existente,[2] dar si pentru uşurinţa prin care o aplicaţie poate fi compusa din elemente funcţionale, abstractizând peste nivelele de jos a programului, permiţând alocarea timpului pe logica aplicaţiei, si nu pe implementarea platformei pe care aplicaţia sa ruleze. Un alt motiv pentru care platforma Spring a fost aleasa, este faptul ca suporta programarea orientată pe aspecte, si injecţia dependinţelor, permiţând scrierea de cod curat si uşor de testat.

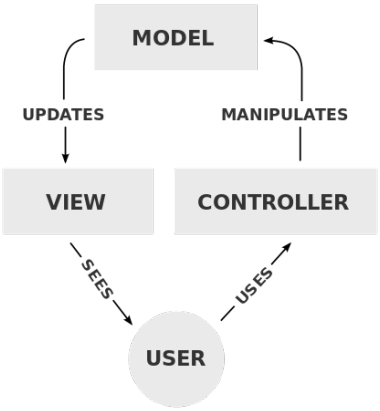

Figura 4.1: Colaborarea intre componentele MVC

Cum aplicaţia este bazata pe arhitectura MVC (model-view-controller) aceasta a fost structurata in trei elemente separate:

- **Interfaţa vizuala**: Realizata in **HTML5**, folosind motorul de templating Thymeleaf pe server si Bootstrap si JQuery in client pentru afişarea paginilor. Aceasta combinaţie permite realizarea de pagini cu un aspect modern, responsiv, care funcţionează atât pe ecranul mare al calculatorului, cat si pe display-ul mic al unui telefon.
- **Modele**: Reprezinta o reprezentare a entităţilor din baza de date in sistemul

[1] *Spring Boot*. URL: `http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/` (visited on 09/02/2015).

[2] *Spring Frameowrk Case Studies*. URL: `http://pivotal.io/resources/1/case-studies` (visited on 09/02/2015).

Alegria.

- **Controller-e**: Realizează legătura intre partea vizuala a aplicaţiei si entităţile din baza de date, asigurând atât metodele care "umplu" template-urile cu date, cat si implementarea interfeţei API care introduce si extrage date.

4.1.1 Securitate

Securitatea este un element de baza, atât pentru accesul la date, cat si pentru accesul la entităţi. Astfel, in implementare s-a folosit framework-ul **Spring Security** care usureaza management-ul securităţii, fiind puternic integrat si cu restul platformei Spring.

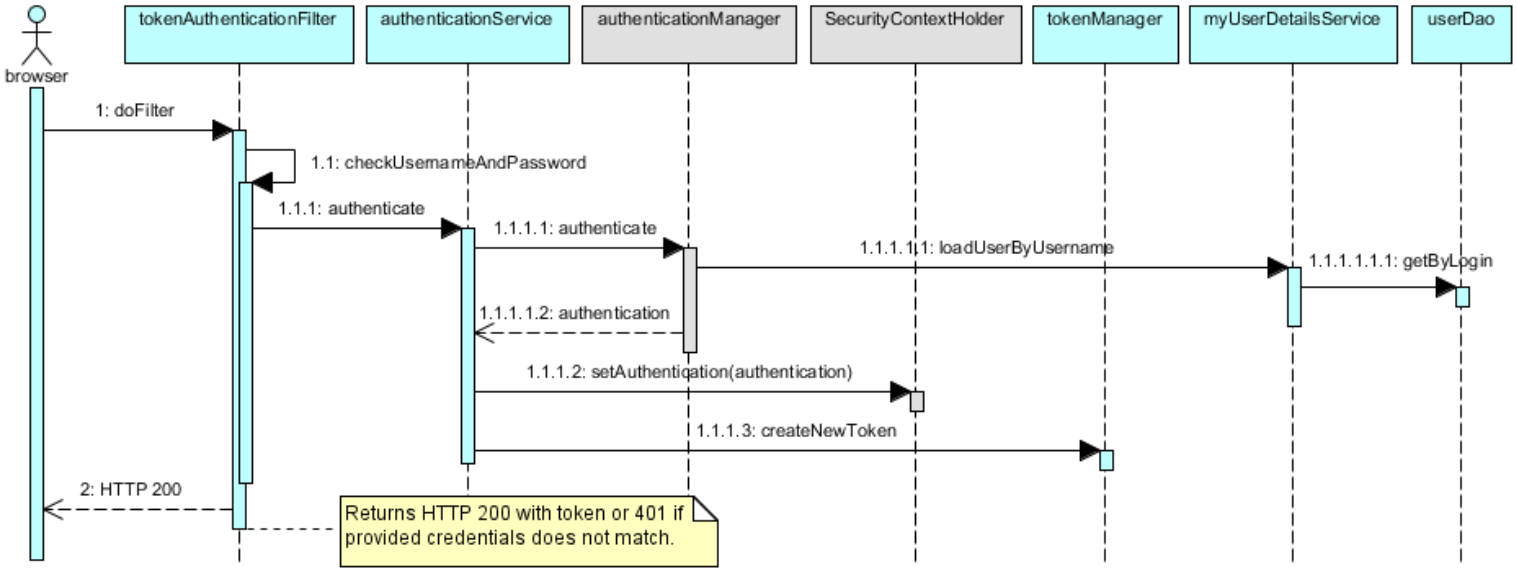

Figura 4.2: Procesul de autentificare in aplicaţie

Pentru autentificare a accesa o resursa protejata, utilizatorul va folosi unul din doua mecanisme:

- Autentificare securizata prin username si parola, aceste detalii fiind stocate in tabela *application_user*, unde parola a fost stocata dupa ce a fost trecuta printr-o funcţie criptografica de hashing. Aceasta metoda de autentificare este folosita pentru autentificarea utilizatorilor in interfaţă de management si monitorizare. Odată ce procesul a reuşit, un token unic va fi generat, iar request-urile următoare vor fi verificate pe baza procesului descris mai jos.

- Autentificare pe baza de token, folosita pentru securizarea API-ului, dar si in cazul in care un user s-a autentificat deja cu username si parola. Fiecare request trebuie sa aibă un token, fie într-un cookie, fie ca parametru in url.

Tot in scopuri de securitate, fiecare entitate care poate fi modificata menţine un istoric al tuturor modificărilor, împreuna cu utilizatorul cu care le-a efectuat, iar, pentru o dezvoltare ulterioara, accesul unui utilizator poate fi limitat doar la obiectele care au aceleaşi tag-uri ca si utilizatorul.

4.1.2 Management

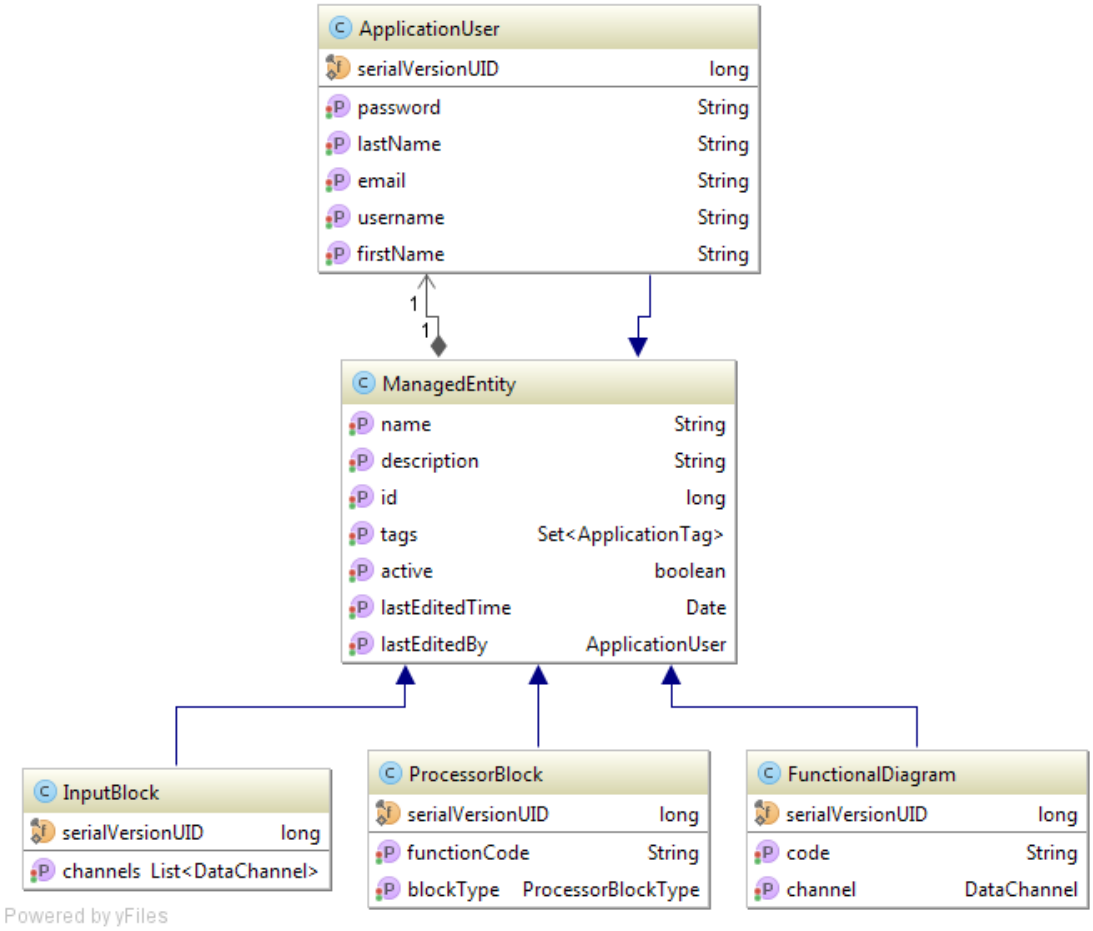

Figura 4.3: Entităţile care sunt administrate de către utilizator si implementeaza `ManagedEntity`

Toate entităţile care implementeaza `ManagedEntity` permit apoi operaţii de adăugare, modificare si ştergere. Acest proces de administrare vizuala foloseşte următoarele resurse, iar un exemplu pentru management-ul :

- Un **repository**, care extinde `JpaRepository` din framework-ul Spring Data. Acesta asigura operaţii de căutare, creare, citire, modificare si ştergere a entităţilor. Un avantaj al folosirii acestui repository, care implementeaza paradigma Data Acces

Object (DAO) este ca interacţiunea cu baza de date se face într-un mod consistent si sigur, incompatibilităţile de tip fiind detectate la compilare, si nu la rulare. In Alegria, toate repository-urile folosite, împreuna cu implementările lor se afla in package-ul `ro.pub.acse.sapd.repository`.

- O **vizualizare**, template Thyeleaf, care într-o singura pagina HTML expune catre utilizator toate operaţiile suportate de repository. Aceasta pagina este una dinamica, ce foloseşte dialoguri modale încărcate prin AJAX pentru a edita entităţi, fără a fi necesar ca utilizatorul sa fie redirecţionat către o alta pagina. Entităţile sunt afişate sub forma tabelara, dinamica, care permite sortarea si filtrarea după diverse condiţii. Dialogul modal de editare este specific entităţii care este modificate. Vizualizările pentru întreaga lista se afla in `resources\templates\management` iar conţinutul dialogului modal se afla in subdirectorul `fragments`.

- Un **controller**, clasa cu adnotarea `@Controller`, care leagă repository-ul de vizualizare, dar si specifica către Spring care sunt endpoint-urile (căile pe care acest controller le tratează) prin adnotarea `@RequestMapping`. Controllere-le pentru managementul entităţilor se afla in `ro.pub.acse.sapd.controller.web.management`. Un exemplu de metode care sunt tratate într-un asemenea controller se găsesc in figura 4.4.

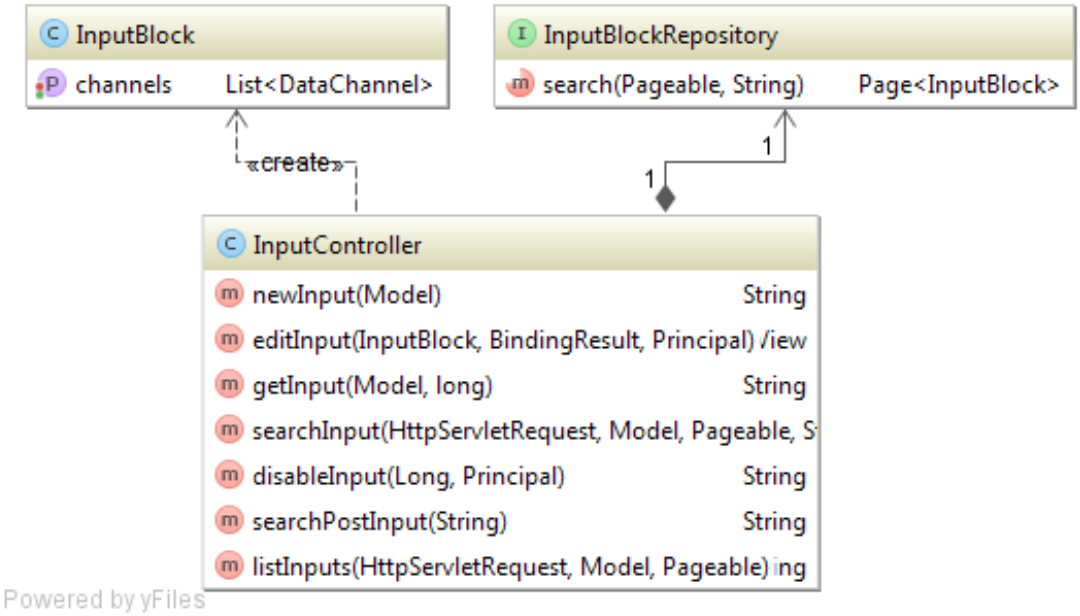

Figura 4.4: Interacţiunea dintre repository, controller si entitate

4.1.2.1 Managementul blocurilor de intrare

Pe lângă elementele descrise mai sus, la management-ul blocurilor de intrare trebuie ca utilizatorul sa poată vizualizeze si modifica lista de canale a unui bloc. In vederea implementării acestei particularităţi, in dialogul modal pentru adăugare si modificare a fost realizat un formular dinamic cu cate o line pentru fiecare canal de date. Pentru blocurile de intrare care au deja un canale ataşate, acest formular este generat de către server, in template-ul thymeleaf `input.html`, iar, dinamismul formularului este implementata cu ajutorul unor funcţii JavaScript care manipulează structura documentului.

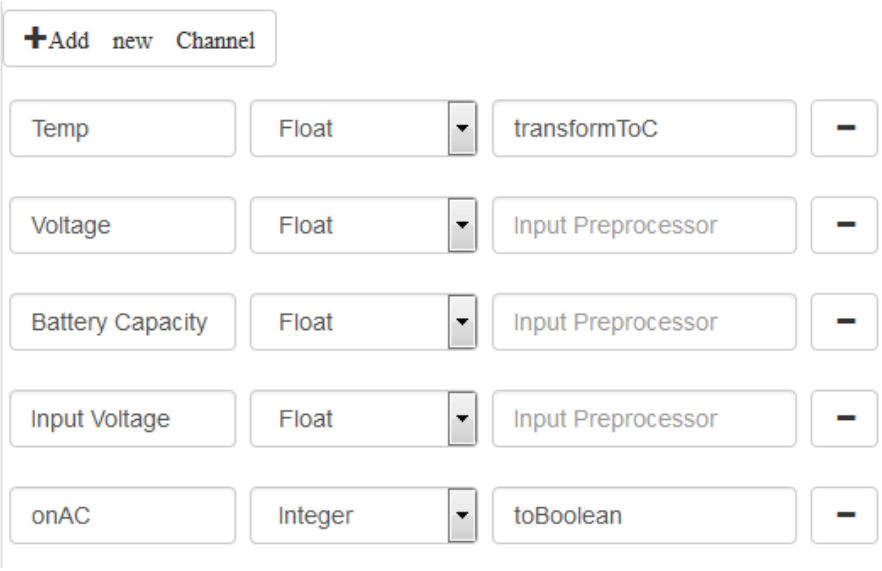

Figura 4.5: Formular HTML dinamic pentru editarea canalelor unui bloc de intrare

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

4.1.2.2 Managementul blocurilor de procesare

O particularitate a editării blocurilor de procesare este folosirea librăriei JavaScript **CodeMirror**[4] pentru afişarea codului, cuvintele cheie ale limbajului dat de tipul blocului fiind evidenţiate, acest lucru făcând dezvoltarea codului mult mai facila. Un alt avantaj al acestei librari este posibilitatea găsirii erorilor de sintaxa mult mai rapid, fără a testa

[3] *Bootstrap 3 Typeahead*. URL: `https://github.com/bassjobsen/Bootstrap-3-Typeahead` (visited on 09/02/2015).

[4] *CodeMirror*. URL: `https://codemirror.net/index.html` (visited on 09/02/2015).

blocul. Aceasta funcţionalitate este implementata cu ajutorul unei funcţii care se executa de fiecare dată când valoarea selectată in input-ul "Block Type" se modifică.

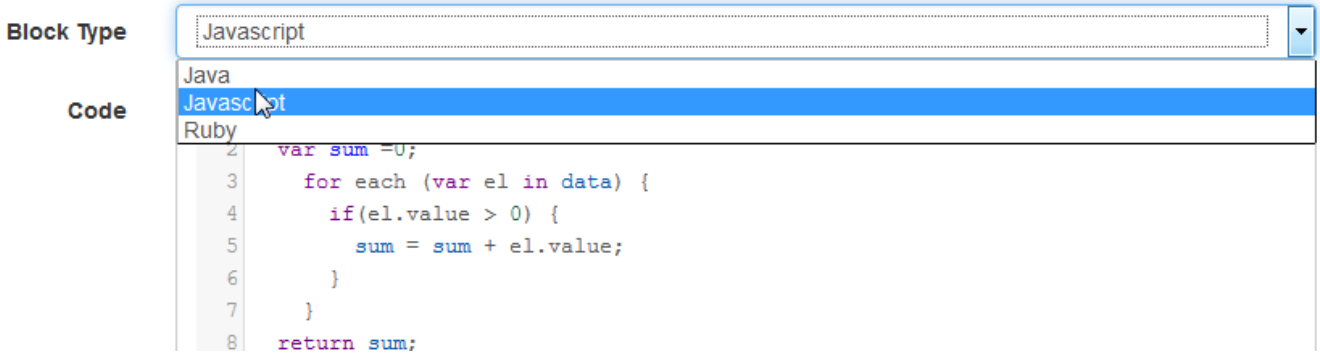

Figura 4.6: Modificarea limbajului din editorul CodeMirror in funcţie de tipul blocului

4.1.2.3 Managementul diagramelor

Pentru implementarea funcţiilor de design vizual al diagramelor funcţionale s-a ales librăria JavaScript **GoJS**.[5] Aceasta permite implementarea a diagrame interactive, fiind compatibila atât cu toate browsere-le cat si cu dispozitivele mobile moderne. Deoarece asigura suport pentru drag-and-drop, copiere si lipire, undo si redo dar si multe alte funcţionalităţi, librăria reprezinta un punct foarte bun de plecare pentru implementarea diagramelor. Un alt avantaj al acestei librarii este ca suporta adăugarea de condiţii asupra diagramei chiar la construcţia acesteia. Din acest motiv, librăria a fost configurata sa nu permită decât legături de la o ieşire la o intrare, si canalul de ieşire sa nu poată fi legat decât la o singura ieşire.

In vederea realizării acestor configurări, fişierul JavaScript `diagram.js`. Aici sunt configurate tipurile de blocuri:

- **Canale de intrare**: acestea sunt configurate sa nu poată avea intrări, si sa doar o singura ieşire. Atunci când un canal de intrare este selectat, utilizatorul poate sa ii atribuie un nume si sa selecteze care este canalul referfenţiat de acel bloc. Aceasta selecţie se face cu ajutorul unui input cu auto completare.

- **Blocuri de procesare**: deoarece numărul de intrări este variabil, funcţia `addPort` a fost implementata. Aceasta este apelata atunci când utilizatorul da click pe opţiunea "Add input" din meniul contextual al blocului. Pentru operaţiunea de ştergere a portului, funcţia `removePort(port)` a fost implementata. Ordinea acestor porturi determina ordinea elementelor din lista cu care este apelat blocul de procesare.

[5] *GoJS Interactive Diagrams for JavaScript and HTML*. URL: `http://gojs.net/latest/index.html` (visited on 09/03/2015).

Atunci când blocul este selectat, utilizatorul poate sa ii atribuie un nume si sa selecteze care este blocul de procesare referfeniţat de acel bloc. Aceasta selecţie se face cu ajutorul unui input cu auto completare. Astfel, acelaşi bloc de procesare poate fi folosit de mai multe ori in cadrul aceleiaşi diagrame. Pentru uşurinţă dezvoltării, linkuri către detaliile blocului de procesare sunt disponibile chiar in interiorul diagramei.

- **Final**: bloc ce semnifica canalul de ieşire a diagramei. Configurat cu un singur port de intrare, el nu poate fi legat decât la o singura ieşire.
- **Comentariu**: nu se poate lega in diagrama si nu ia parte la execuţia acesteia.

Toate aceste blocuri sunt adăugate si într-o paleta pentru adăugare uşoară in diagrama. Deoarece blocul "Final" nu poate avea decât o singura instanta, acesta nu este disponibil in paleta.

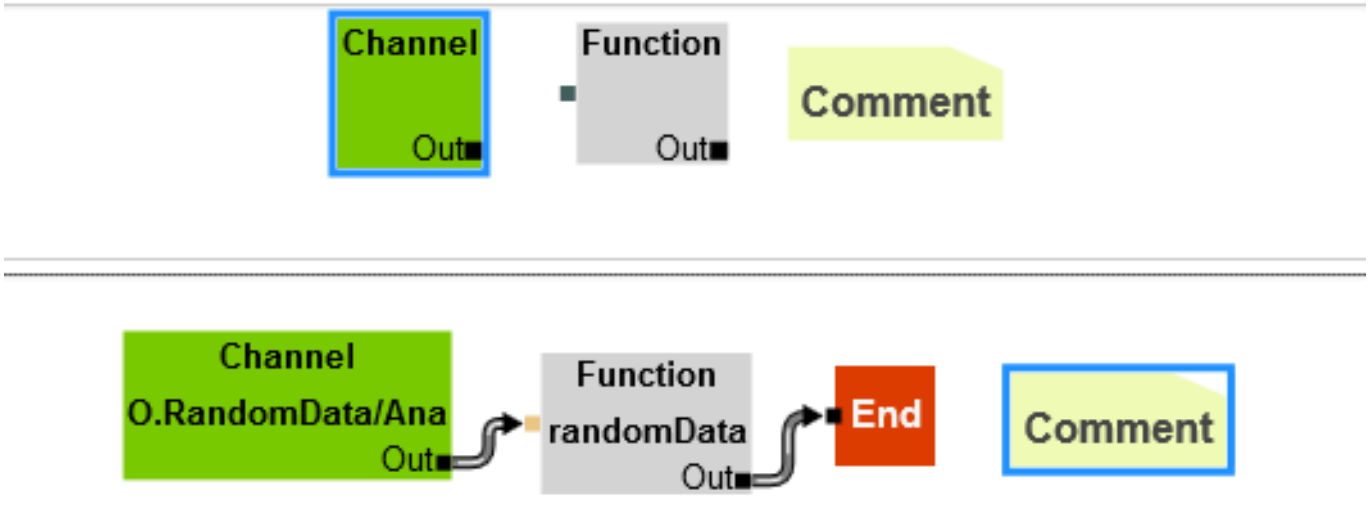

Figura 4.7: Paleta de blocuri si exemplu de instante ale acestora

Când utilizatorul salvează o diagrama, un model JSON a acesteia este generat, model ce este apoi salvat in baza de date. In acest model sunt salvate atât proprietăţile diagramei, urmate de o lista nodurilor, si nu in ultimul rând legăturile dintre noduri.

4.1.2.4 Administrare tag-uri

După cum s-a discutat in secţiunea despre securitate, fiecare entitate care este administrata de utilizator poate avea mai multe taguri. Acestea reprezinta o lista de cuvinte care specifica un concept, spre exemplu, toate entităţile care sunt folosite într-o diagrama pot avea acelaşi tag. Tagurile permit astfel gruparea entităţilor, fiind folosite in căutare.

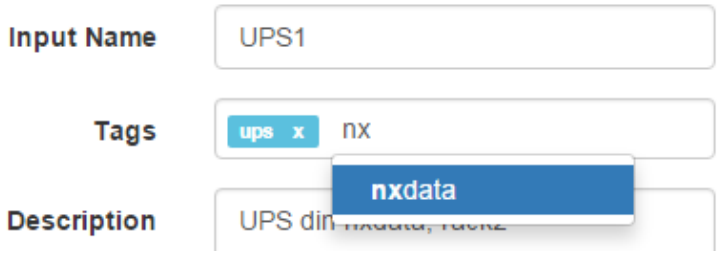

Figura 4.8: Editarea tag-urilor unui bloc de intrare

Pentru implementare, pe partea de server au fost creata clasa `ApplicationTag`, cu doua câmpuri: nume, si Id. Toate instantele acestei clase sunt salvate in baza de date in tabela `application_tag` iar endpoint-ul `/tags/getTags` care întoarce toate tag-urile din acea tabela. Acest serviciu API este folosit de librăria JavaScript Bootstrap Tags Input,[6] care, împreună cu librăria de autocompletare[7] permite utilizatorului sa selecteze taguri deja existente. Atunci când utilizatorul introduce totuşi un tag care nu exista deja in baza de date, acesta este creat automat, fiind setat direct pe entitatea modificata. Fiecare `ManagedEntity` are un set de taguri, permiţând salvarea tagurilor in baza de date, într-un tabel separat, care implementeaza relaţia de Multi-la-Multi dintre entitate si `ApplicationTag`.

[6] *Bootstrap Tags Input*. URL: `http://timschlechter.github.io/bootstrap-tagsinput/examples/` (visited on 09/04/2015).

[7] *Bootstrap 3 Typeahead*.

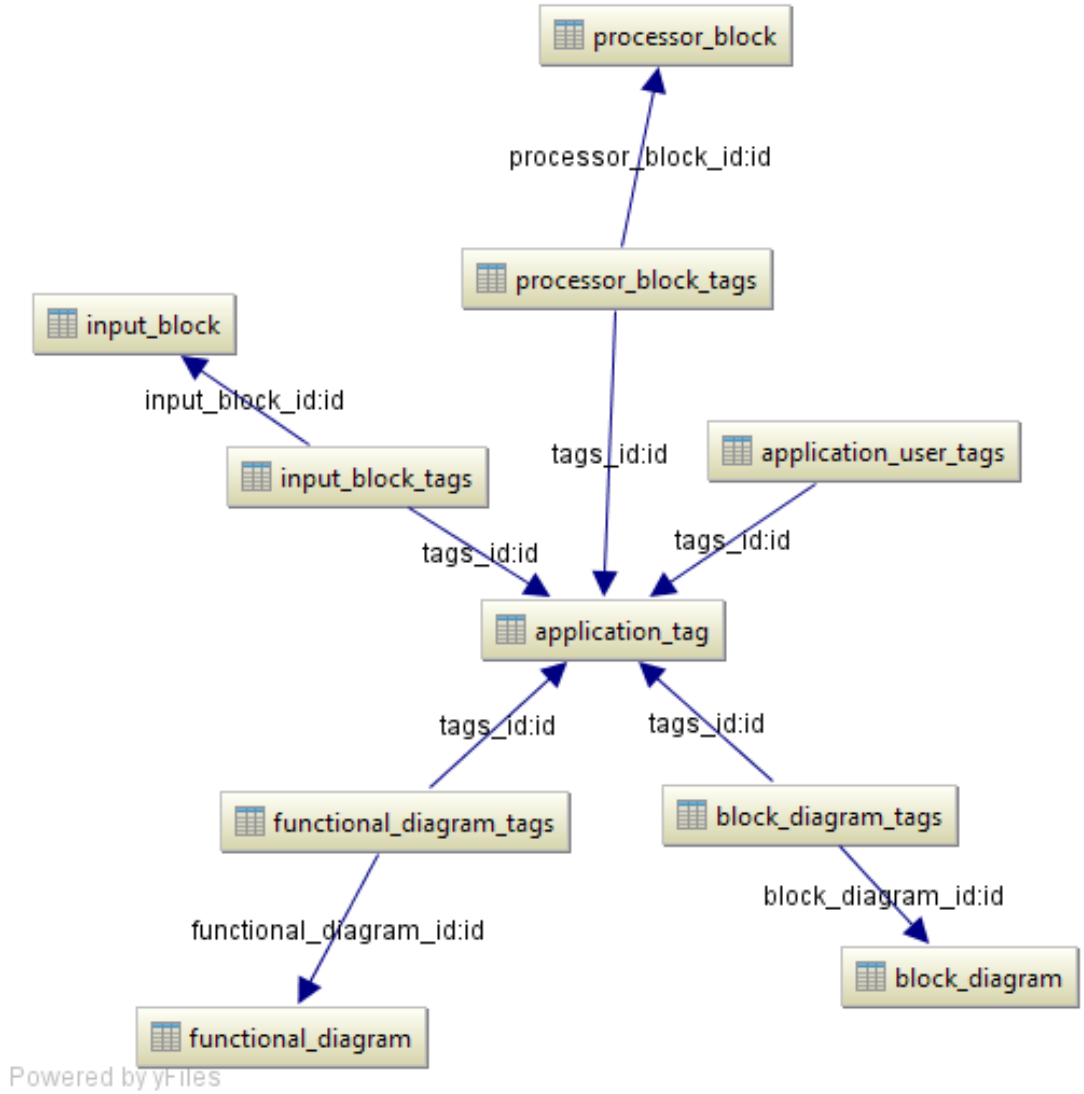

Figura 4.9: Relaţiile dintre tabela `tags` si celelalte entităţi

4.1.3 Monitorizare

O chestiune de mare importanta este vizualizarea datelor ce au intrat sau au fost procesate de aplicaţie. In acest scop, a fost implementata vizualizarea datelor in timp real. Pentru extragerea informaţiilor de pe un canal se folosesc endpoint-urile din API de la `/api/fetch/channelId` si , care întorc punctele de pe un canal dintr-un anumit interval de timp, într-un format JSON.

Deoarece datele procesate de aplicaţie sunt de mai multe tipuri, apare totuşi problema modului lor de afişare. In funcţie de datele întoarse prin API, se selectează unul din doua moduri de afişare; Pentru datele numerice a fost aleasa o reprezentare cu ajutorul unui

grafic, folosind librăria dygraphs,[8] iar pentru datele care nu pot fi reprezentate numeric, acestea sunt afişate tabelar.

Pentru ambele moduri de reprezentare, utilizatorul are opţiunea sa obţină doar datele dintr-o anumita perioada de timp, sau chiar ultimele înregistrări pe acel canal. Acest mod de selecţie este ilustrat in figura 4.10

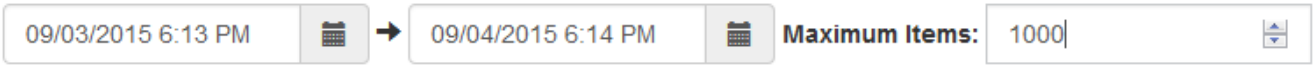

Figura 4.10: Filtrarea datelor obţinute

4.2 API-ul aplicaţiei

Pentru a permite interfaţarea aplicaţiei cu alte servicii, dar si pentru a facilita dezvoltarea unor interfeţe web dinamice, aplicaţia dispune de un api REST. Acesta a fost configurat sa folosească date in format JSON.

In următoarea lista sunt descrise toate endpoint-urile API-ului, împreună cu detalii despre folosirea acestora. In afara de câteva excepţii menţionate, toate aceste servicii folosesc metoda HTTP GET.

- `/blocks/getBlocks`: întoarce toate blocurile de procesare din baza de date.
- `/blocks/getChannels`: întoarce toate canalele declarate in baza de date. Pentru fiecare canal returnat, se specifica de la ce bloc de intrare face parte, sau dacă este ieşirea unui diagrame funcţionale.
- `/tags/getTags`: întoarce toate tagurile existente in aplicaţie.
- `/api/put/{inputId}/{channelId}/{data}`: Adaugă punctul `data` pe canalul specificat prin `channelId`. Datele primite sunt in format `String`, care respecta standardul specificat in RFC3986.[9] Foloseşte metoda PUT.
- `/api/put/openTDSB`: Adaugă date in formatul openTDSB.[10] In corpul requestului trebuie sa se afle un JSON care respecta standardul impus. Foloseşte metoda PUT.
- `/api/fetch/{channelId}`: Întoarce date de pe canalul `channelId`. Următorii parametrii pot fi folosiţi pentru a extrage doar anumite date:

[8] *Dygraphs is a fast, flexible open source JavaScript charting library*. URL: `http://dygraphs.com/` (visited on 09/03/2015).

[9] *Uniform Resource Identifier (URI): Generic Syntax*. RFC 4627. RFC Editor, Jan. 2005, pp. 1–61. URL: `https://tools.ietf.org/html/rfc3986`.

[10] *The Scalable Time Series Database*.

  - `from`: Data de început a istoricului, in format ISO 8601.
  - `to`: Data de final a istoricului, in format ISO 8601.
  - `maxItems`: Numărul maxim de înregistrări ce sunt returnate. Acest parametru are 500 ca valoare implicita.

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

Pe lângă aceste metode, aplicaţia mai pune la dispoziţie si API-ul generat automat cu ajutorul Spring Data. Acesta permite adăugarea, modificarea, ştergerea si cutarea entităţilor folosite in aplicaţie programatic.

4.3 Executarea unui bloc de procesare

Blocurile de procesare este folosit Spring Bean-ul `BlockExecutor`. Acesta este responsabil de iniţializarea tuturor implementărilor pentru interfaţa `GenericBlockExecutor`. Pentru fiecare implementare exista un câmp asociat in enumeraţia `ProcessorBlockType`, câmp care este folosit ca proprietate in entitatea `ProcessorBlock`. Acest Bean poate fi apoi injectat folosind adnotarea `@Autowired` in toate clasele care doresc sa execute blocuri.

Interfaţa `GenericBlockExecutor` are o singura metoda `processData` cu doi parametrii: unul de tip String, in care este codul funcţiei ce trebuie executat, si unul de tip `List<DataPoint>` care conţine toate punctele ce pot fi folosite in blocul respectiv. Clasa `DataPoint` au doua câmpuri: valoare si instanta de timp. Următoarele implementări a interfeţei `GenericBlockExecutor` exista in sistem:

- `JavaBlockExecutor`: Acest executor reprezinta un caz special deoarece foloseşte clase locale, ce sa afla deja in classpath-ul JVM-ului pe care rulează aplicaţia. Aceste blocuri pot fi folosite pentru a extinde aplicaţia cu cod de înaltă performaţă, acţionând ca un mecanism de extensii ale aplicatei, permiţând interacţiunea cu alte sisteme. Aceste blocuri pot sa reprezinte doar o interfaţa pentru apelul unor librarii externe, făcând posibila, spre exemplu, implementarea unui bloc care sa execute scripturi MATLAB. Acest bloc primeşte ca parametru doar numele canonic al clasei, iar, la execuţie încarcă clasa referentiata folosind funcţii din pachetul `java.lang.reflect`. Dacă clasa nu este găsită, sau o eroare are loc la execuţia clasei, o excepţie de tipul `BlockExecutionException` este generată.

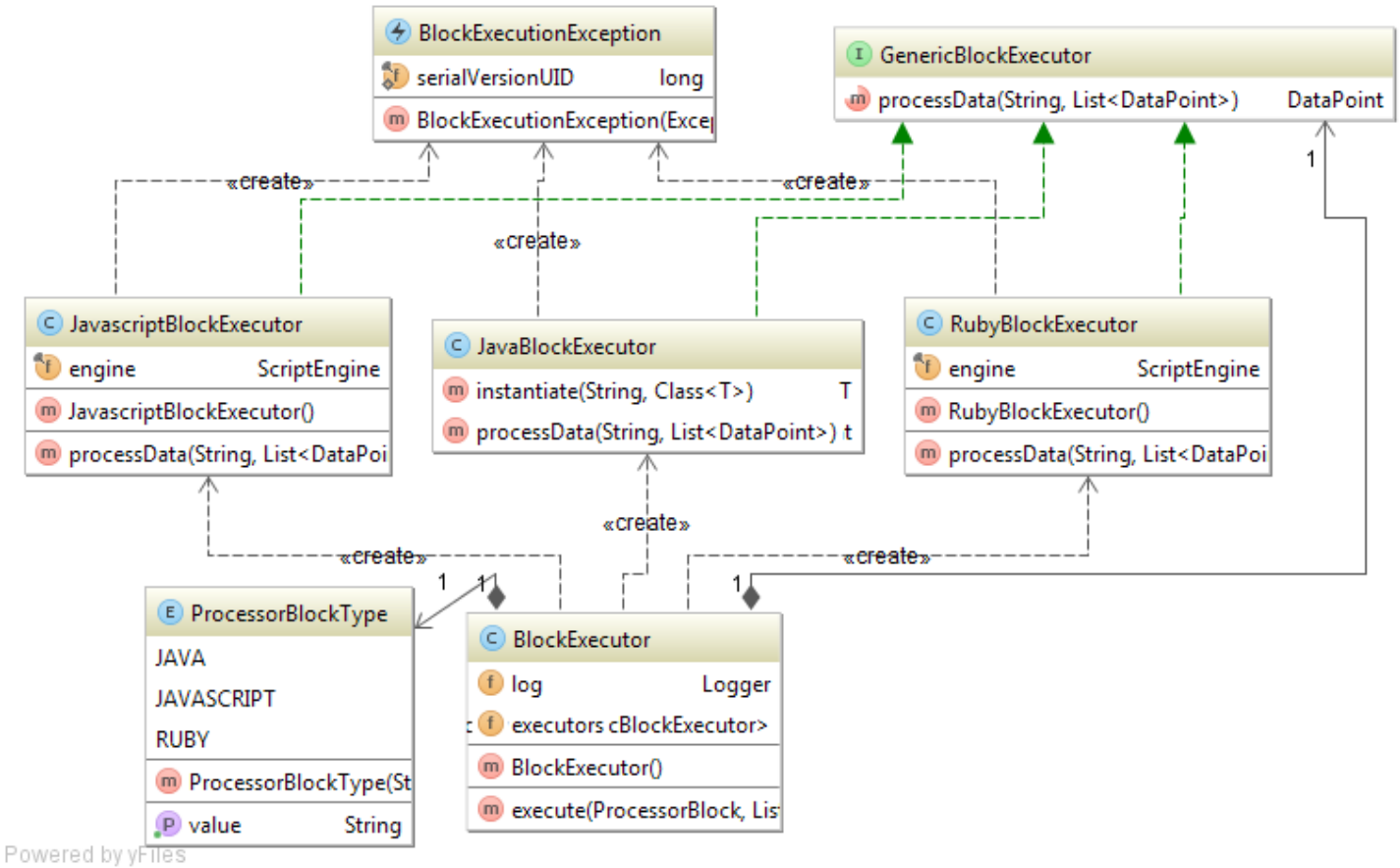

Figura 4.11: Clasele ce asigura executarea unei diagrame

- `JavascriptBlockExecutor`: Executor ce poate rula cod JavaScript pe server. Aceste blocuri trebuie sa conţină o funcţie `parseInput` care ia ca parametru o lista de `DataPoint`. Pentru execuţia blocurilor de acest tip, standardul JSR 223[11] vine in ajutor prin abstractizarea funcţionalităţii interne necesare execuţiei de cod scris într-un limbaj dinamic, direct in maşina virtuala Java. In versiunea 8 a JVM-ului aceste funcţii sunt executate folosind runtime-ul de înaltă performanta Nashorn, care este accelerat de modificări recente, introduse in JSR 292,[12] ridicând performanta codului JavaScript la un nivela apropiat de funcţiile scrise in Java. Pentru versiunile precedente de JVM este folosit Mozilla Rhino. Dacă funcţia `parseInput` nu este găsită, sau o eroare are loc la execuţia script-ului, o excepţie de tipul `BlockExecutionException` este generată. Scriptul poate returna direct instante ale interfeţei `DataPoint`, sau alte obiecte care sunt apoi reprezentate ca un

[11] *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`.

[12] *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`.

`StringDataPoint`.

- `RubyBlockExecutor`: Executor ce rulează cod Ruby pe server. Din punct de vedere al implementării este similar cu `JavascriptBlockExecutor`, însă foloseşte librăria JRuby[13] .

4.4 Executarea unei diagrame funcţionale

Pentru execuţia diagramelor este folosit Spring Bean-ul `DiagramExecutor`. Acesta este responsabil atât de parsarea unei diagrame cat si de execuţia si extragerea rezultatelor. Acest Bean poate fi apoi injectat folosind adnotarea `@Autowired` in toate clasele care doresc sa execute diagrame.

Parsarea este realizata cu ajutorul interfeţei `DiagramParser`, aceasta permiţând transformarea diagramei într-un graf reprezentat printr-o lista de adiacenţă. O implementarea a acestei interfeţe este clasa `GoJsDiagramParser` care parsează JSON-ul general de librăria GoJS. Aceasta parsare se face cu ajutorul librăriei Jackson, folosind schema din figura figura 4.13. Dacă parsarea nu este realizata cu succes, atunci o excepţie de tipul `DiagramParseException` este aruncată. Dacă se doreşte înlocuirea librăriei de front-end pentru realizarea diagramelor, sistemul poate fi adaptat doar prin scrierea unei noi clase care implementează `DiagramParser`.

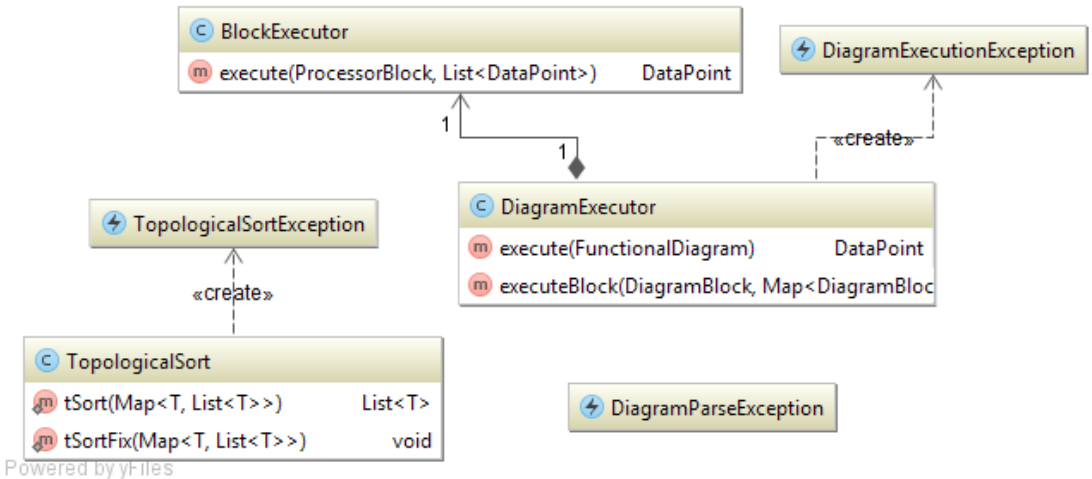

Figura 4.12: Clasele implicate in executarea unei diagrame

Odată ce parsarea a fost efectuata, execuţia blocului poate continua: folosind clasa `TopologicalSort`, o implementare a alg. 1 cu tipuri generice si expresii lambda, este

[13] *The Ruby Programming Language on the JVM*. URL: `http://jruby.org/` (visited on 09/04/2015).

executata. Apoi, alg. 2 este executat, si, cu ajutorul bean-ului `BlockExecutor` blocurile din diagrama sunt executate. Rezultatul final al metodei `execute` este un `DataPoint` care va fi salvat in baza de date, pe canalul de ieşire a diagramei.

O alta chestiune importantă este determinarea momentelor când o diagrama trebuie executată. In acest scop, pe fiecare canal are un set de diagrame care trebuie lansate atunci când se primesc informaţii noi. Aceasta lista este actualizată de fiecare data când diagrama se modifica prin interfaţa web. Metodele din `DiagramParser` folosite pentru a obţine lista de canale utilizate si pe fiecare dintre acestea, pe proprietatea `Set<FunctionalDiagram> subscribedDiagrams` este adăugată acea diagrama. Pentru a nu se pierde consitenţa datelor, cat timp diagrama este modificata, ea nu se mai poate lansa in execuţie.

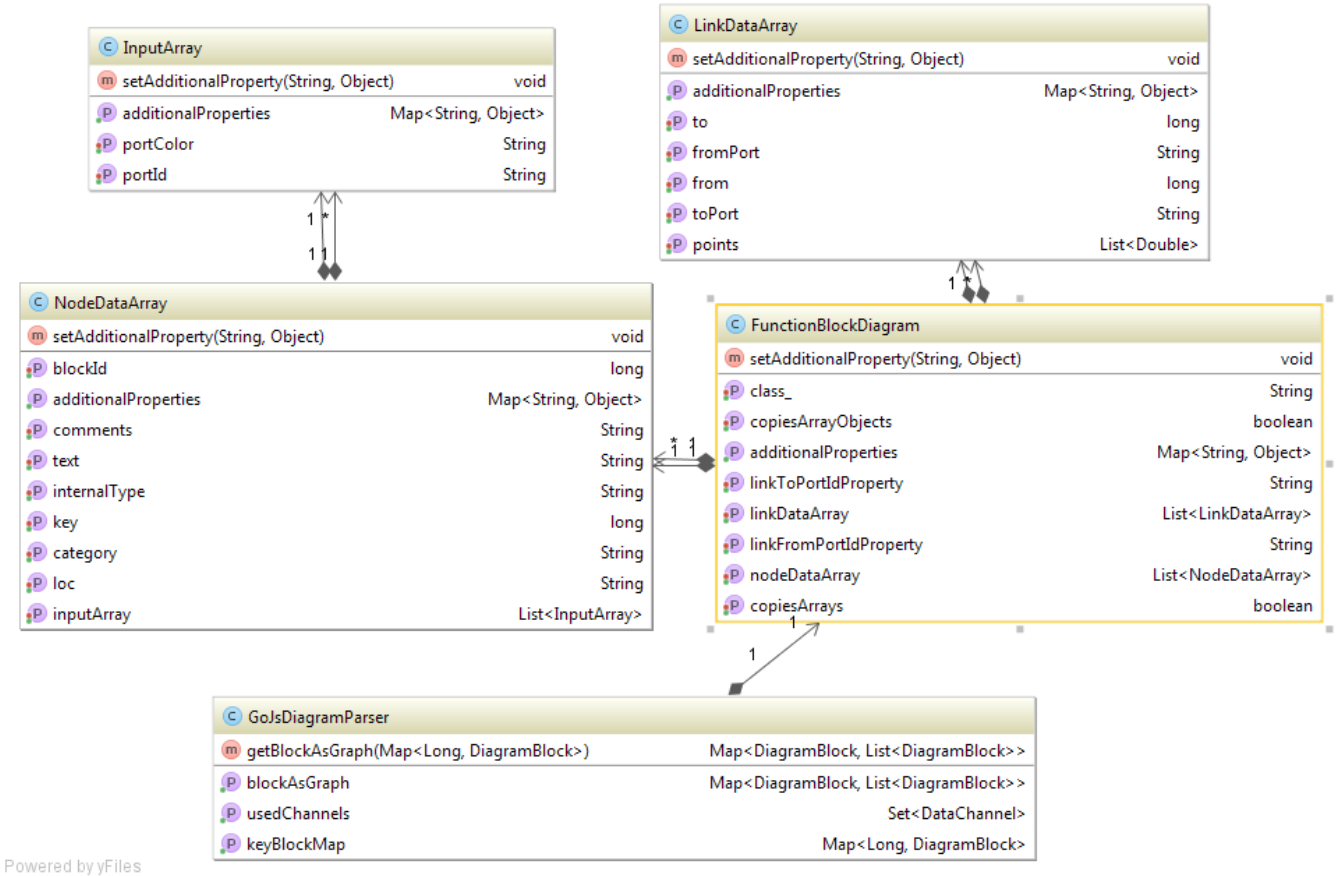

Figura 4.13: Modelul de date pentru diagramele GoJS

4.5 Serviciul de date

Serviciul de date, implementat prin Spring Bean-ul `DataService` permite abstractizarea modului in care `DataPoint`-urile sunt scrise in baza de date. In aceasta implementare, datele sunt stocate in aceeaşi instanta de PostgreSQL ca si entităţile.

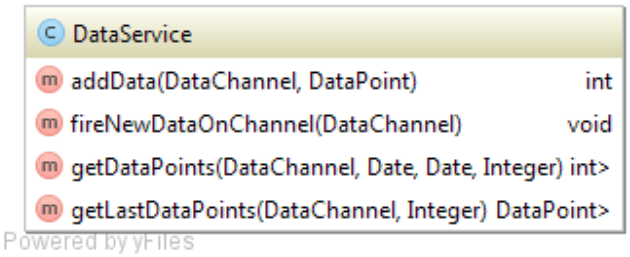

Figura 4.14: Interfaţa aplicaţiei cu baza de date pentru informaţii

Aceasta clasa asigura si execuţia diagramelor din lista de `subscribedDiagrams` a canalului ce primeşte date. Aceasta execuţie se executa in mod **asincron**, pentru a nu bloca firul de execuţie care primeşte date. Asincornicitatea este implementata printr-un pool de thread-uri care primesc task-uri de execuţie, thread-ul apelant nefiind interesat de rezultatul execuţiei. Astfel, putem considera ca implementarea pe mai multe fire de execuţie este de tipul fire-and-forget.

4.6 Baza de date

Capitolul 5

Studiu de caz: Smart Home

Capitolul 6

Concluzii

Bibliografie

Bootstrap 3 Typeahead. URL: https://github.com/bassjobsen/Bootstrap-3-Typeahead (visited on 09/02/2015).

Bootstrap Tags Input. URL: http://timschlechter.github.io/bootstrap-tagsinput/examples/ (visited on 09/04/2015).

CodeMirror. URL: https://codemirror.net/index.html (visited on 09/02/2015).

DAGs and Topological Ordering. URL: http://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf (visited on 08/20/2015).

Dygraphs is a fast, flexible open source JavaScript charting library. URL: http://dygraphs.com/ (visited on 09/03/2015).

Function Blocks. URL: http://www.functionblocks.org/index.html (visited on 08/20/2015).

GE FANUC Function Block Diagram Lab. URL: http://geplc.com/downloads/Labs/GFS-384%20M03%20Function%20Block%20Diagram.pdf (visited on 08/20/2015).

GoJS Interactive Diagrams for JavaScript and HTML. URL: http://gojs.net/latest/index.html (visited on 09/03/2015).

IEEE Standard for Floating-Point Arithmetic. Aug. 2008, pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

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.

Kahn, A. B. *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.

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).

Pearce, David J. 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.

Programmable controllers - Part 3: Programming languages. IEC. TC 65/SC 65B - Measurement and control devices, July 2013. URL: http://www.dee.ufrj.br/controle_automatico/cursos/IEC61131-3_Programming_Industrial_Automation_Systems.pdf.

Scripting for the Java^{TM} Platform. JCP 223. Sun Microsystems, Inc., Dec. 2006, pp. 1–140. URL: https://www.jcp.org/en/jsr/detail?id=223.

Spring Boot. URL: http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/ (visited on 09/02/2015).

Spring Frameowrk Case Studies. URL: http://pivotal.io/resources/1/case-studies (visited on 09/02/2015).

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.

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.

The Ruby Programming Language on the JVM. URL: http://jruby.org/ (visited on 09/04/2015).

The Scalable Time Series Database. URL: http://opentsdb.net/index.html (visited on 08/20/2015).

TM241 Programming Manual. URL: http://www.kongzhi.net/files/download.php?id=8362 (visited on 08/20/2015).

Uniform Resource Identifier (URI): Generic Syntax. RFC 4627. RFC Editor, Jan. 2005, pp. 1–61. URL: https://tools.ietf.org/html/rfc3986.