Sisteme Distribuite - Laborator 9

Microservicii cu Spring Cloud

Spring Cloud Data Flow

Data Flow reprezintă o componentă Spring Cloud utilizată pentru procesarea bazată pe microservicii a datelor în flux (eng. *stream*). Data Flow furnizează utilitare pentru crearea de topologii complexe de emitere în flux (eng. *streaming*) şi *pipeline*-uri de date.

***Pipeline*-urile de date** constau în aplicaţii Spring Boot construite utilizând framework-uri diverse pentru microservicii, precum **Spring Cloud Stream** sau **Spring Cloud Task**.

Atenţie! Cei care lucrează de pe stațiile din laborator, săriți direct la Secțiunea “Server RabbitMQ”.

Instalarea unui server local Data Flow

Pentru a putea utiliza Data Flow, trebuie descărcate şi instalate câteva componente cu care se creează pipeline-uri de date. În continuare, veţi executa comenzi din terminal şi veţi descărca artefacte JAR necesare pentru server-ul Data Flow. Se recomandă crearea în prealabil a unui folder separat, numit **DataFlow**, spre exemplu, pentru componentele descărcate, în care se vor executa comenzile „**`wget ...`**”:

```
mkdir DataFlow
cd DataFlow
```

Data Flow Server

Se preia artefactul gata împachetat din Maven Central, ce încapsulează server-ul pregătit pentru utilizare:

```
wget https://repo1.maven.org/maven2/org/springframework/cloud/spring-cloud-dataflow-server/2.5.4.RELEASE/spring-cloud-dataflow-server-2.5.4.RELEASE.jar
```

URL-ul către artefactul din Maven Central (ultima versiune disponibilă la momentul scrierii laboratorului):
https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dataflow-server/2.5.4.RELEASE

Data Flow Shell

***Shell*-ul Data Flow** expune o interfaţă la linia de comandă cu care utilizatorul poate gestiona setările pipeline-urilor şi fluxurilor de date. Ca şi în cazul componentei anterioare, se descarcă artefactul gata de utilizare, astfel:

```
wget https://repo1.maven.org/maven2/org/springframework/cloud/spring-cloud-dataflow-shell/2.5.4.RELEASE/spring-cloud-dataflow-shell-2.5.4.RELEASE.jar
```

URL-ul către artefactul din Maven Central (ultima versiune disponibilă la momentul scrierii laboratorului):
https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dataflow-shell/2.5.4.RELEASE

Spring Cloud Skipper

Skipper este un utilitar ce facilitează descoperirea şi gestionarea ciclului de viaţă al aplicaţiilor Spring Boot. Se poate folosi ca utilitar de sine stătător, pe maşina locală, sau poate fi utilizat pe o platformă *cloud* pentru a-l integra cu *pipeline*-uri de tip **CI-CD** (*Continuous Integration - Continuous Deployment*).

Descărcaţi această componentă asemănător:

```
wget https://repo1.maven.org/maven2/org/springframework/cloud/spring-cloud-skipper-server/2.4.3.RELEASE/spring-cloud-skipper-server-2.4.3.RELEASE.jar
```

URL-ul către artefactul din Maven Central (ultima versiune disponibilă la momentul scrierii laboratorului):
https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-skipper-server/2.4.3.RELEASE

Server RabbitMQ

Pentru aplicaţiile din laborator, este necesar şi un *broker* de mesaje (eng. *message broker*). Aplicaţiile implicate în *pipeline*-ul de mesaje au nevoie de o entitate de tip *middleware* pentru a comunica. În exemplele din laborator, veţi folosi **RabbitMQ**.

Instalaţi şi porniţi un server local de RabbitMQ conform instrucţiunilor din **laboratorul 5** de Sisteme Distribuite. Dacă îl aveţi deja instalat, verificaţi funcţionarea acestuia navigând la URL-ul http://localhost:15672/.

ALTERNATIVĂ: folosiţi un server containerizat de RabbitMQ, astfel:

```
docker run -d --hostname rabbitmq --name rabbitmq -p 15672:15672 -p 5672:5672 rabbitmq
```

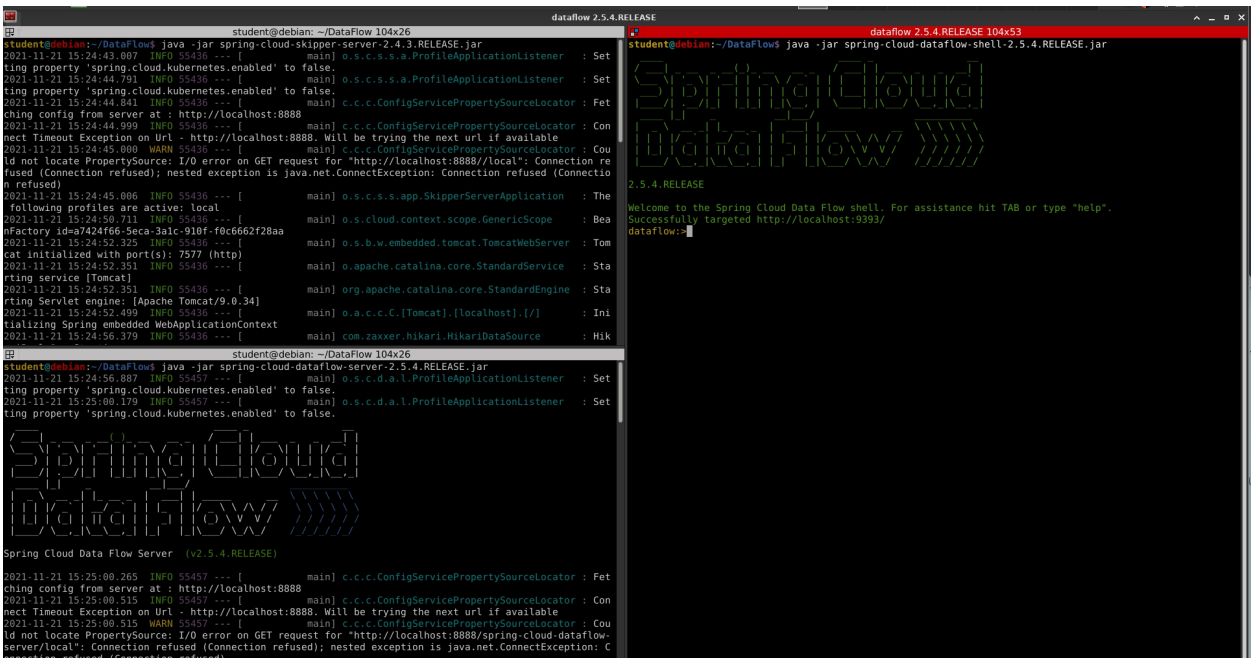
Comanda de mai sus execută un server RabbitMQ folosind Docker, astfel încât nu veţi avea nevoie de instalarea unei instanţe locale.

Pornire server Data Flow

Server-ul local Data Flow se porneşte odată cu celelalte 2 componente menţionate anterior. Executaţi următoarele comenzi din folder-ul unde aţi descărcat componentele Skipper, Data Flow Server şi Data Flow Shell, **fiecare într-un terminal separat**:

```
java -jar spring-cloud-skipper-server-2.4.3.RELEASE.jar
java -jar spring-cloud-dataflow-server-2.5.4.RELEASE.jar
java -jar spring-cloud-dataflow-shell-2.5.4.RELEASE.jar
```

Atenţie: executaţi ultima comandă DOAR după ce server-ul Data Flow şi componenta Skipper au pornit complet! Executaţi primele 2 comenzi mai întâi, aşteptaţi să pornească server-ul, apoi executaţi-o pe ultima, ca *shell*-ul să se conecteze corect la server.



Păstrați cele 3 sesiuni de terminal pornite. Vor fi utilizate în aplicaţiile din laborator.

Accesarea Data Flow Dashboard

Panoul de control al componentei Data Flow se poate accesa la următorul URL: http://localhost:9393/dashboard

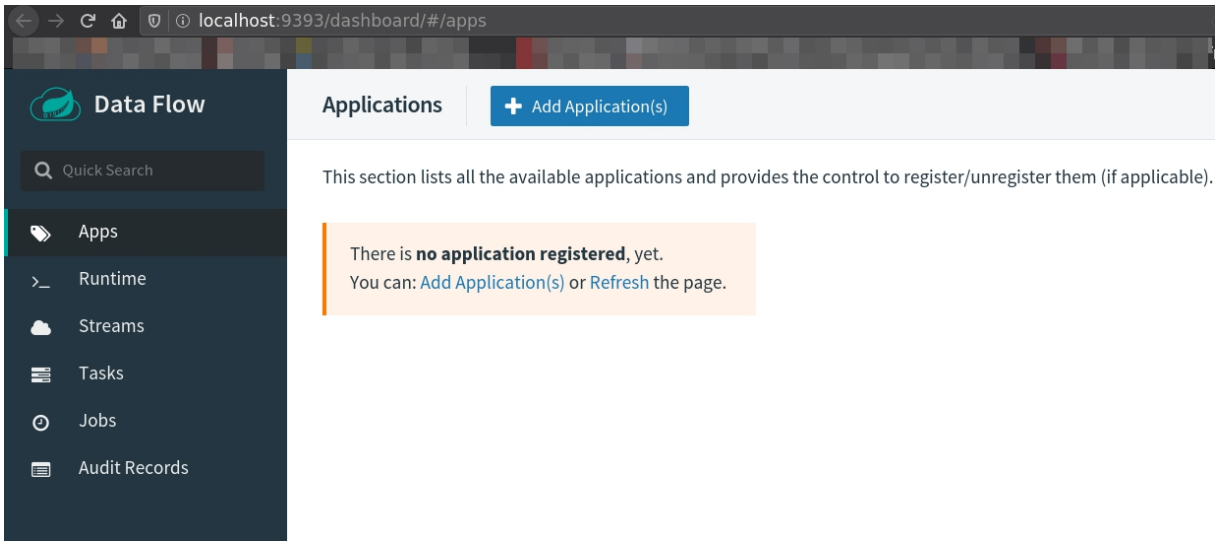


Figura 1 - Panoul de control Spring Data Flow

Veţi avea nevoie de consola de administrare pentru vizualizarea fişierelor log şi verificarea funcţionării corecte a fluxurilor de date create.

Componentele unui pipeline de procesare a datelor în flux (stream processing pipeline)

Un *pipeline* de procesare a datelor în flux este alcătuit, conform specificaţiilor Spring Data Flow, din 3 componente de bază:

- **Sursă** (eng. ***Source***) - reprezintă generatorul de evenimente, sursa de date din pipeline, care produce date ce urmează a fi procesate

• **Procesor** (eng. ***Processor***) - entitate care preia evenimente de la sursă şi le procesează sub o anumită formă

• ***Sink*** - reprezintă destinaţia evenimentelor procesate; această entitate interceptează mesajele de la Procesor.

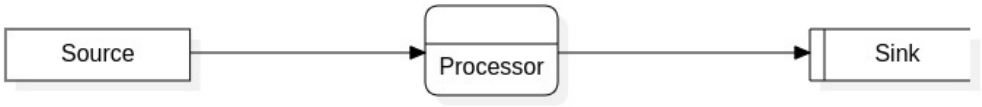


Figura 2 - Diagrama unui flux de date

Exemplul 1 - aplicaţie pipeline simplă cu Spring Cloud Data Flow

Pentru prima aplicaţie, se vor crea 3 microservicii Spring Boot corespunzătoare celor 3 tipuri de entităţi dintr-un pipeline Data Flow: o sursă de date, un procesor de date, şi o entitate *sink*.

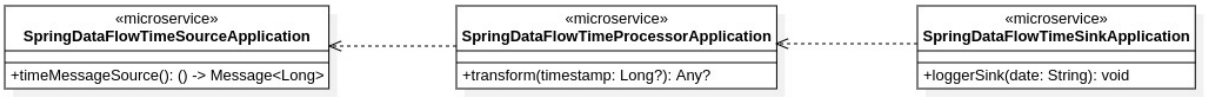


Figura 3 - Diagrama de clase

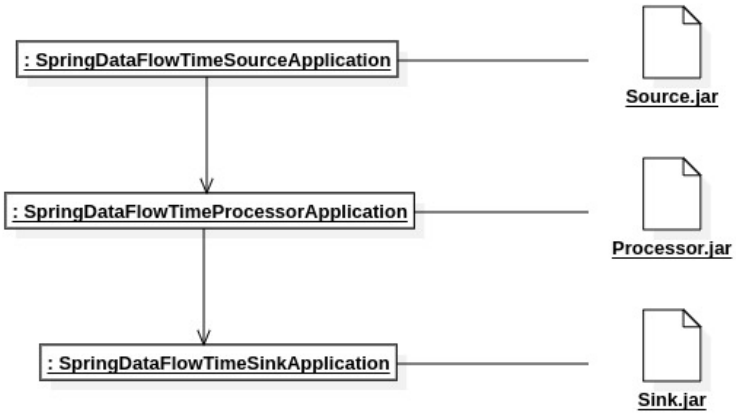


Figura 4 - Diagrama de obiecte

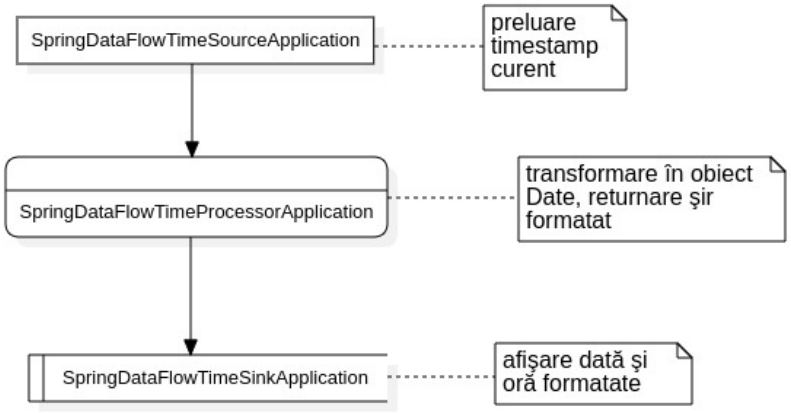


Figura 5 - Diagrama fluxului de date

Microserviciul sursă

Creaţi un proiect **Spring Boot** (folosind **Maven** sau **Gradle**), conform cu instrucţiunile din **laboratorul 3** de Sisteme Distribuite. **Nu aveţi nevoie de dependenţa Spring Web (`spring-boot-starter-web`).**

Adăugaţi dependenţa **Spring Cloud Starter Stream Rabbit** la proiect:

- **pentru Maven**: adăugaţi următoarea dependenţă în **`pom.xml`**:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
    <version>3.0.3.RELEASE</version>
</dependency>
```

- **pentru Gradle**: adăugaţi următorul *compile group* în **`build.gradle`**:

```
compile group: 'org.springframework.cloud', name: 'spring-cloud-
starter-stream-rabbit', version: '3.0.3.RELEASE'
```

Creaţi un fişier sursă, denumit **`Source.kt`**, cu următorul conţinut:

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Source
import org.springframework.context.annotation.Bean
import
org.springframework.integration.annotation.InboundChannelAdapter
import org.springframework.integration.annotation.Poller
import org.springframework.messaging.Message
import org.springframework.messaging.support.MessageBuilder
import java.util.*

@EnableBinding(Source::class)
@SpringBootApplication
class SpringDataFlowTimeSourceApplication {
    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT, poller =
[Poller(fixedDelay = "10000", maxMessagesPerPoll = "1")])
    fun timeMessageSource(): () -> Message<Long> {
        return { MessageBuilder.withPayload(Date().time).build() }
    }
}

fun main(args: Array<String>) {
    runApplication<SpringDataFlowTimeSourceApplication>(*args)
}
```

Se observă că aplicaţia Spring Boot conţine un *bean* care produce un mesaj o dată la 10 secunde. Mesajul conţine un obiect ***Timestamp*** în corpul acestuia. Clasa de configurare a aplicaţiei Spring este adnotată cu **`@EnableBinding(Source::class)`** pentru a fi legată de un *broker* (cel specificat între paranteze), şi deci pentru a primi rolul de aplicaţie sursă în pipeline.

Microserviciul procesor

La fel ca în cazul microserviciului sursă, creaţi un alt proiect **Spring Boot** şi adăugaţi dependenţa **Spring Cloud Starter Stream Rabbit** în maniera descrisă mai sus.

Adăugaţi un fişier sursă, denumit **`Processor.kt`**, cu următorul conţinut:

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import java.text.DateFormat
import java.text.SimpleDateFormat

@EnableBinding(Processor::class)
@SpringBootApplication
class SpringDataFlowTimeProcessorApplication {
    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
    fun transform(timestamp: Long?): Any? {
        val dateFormat: DateFormat = SimpleDateFormat("dd/MM/yyyy
hh:mm:ss")
        return dateFormat.format(timestamp)
    }
}

fun main(args: Array<String>) {
    runApplication<SpringDataFlowTimeProcessorApplication>(*args)
}
```

Acest microserviciu are rolul de procesor de date (conform adnotării **`@EnableBinding(Processor::class)`**) şi transformă valorile de tip ***Timestamp*** pe care le primeşte pe canalul de intrare în şiruri de caractere ce încapsulează data şi ora indicată de *timestamp*.

Microserviciul sink

Creaţi şi al 3-lea proiect Spring Boot în aceeaşi manieră, cu adăugarea dependenţei **Spring Cloud Starter Stream Rabbit**.

Adăugaţi un fişier sursă, denumit **`Sink.kt`**, cu următorul conţinut:

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.annotation.StreamListener
import org.springframework.cloud.stream.messaging.Sink

@EnableBinding(Sink::class)
@SpringBootApplication
class SpringDataFlowTimeSinkApplication {
    @StreamListener(Sink.INPUT)
    fun loggerSink(date: String) {
        println("Am primit urmatorul mesaj: $date")
    }
```

```
}

fun main(args: Array<String>) {
    runApplication<SpringDataFlowTimeSinkApplication>(*args)
}
```

Conform adnotării **`@EnableBinding(Sink::class)`**, aplicate clasei de configurare Spring, rolul celui de-al 3-lea microserviciu este de entitate *sink*, care primeşte mesajele transformate de la procesor.

Bean-ul **`loggerSink`** „ascultă” pe canalul de intrare şi primeşte rezultatul procesării din pipeline sub formă de şir de caractere şi îl afişează la consolă.

Instalarea microserviciilor în server-ul Data Flow local

1. Împachetare microservicii sub formă de artefacte JAR

Folosiţi *lifecycle*-ul Maven **`package`** sau *target*-ul Gradle **`bootJar`** pentru a împacheta cele 3 aplicaţii în artefacte JAR cu tot cu dependenţele necesare.

Artefactele rezultate vor fi plasate în următoarele locaţii:

- **pentru Maven**: **`target/<NUME_ARTEFACT>.jar`**
- **pentru Gradle**: **`build/libs/<NUME_ARTEFACT>.jar`**

2. Înregistrarea microserviciilor sub formă de aplicaţii Data Flow

Reveniţi la terminalul **Spring Data Flow Shell** deschis anterior.

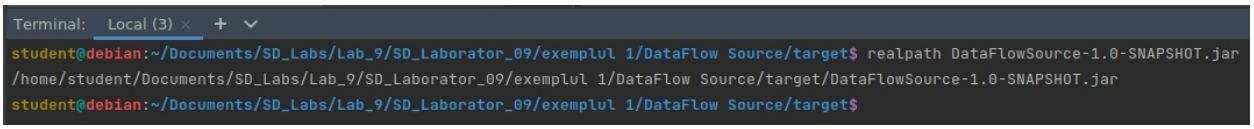
Înregistrarea unui microserviciu ca aplicaţie Data Flow se face cu următoarea comandă:

```
app register --name <NUME_APLICAŢIE> --type <TIP_APLICAŢIE> --uri
file:///CALE_CĂTRE_ARTEFACTUL_JAR_AL_APLICAŢIEI_SURSĂ
```

Înlocuiţi **`CALE_CĂTRE_ARTEFACTUL_JAR_AL_APLICAŢIEI_SURSĂ`** cu calea **absolută** către artefactul JAR generat în pasul de împachetare anterior.

Atenţie: în cazul în care calea conţine spaţii, înlocuiţi-le cu şirul de caractere `%20`, întrucât calea absolută este trimisă ca parametru sub formă de URI.

Ca să preluaţi în mod facil calea completă a artefactului JAR, puteţi deschide un terminal din IntelliJ în folder-ul care conţine acel artefact, apoi folosiţi comanda **`realpath`**:



Înlocuiţi toate spaţiile (dacă există) cu **`%20`** şi asamblaţi comanda completă de înregistrare pentru Data Flow Shell. Spre exemplu:

```
app register --name time-source --type source --uri
file:////home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul
%201/DataFlow%20Source/target/DataFlowSource-1.0-SNAPSHOT.jar
```

Procedaţi asemănător cu celelalte 2 microservicii procesor şi *sink*:

```
app register --name time-processor --type processor --uri
file:////home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul
%201/DataFlow%20Processor/target/DataFlowProcessor-1.0-SNAPSHOT.jar
```

```
app register --name logging-sink --type sink --uri
file:////home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul
%201/DataFlow%20Sink/target/DataFlowSink-1.0-SNAPSHOT.jar
```

Rezultatul comenzilor ar trebui să arate asemănător cu ce apare în captura următoare:

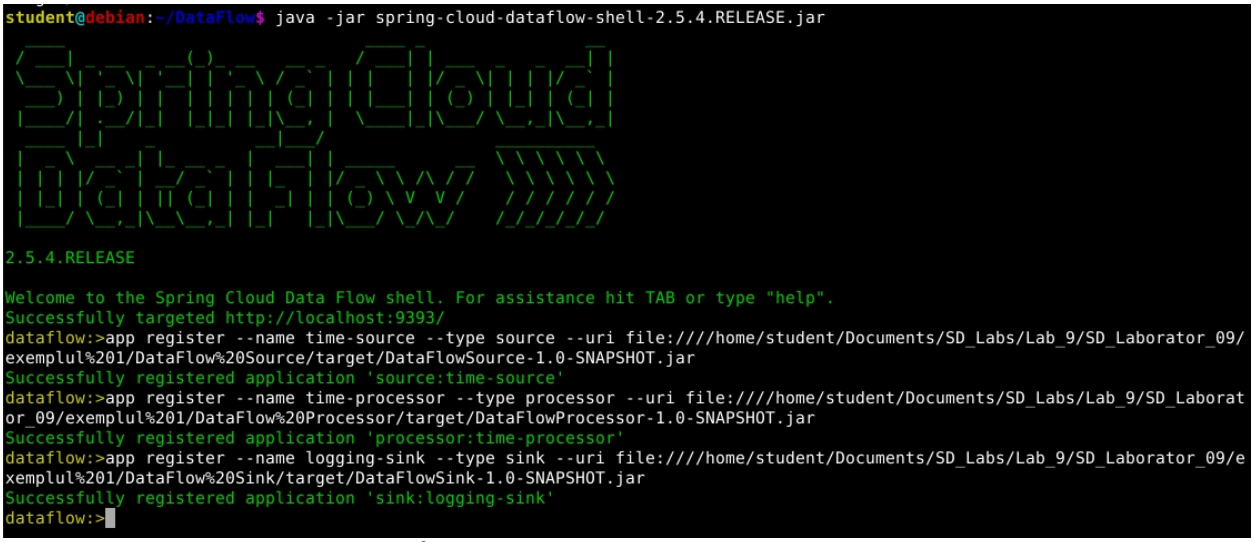


Figura 6 - Înregistrare cu succes a entităţilor din pipeline

Dacă este necesar, pentru a şterge o aplicaţie înregistrată pe serverul Data Flow, se poate folosi comanda:

```
app unregister --name <NUME_APLICAŢIE> --type <TIP_APLICAŢIE>
```

Crearea fluxului de date din entităţile înregistrate în Data Flow

Se creează fluxul de date conform diagramei din **Figura 5**, utilizând următoarea comandă în Data Flow Shell:

```
stream create --name time-to-log --definition 'time-source | time-
processor | logging-sink'
```

Observaţi sintaxa specifică a definiţiei fluxului de date (parametrul **--definition**). Aceasta provine de la sintaxa *pipe*-urilor UNIX, spre exemplu o comandă de tipul:

```
ls -l | grep abc | wc -l
```

Aşa cum în *shell*-ul Linux, comanda de sus creează un *pipeline* de 3 procese, conectând fluxul de ieşire al unuia la fluxul de intrare al următorului, aşa se comportă şi definiţia fluxului de date din Data Flow:

```
time-source | time-processor | logging-sink
```

Canalul de ieşire al entităţii **time-source** este conectat la canalul de intrare al entităţii time-processor, respectiv canalul de ieşire al **time-processor** este conectat la canalul de intrare al **logging-sink**.

Odată de fluxul de date a fost creat, acesta trebuie instalat (*deployed*) utilizând comanda următoare:

```
stream deploy --name time-to-log
```

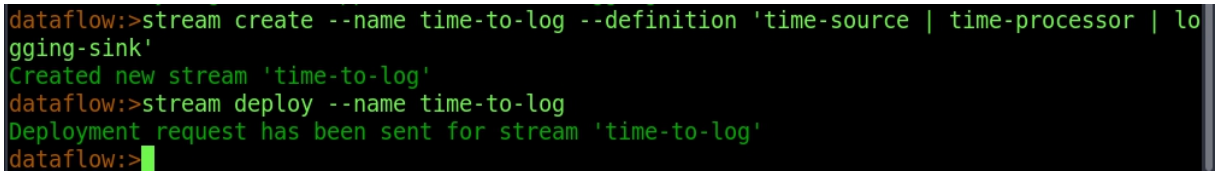


Figura 7 - Instalarea fluxului de date

La nevoie, pentru a dezinstala un flux de date, folosiţi comanda:

```
stream undeploy --name <NUME_STREAM>
```

În acest moment, aveţi *pipeline*-ul instalat în server-ul local Data Flow, iar entitatea **Sursă** va începe imediat să emită date.

Verificarea funcţionării fluxului de date

Pentru a verifica buna funcţionare a fluxului de date configurat anterior, puteţi folosi panoul grafic de configurare pus la dispoziţie la URL-ul: http://localhost:9393/dashboard

Accesaţi secţiunea **Apps** din meniul din stânga şi interfaţa vă afişează lista de aplicaţii instalate anterior. Veţi observa cele 3 entităţi pe care le-aţi instalat pe server:

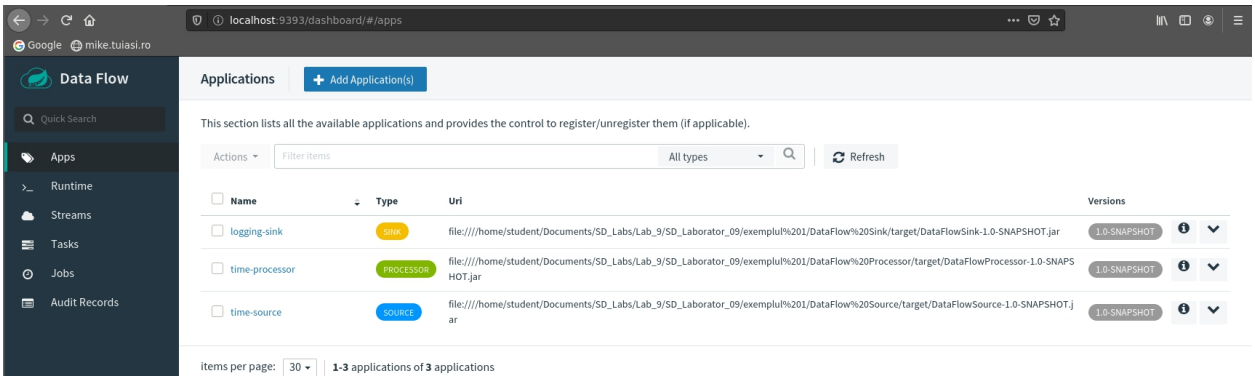


Figura 8 - Vizualizarea listei de aplicaţii Data Flow

Secţiunea „**Runtime**” vă afişează starea de funcţionare a aplicaţiilor. Dacă sunt marcate cu roşu, în starea „**Failed**”, atunci verificaţi log-urile pentru eventualele probleme apărute în cod sau la instalare (se explică mai jos).

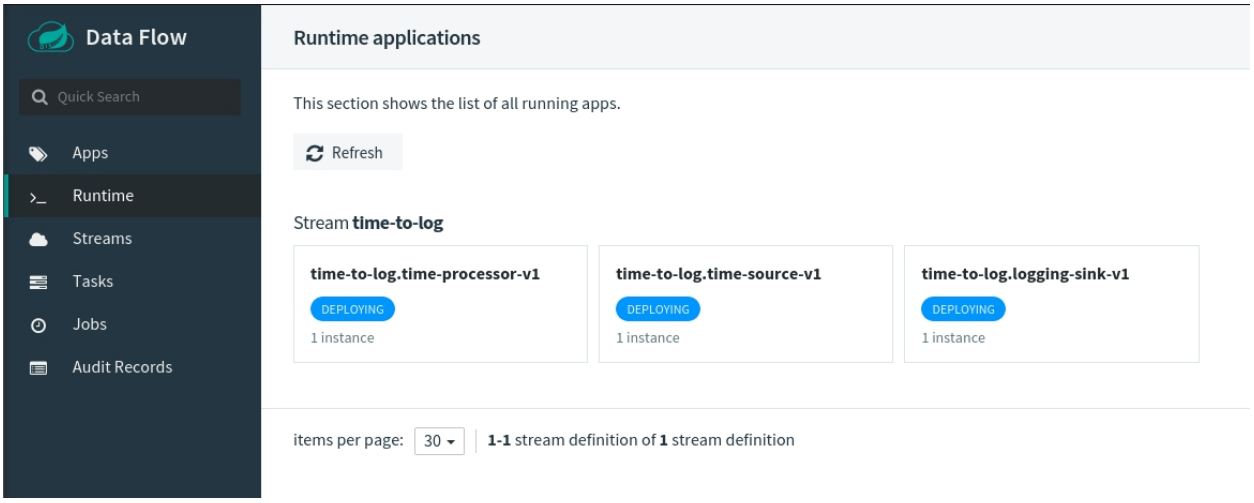


Figura 9 - Verificarea stării de funcţionare a aplicaţiilor

În secţiunea „**Streams**” puteţi vizualiza care sunt fluxurile de date configurate şi instalate.

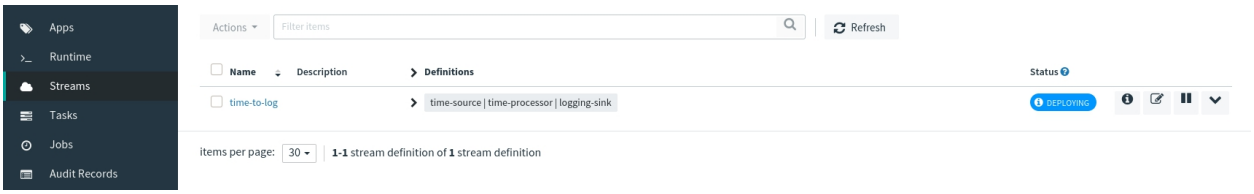


Figura 10 - Verificarea fluxurilor de date

Verificarea log-urilor

În secţiunea „**Runtime**”, apăsaţi pe entitatea de tip Sink (acolo ajung rezultatele procesării fluxului de date, deci acolo trebuie verificat *output*-ul aplicaţiei).

Din panoul deschis deasupra interfeţei, copiaţi calea către *log*-ul **`stdout`**:

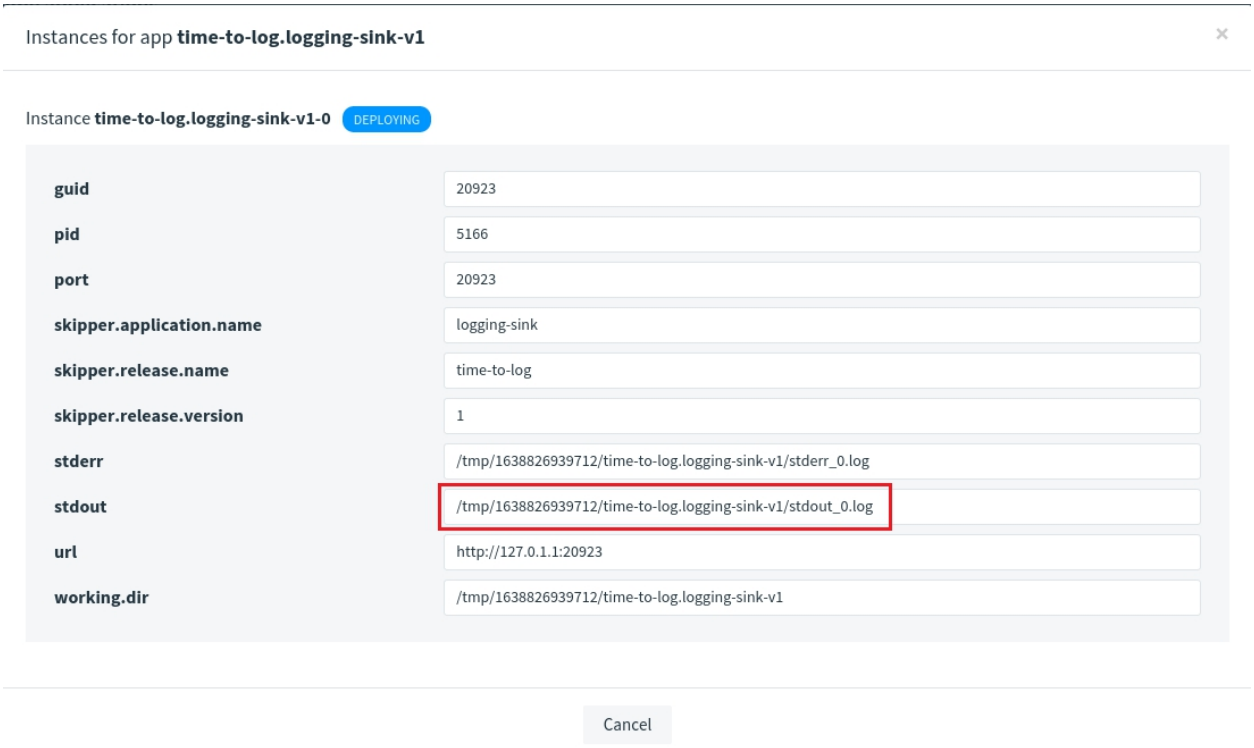


Figura 11 - Preluarea *log*-ului unei aplicaţii Data Flow

Deschideţi acel fişier log cu un editor de text şi astfel puteţi verifica ce anume a afişat aplicaţia la consolă (mai precis, ultima entitate din fluxul de date, *sink*-ul).

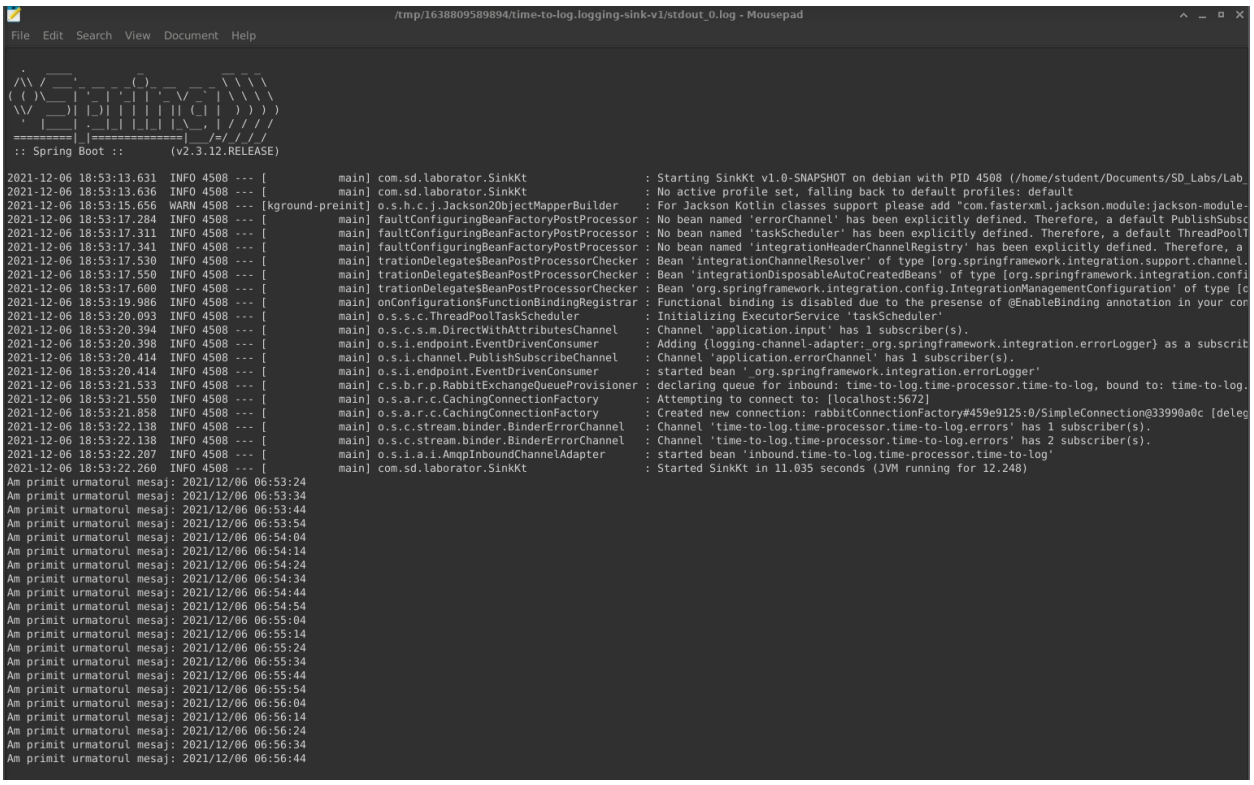


Figura 12 - Verificarea log-ului entității *Sink*

Reîncărcarea aplicațiilor din fluxul Data Flow după modificări în cod

Dacă modificați codul microserviciilor componente din fluxul de date, reîmpachetați aplicația / aplicațiile modificate și, din consola Data Flow Shell, dezinstalați și reinstalați fluxul, astfel:

```
stream undeploy --name time-to-log
stream deploy --name time-to-log
```

```
dataflow:>stream undeploy --name time-to-log
Un-deployed stream 'time-to-log'
dataflow:>stream deploy --name time-to-log
Deployment request has been sent for stream 'time-to-log'
dataflow:>
```

Exemplul 2 - aplicaţie Data Flow cu proiectare DDD (Domain Driven Design)

Pentru exemplul 2, se va proiecta o aplicaţie conform principiilor ***Domain Driven Design***. **Accentul se pune pe etapele de proiectare, implementarea efectivă va acoperi un set restrâns de cazuri de utilizare.**

Se recomandă consultarea următoarelor referinţe:

- noţiunile despre DDD sumarizate în **laboratorul 6** (pag. 4) de la disciplina Sisteme Distribuite
- capitolul **Domain-Driven Design (DDD) Principles and Patterns** din cartea **Architectural Patterns: Uncover Essential Patterns in the Most Indispensable Realm of Enterprise Architecture**, de Pethuru Raj Chelliah

Primul pas este identificarea fluxului de afaceri (eng. *business flow*), care modelează funcţionarea unei companii: care sunt paşii necesari pentru a îndeplini o anumită cerinţă de afacere (eng. *business request*). Aplicaţia din laborator va modela <u>într-un mod simplist</u> funcţionarea unei fabrici de produse Y, care primeşte comenzi de la clienţi şi livrează produsele acestora. În cazul în care cantitatea dorită de client nu este pe stoc, se cere fabricarea mai multor bucăţi şi aducerea lor pe stoc în depozit.

Diagrama de activitate - fluxul de afaceri

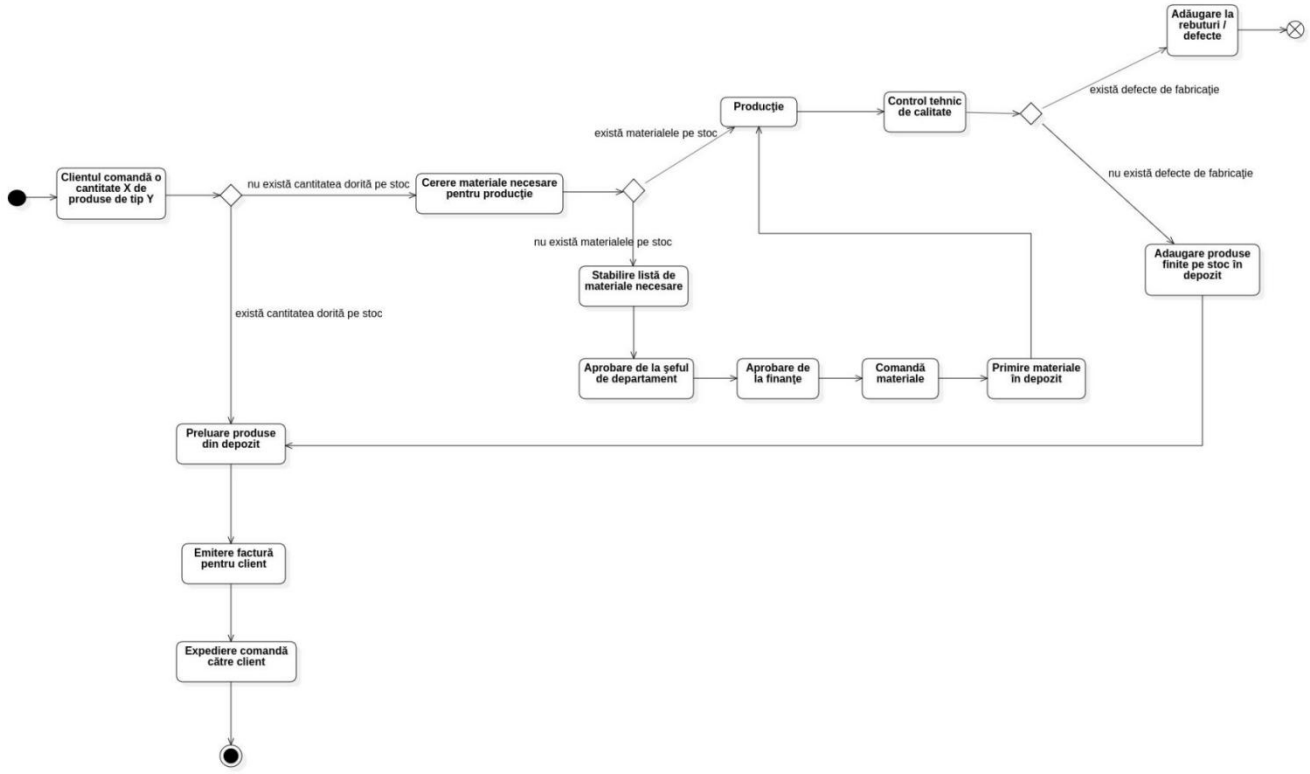


Figura 13 - Fluxul de afaceri

Diagrama de activitate - împărţirea pe departamente

O viziune mai clară asupra modului de funcţionare a afacerii modelate se poate observa în diagrama următoare, cu activităţile împărţite în funcţie de departamentul care le poate efectua.

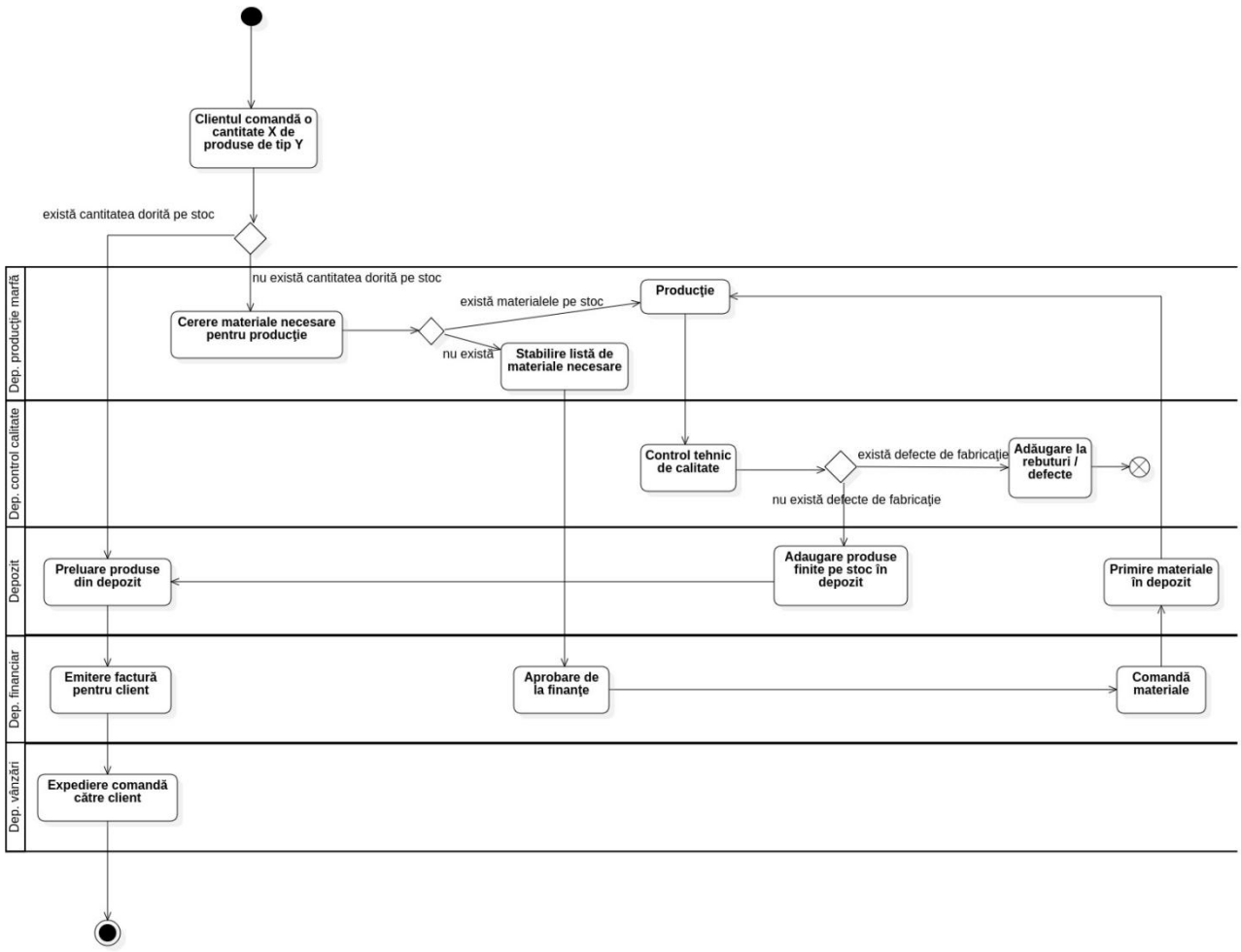


Figura 14 - Împărţirea pe departamente

Identificarea contextelor mărginite (bounded contexts)

Următorul pas este identificarea contextelor mărginite din fluxul de afacere. Acestea reprezintă linii conceptuale distinctive care definesc graniţele ce separă contextele de alte componente ale sistemului.

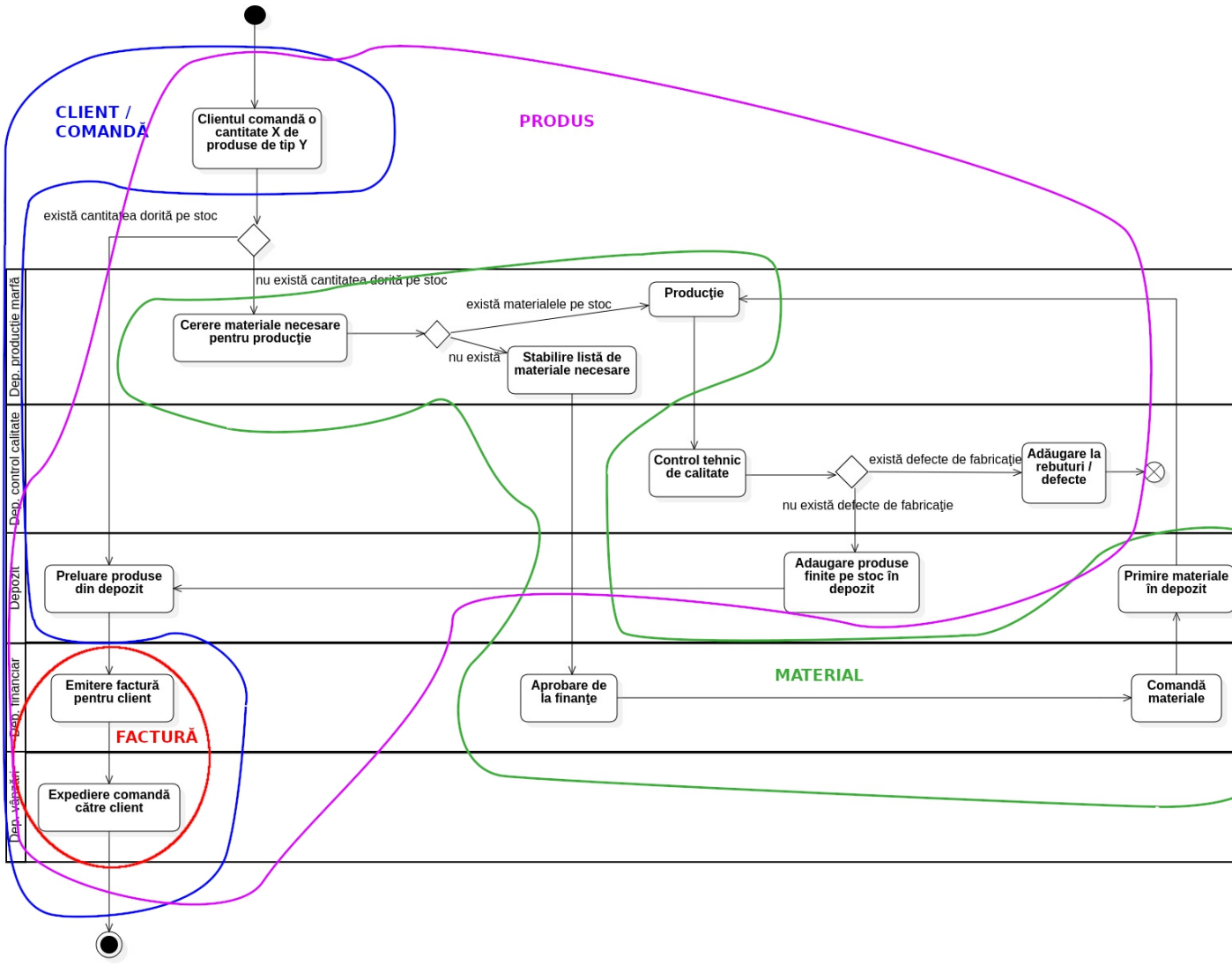


Figura 15 - Contextele mărginite

Identificarea entităţilor

O entitate reprezintă un obiect mutabil: acesta îşi poate schimba proprietăţile fără a-şi modifica identitatea. De exemplu, un **Produs** este o entitate: produsul este unic şi nu-şi va schimba identitatea (ceea ce îl distinge în mod unic) odată ce aceasta este stabilită. Totuşi, preţul, descrierea, şi alte atribute specifice pot fi schimbate de câte ori este nevoie.

Entităţile care reies, la o primă analiză, din fluxul de afacere modelat ar fi următoarele:

- CLIENT
- COMANDĂ
- PRODUS
- MATERIAL
- FACTURĂ

Pentru că avem nevoie şi de stratul de persistenţă s-a ales utilizarea unui SGBD care suportă SQL. Proprietăţile, respectiv relaţiile între aceste entităţi se pot modela utilizând o **diagramă entitate-relaţie** (*Entity-Relationship Diagram*).

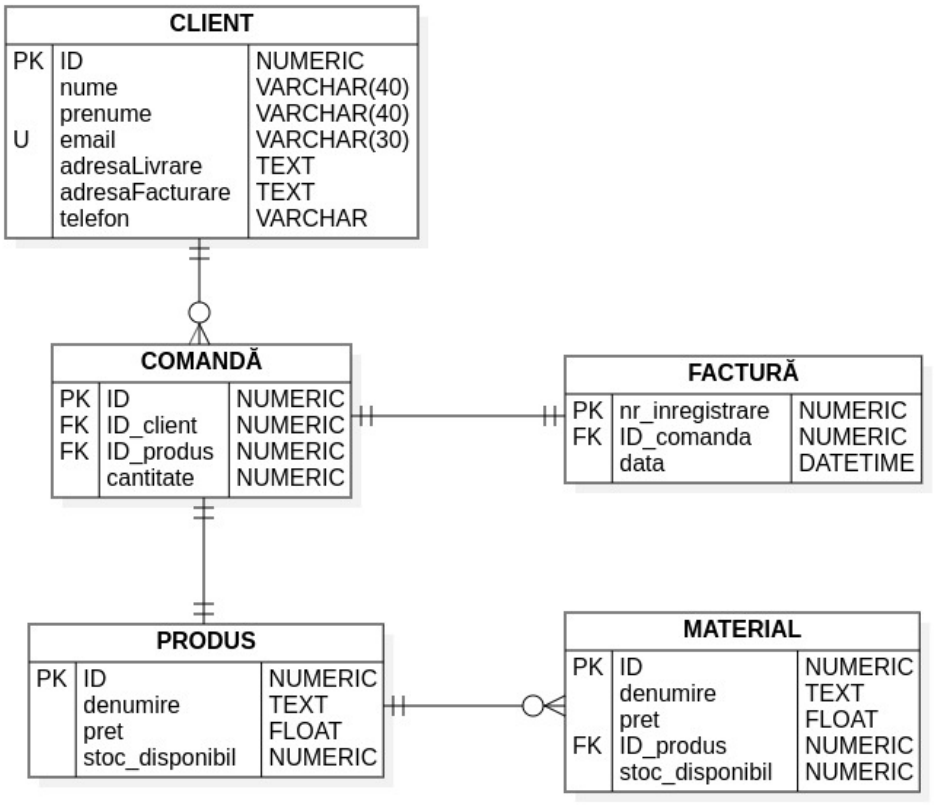


Figura 16 - Diagrama ERD

Identificarea serviciilor de domeniu (domain services)

În limbajul specific analizei aplicației (eng. *ubiquitous language*), există situații în care acțiunile (cerințele de afacere) nu pot fi atribuite unei entități sau a unui obiect de valoare (eng. *value object*). Acele operațiuni pot fi asociate unui așa numit **serviciu de domeniu** (eng. *domain service*). Aceste servicii încapsulează logica de domeniu dar și concepte care nu pot fi modelate ca și entități sau obiecte de valoare (o cârpăceală de fapt din punct de vedere al proiectării pentru sisteme mari - acolo astfel de nedeterminări sunt semnale de alarmă ca analiza și proiectarea au scăpat mai multe aspecte din vedere). Serviciile de domeniu sunt responsabile pentru orchestrarea logicii de afacere (eng. *business logic*).

Pentru fiecare context mărginit, se specifică în continuare care sunt serviciile de domeniu identificate. Aceste servicii sunt descrise utilizând **limbajul specific asociat respectivelor contexte (de fapt în acest caz trivial se crează simple asocieri de operații prin care trebuie sa treacă un produs (vezi rețelel petri temporizate și colorate)**:

- **PRODUS**
  - atunci când un client comandă un număr de produse, trebuie **cerut stocul** disponibil
  - **identificarea materialelor** necesare fabricării unui produs
  - **fabricarea** efectivă a produsului
  - după ce un produs este fabricat, trebuie **verificat pentru** eventualele **defecte de fabricație**
  - la detectarea unui defect, **produsul** în cauză **se adaugă la rebuturi**
  - **adăugarea unui produs finit pe stoc**, dacă acesta nu are defecte de fabricație
  - **emiterea facturii** pentru un anumit produs comandat de client
  - după ce cantitatea de produse dorite de client este disponibilă, comanda **se expediază**

către adresa clientului

- **MATERIAL**
  - **materialele sunt cerute de pe stoc** în cazul în care un produs nu există în cantitatea suficientă
  - dacă nu există suficiente materiale de producţie, **se cere aprobarea achiziţionării lor** de la departamentul financiar
  - **materialele necesare sunt comandate** de la furnizori de către departamentul financiar
  - odată ce comanda de **materiale** este onorată, acestea **se adaugă pe stoc** în depozit
  - **utilizarea materialelor de producţie** pentru fabricarea unui produs
- **CLIENT**
  - clientul **se înregistrează în sistem** pentru a putea da comenzi
  - clientul poate **comanda o anumită cantitate de un anumit tip de produs**: „doresc X bucăţi din produsul de tip Y”
  - clientului **îi este emisă o factură** pe comanda plasată, după ce produsele comandate sunt disponibile pentru livrare
  - clientului îi este **expediată comanda** pe care a plasat-o
- **COMANDĂ**
  - comanda este **înregistrată în sistem** după ce este plasată de client
  - pe baza unei comenzi înregistrate, **se emite o factură**, după ce produsele sunt pregătite pentru livrare
  - comanda **este expediată clientului** după ce este pregătită pentru livrare
- **FACTURĂ**
  - factura este emisă clientului după ce comanda sa este pregătită pentru livrare
  - comanda expediată **corespunde şi include factura** aferentă pentru client

Proiectarea diagramei de clase

Conform serviciilor de domeniu identificate mai sus, se proiectează diagrama de clase care va scoate în evidenţă şi modalităţile de implementare a cerinţelor de afacere (*business requirements*). Abordarea este greșită din punct de vedere al proiectării cu microservicii unde trebuia să începem cu diagrama de microservicii și apoi să vedem descrierea fiecăruia la un nivel mai fin de proiectare (de exemplu utilizând diagrama de clase). Cei buni ar trebui să treacă mai întâi prin această fază de proiectare. Totuși pentru simplitatea exemplului în acest caz s-a făcut rabat la aceste reguli și s-a trecut direct la diagrama de clase.

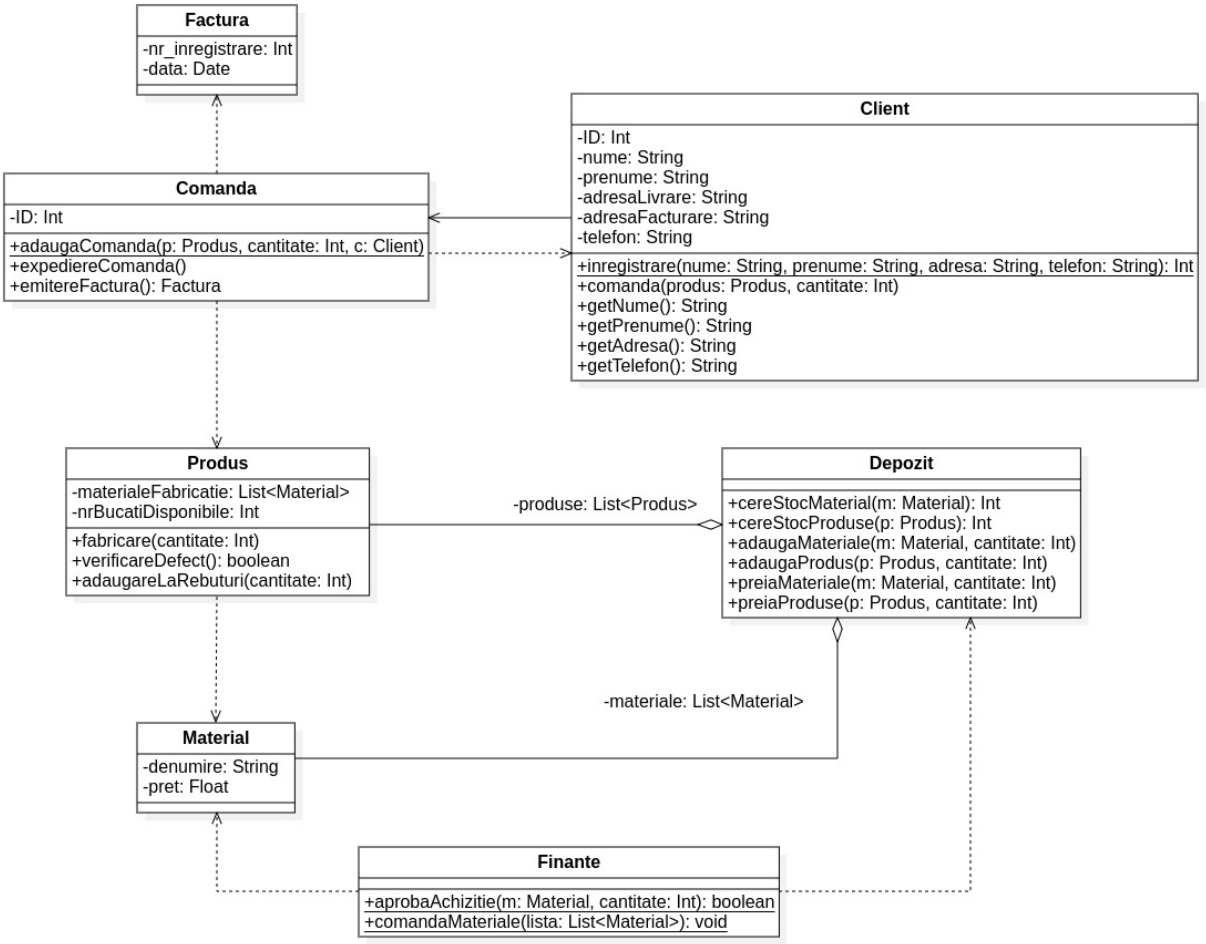


Figura 17 - Diagrama de clase

Reducerea cuplărilor şi respectarea GDPR

Această etapă presupune asigurarea, pe cât posibil, a unei cuplări cât mai slabe între entităţile implicate în fluxul de afacere. De asemenea, din punct de vedere al **GDPR**, trebuie minimizată răspândirea informaţiilor cu caracter personal prin fluxul de mesaje.

Spre exemplu, depozitul modelat în aplicaţia exemplu nu trebuie să ştie pentru ce client pregăteşte şi împachetează produsele pentru livrare. Pachetul va fi etichetat corespunzător înainte de a fi predat curierului, şi abia atunci se poate accesa adresa de livrare şi datele strict necesare ale clientului.

Asemănător, datele clientului nu au nicio relevanţă şi nu trebuie să ajungă în departamentul financiar, care doar comandă materiale necesare pentru producţie, indiferent ce comandă a cărui client se datorează acestui fapt.

Din punct de vedere al identificării unui client sau a unei comenzi prin fluxul de mesaje, se pune problema încapsulării unui identificator unic, care este suficient pentru păstrarea consistenţei informaţiilor pe parcursul fluxului.

Proiectarea şi implementarea microserviciilor

Microserviciile se proiectează şi implementează corespunzător unui set restrâns de cazuri de utilizare, pentru simplitate. Aşadar, implementarea din laborator se va conforma următorului graf de lucru:

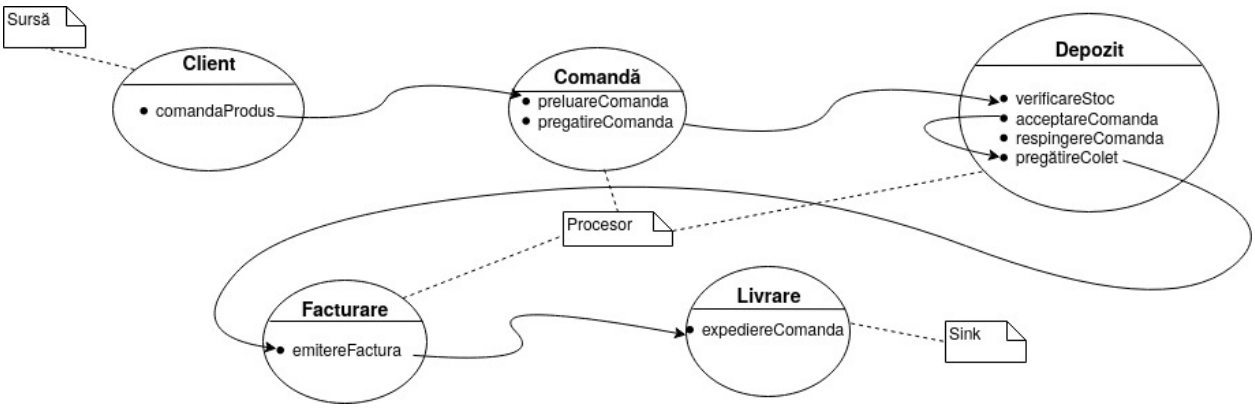


Figura 18 - Diagrama de microservicii

S-au marcat pe diagrama de mai sus şi cele 3 tipuri de entităţi care intervin în *pipeline*-ul care se formează în consecinţă.

Fiecare microserviciu va fi creat într-un proiect Spring Boot separat, conform modelului din exemplul 1.

Client - microserviciu Sursă

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Source
import org.springframework.context.annotation.Bean
import
org.springframework.integration.annotation.InboundChannelAdapter
import org.springframework.integration.annotation.Poller
import org.springframework.messaging.Message
import org.springframework.messaging.support.MessageBuilder
import kotlin.random.Random

@EnableBinding(Source::class)
@SpringBootApplication
class ClientMicroservice {
    companion object {
        val listaProduse: List<String> = arrayListOf(
            "Masca protectie",
            "Vaccin anti-COVID-19",
            "Combinezon",
            "Manusa chirurgicala"
        )

        ///TODO - lista de produse sa fie preluata din baza de date /
din fisier
    }

    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT, poller =
[Poller(fixedDelay = "10000", maxMessagesPerPoll = "1")])
    fun comandaProdus(): () -> Message<String> {
        return {
            val produsComandat = listaProduse[(0 until
```

```
listaProduse.size).shuffled()[0]]
            val cantitate: Int = Random.nextInt(1, 100)
            val identitateClient = "Popescu Ion"
            val adresaLivrare = "Codrii Vlasiei nr 14"

            ///TODO - datele clientului sa fie citite dinamic si
inregistrate in baza de date

            val mesaj =
"$identitateClient|$produsComandat|$cantitate|$adresaLivrare"
            MessageBuilder.withPayload(mesaj).build()
        }
    }
}

fun main(args: Array<String>) {
    runApplication<ClientMicroservice>(*args)
}
```

Comandă - microserviciu Procesor

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import org.springframework.messaging.support.MessageBuilder
import java.text.DateFormat
import java.text.SimpleDateFormat
import kotlin.random.Random

@EnableBinding(Processor::class)
@SpringBootApplication
class ComandaMicroservice {
    private fun pregatireComanda(produs: String, cantitate: Int): Int
{
        println("Se pregateste comanda $cantitate x \"$produs\"...")

        ///TODO - asignare numar de inregistrare
        ///TODO - inregistrare comanda in baza de date
        return Random.nextInt()
    }

    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
    fun preluareComanda(comanda: String?): String {
        val (identitateClient, produsComandat, cantitate,
adresaLivrare) = comanda!!.split("|")
        println("Am primit comanda urmatoare: ")
        println("$identitateClient | $produsComandat | $cantitate |
$adresaLivrare")

        val idComanda = pregatireComanda(produsComandat,
cantitate.toInt())
```

```
        ///TODO - in loc sa se trimita mesajul cu toate datele in
continuare, trebuie trimis doar ID-ul comenzii
        return "$comanda"
    }
}

fun main(args: Array<String>) {
    runApplication<ComandaMicroservice>(*args)
}
```

Depozit - microserviciu Procesor

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import org.springframework.messaging.support.MessageBuilder
import kotlin.random.Random

@EnableBinding(Processor::class)
@SpringBootApplication
class DepozitMicroservice {
    companion object {
        ///TODO - modelare stoc depozit (baza de date cu stocurile de
produse)
        val stocProduse: List<Pair<String, Int>> = mutableListOf(
            "Masca protectie" to 100,
            "Vaccin anti-COVID-19" to 20,
            "Combinezon" to 30,
            "Manusa chirurgicala" to 40
        )
    }

    private fun acceptareComanda(identificator: Int): String {
        println("Comanda cu identificatorul $identificator a fost
acceptata!")

        val produsDeExpediat = stocProduse[(0 until
stocProduse.size).shuffled()[0]]
        val cantitate = Random.nextInt(produsDeExpediat.second)

        return pregatireColet(produsDeExpediat.first, cantitate)
    }

    private fun respingereComanda(identificator: Int): String {
        println("Comanda cu identificatorul $identificator a fost
respinsa! Stoc insuficient.")
        return "RESPINSA"
    }

    private fun verificareStoc(produs: String, cantitate: Int):
Boolean {
```

```
        ///TODO - verificare stoc produs
        return true
    }

    private fun pregatireColet(produs: String, cantitate: Int): String
{
        ///TODO - retragere produs de pe stoc in cantitatea
specificata
        println("Produsul $produs in cantitate de $cantitate buc. este
pregatit de livrare.")
        return "$produs|$cantitate"
    }

    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
    ///TODO - parametrul ar trebui sa fie doar numarul de inregistrare
al comenzii si atat
    fun procesareComanda(comanda: String?): String {
        val identificatorComanda = Random.nextInt()
        println("Procesez comanda cu identificatorul
$identificatorComanda...")

        ///TODO - procesare comanda in depozit
        val rezultatProcesareComanda: String = if (verificareStoc("",
100)) {
            acceptareComanda(identificatorComanda)
        } else {
            respingereComanda(identificatorComanda)
        }

        ///TODO - in loc sa se trimita mesajul cu toate datele in
continuare, trebuie trimis doar ID-ul comenzii
        return "$comanda"
    }
}

fun main(args: Array<String>) {
    runApplication<DepozitMicroservice>(*args)
}
```

Facturare - microserviciu Procesor

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import kotlin.random.Random

@EnableBinding(Processor::class)
@SpringBootApplication
class FacturareMicroservice {
    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
```

```
    ///TODO - parametrul ar trebui sa fie doar numarul de inregistrare
al comenzii si atat
    fun emitereFactura(comanda: String?): String {
        val (identitateClient, produsComandat, cantitate,
adresaLivrare) = comanda!!.split("|")
        println("Emit factura pentru comanda $comanda...")
        val nrFactura = abs(Random.nextInt())
        println("S-a emis factura cu nr $nrFactura.")

        ///TODO - inregistrare factura in baza de date

        return "$comanda"
    }
}

fun main(args: Array<String>) {
    runApplication<FacturareMicroservice>(*args)
}
```

Livrare - microserviciu Sink

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.annotation.StreamListener
import org.springframework.cloud.stream.messaging.Sink

@EnableBinding(Sink::class)
@SpringBootApplication
class LivrareMicroservice {
    @StreamListener(Sink.INPUT)
    ///TODO - parametrul ar trebui sa fie doar numarul de inregistrare
al comenzii si atat
    fun expediereComanda(comanda: String) {
        println("S-a expediat urmatoarea comanda: $comanda")
    }
}

fun main(args: Array<String>) {
    runApplication<LivrareMicroservice>(*args)
}
```

Instalarea pipeline-ului în Data Flow

Înregistraţi cele 5 aplicaţii în Data Flow conform tipului fiecăreia dintre ele. Apoi, configuraţi următorul pipeline:

```
client | comanda | depozit | facturare | livrare
```

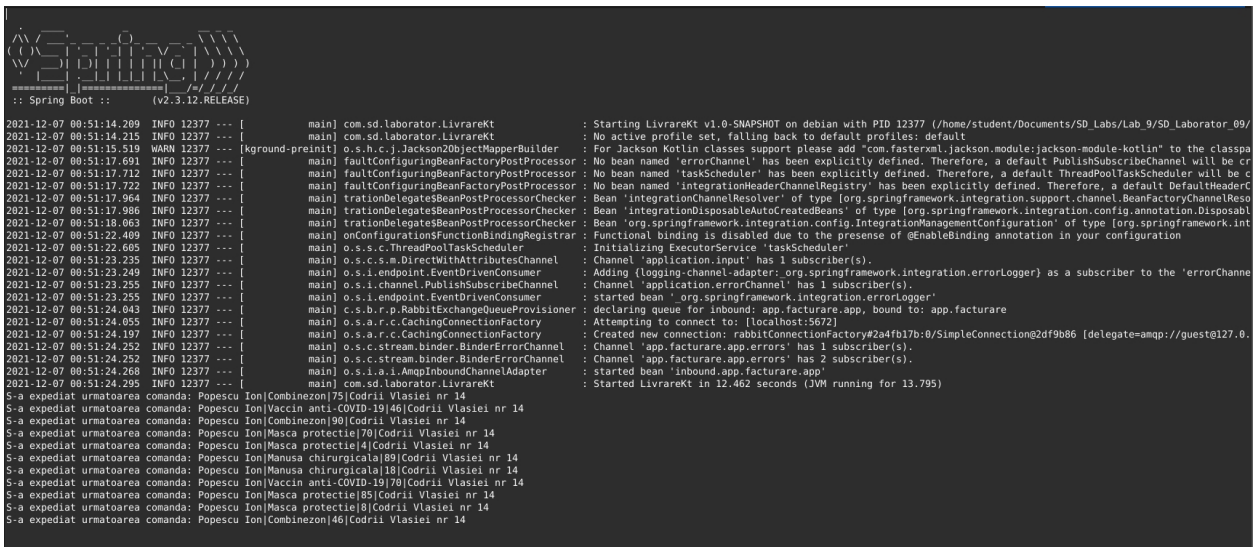


Figura 19 - Verificarea log-ului microserviciului de Livrare

Aplicaţii şi teme

Temă de laborator

Completaţi aplicaţia a 2-a din laborator utilizând scheletul de cod dat. Pentru simplitate, la laborator folosiţi fişiere text pe post de „bază de date”. Instalaţi aplicaţiile şi configuraţi *pipeline*-ul în Data Flow.

Teme pentru acasă

1. Implementaţi o aplicaţie care să poată executa comenzi Linux sub formă de *pipeline*, fără a folosi *pipe*-urile *shell*-ului, ci folosind microservicii care interacţionează conform unui automat de stare simulat prin interacţiuni simultane între mai multe *pipeline-uri de tip* Spring Data Flow.

   Exemplu de comenzi conectate prin *pipeline*:

```
fortune | cowsay | lolcat
```

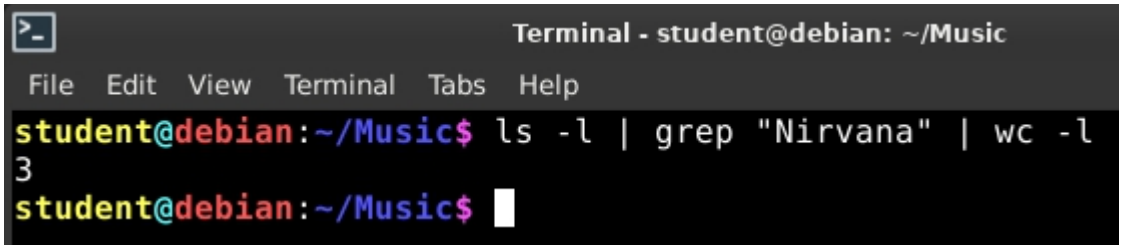
Exemplu de rezultat:

(Aceste comenzi sunt puse la dispoziţie de pachetele corespunzătoare numelor lor, pe care le puteţi instala utilizând comanda: **`sudo apt install fortune cowsay lolcat`**)

Un alt exemplu de pipeline Linux care afişează numărul de melodii de la formaţia Nirvana dintr-un folder:

```
ls -l | grep "Nirvana" | wc -l
```

Exemplu de rezultat:



Sfaturi pentru rezolvare:

• microserviciul **sursă** poate citi o listă predefinită de comenzi scrise într-un fişier, câte una pe o linie, iar fiecare comandă va fi emisă în pipeline după un anumit timp.

Pentru simplitate, puteţi considera comenzi care au **număr fix de *pipe*-uri** (simbolul „|“), stabilit de la bun început apoi la analiza unui număr variabil de cazuri (să spunem 3-4) va rezulta acel automat de stare (care descrie procesarea) .

• **procesoarele de flux** preiau câte o parte din comanda Linux şi execută partea „cea mai din stânga”, după care pasează mai departe rezultatul parţial, împreună cu restul comenzii care a mai rămas de executat către următorul procesor, ş.a.m.d.

Exemplu:

```
ls -l | grep "Nirvana" | wc -l
```

Primul procesor din flux va executa **`ls -l`**, preia rezultatul parţial şi trimite mai departe către următorul procesor acest rezultat, împreună cu ce a mai rămas din comandă:

```
grep "Nirvana" | wc -l
```

Al doilea procesor primeşte acel rezultat şi comanda rămasă de mai sus, aplică **`grep "Nirvana"`** pe ce a primit, apoi rezultatul îl trimite mai departe către următorul procesor, împreună cu ce a mai rămas din comandă:

```
wc -l
```

Ultimul procesor încheie *pipeline*-ul, prin execuţia ultimei părţi (**`wc -l`**) pe rezultatul primit de la procesorul anterior.

• entitatea ***sink*** va fi consola (log-ul **`stdout`**), în care afişaţi rezultatele prelucrării comenzilor în pipeline

• luaţi în considerare cazul în care numărul de comenzi din *pipeline* este variabil. Deci, **procesoarele de flux** sunt în număr dinamic: nu se ştie câte comenzi intermediare există în pipeline, aşadar, în funcţie de numărul de pipe-uri (simboluri „|”) din comandă, se vor instala (*deploy*) atâtea microservicii procesor câte sunt necesare (mai exact **numărul de *pipe*-uri** din comanda citită).

• procesoarele de flux să fie scalate orizontal folosind Docker. Puteţi folosi aplicaţii predefinite Spring Data Flow dintr-un pachet Stream App Starter cu Docker: https://cloud.spring.io/spring-cloud-stream-app-starters/

Se poate importa un pachet de aplicaţii în Data Flow astfel:

Din Data Flow Shell, se foloseşte comanda:

```
app import --uri https://dataflow.spring.io/rabbitmq-docker-latest
```

```
dataflow:>app import --uri https://dataflow.spring.io/rabbitmq-docker-latest
Successfully registered applications: [source.tcp, sink.twitter-update, source.http, sink.jdb
c, sink.rabbit, source.rabbit, sink.mongodb, source.ftp, processor.object-detection, processo
r.image-recognition, sink.rsocket, processor.transform, processor.twitter-trend, source.sftp,
 processor.filter, source.file, sink.cassandra, sink.router, sink.twitter-message, source.twi
tter-message, source.mqtt, processor.splitter, processor.groovy, sink.redis, source.load-gene
rator, sink.sftp, processor.semantic-segmentation, source.websocket, sink.file, source.cdc-de
bezium, processor.http-request, processor.header-enricher, source.time, sink.wavefront, proce
ssor.script, sink.mqtt, sink.tcp, source.jdbc, source.geode, sink.analytics, processor.bridge
, source.s3, sink.ftp, sink.log, processor.aggregator, sink.elasticsearch, sink.throughput, s
ource.zeromq, sink.s3, source.jms, sink.zeromq, sink.websocket, source.mongodb, source.mail,
source.twitter-stream, sink.pgcopy, source.twitter-search, sink.geode, source.syslog]
dataflow:>
```

2. Să se modifice exemplul 2 din laborator astfel:
   a) realizarea unei interfețe GUI (puteți merge pe Flask, pyQt, Qt)
   b) utilizarea unei baze de date relaţionale (de ex. SQLite) evident încapsulată ca un microserviciu
   c) să se trateze cazul în care o comandă nu poate fi onorată din cauza stocului lipsă: în acest caz, comanda va fi păstrată în contul clientului pentru reutilizare ulterioară (se modifică diagrama de uservicii, verific SOLID pentru userv si apoi descriu implementarea cu clase (de ex)).
   d) când se adună mai multe comenzi (minim 3) pentru acelaşi produs, depozitul să genereze o comandă pentru respectivul produs către producător (deci, un microserviciu separat pentru producător şi o funcţie dedicată în acest sens în microserviciul **Depozit**). Se va modifica descrierea ER astfel încât să se poată stabili diferențiat acest număr minim de produse. Se vor utiliza evenimente nor în implementare.

Bibliografie

[1]: Instalare Spring Data Flow - https://dataflow.spring.io/docs/installation/local/manual/
[2]: Documentaţie Spring Data Flow - https://dataflow.spring.io/docs/
[3]: Înregistrarea aplicaţiilor flux în server-ul local de Spring Data Flow - https://docs.spring.io/spring-cloud-dataflow-server-cloudfoundry/docs/1.0.0.M4/reference/html/spring-cloud-dataflow-register-apps.html
[4]: Procesarea fluxurilor de date utilizând Data Flow şi RabbitMQ - https://dataflow.spring.io/docs/stream-developer-guides/streams/data-flow-stream/
[5]: Fluxuri de date Spring Cloud Data Flow - https://docs.spring.io/spring-cloud-dataflow/docs/2.5.0.BUILD-SNAPSHOT/reference/htmlsingle/#spring-cloud-dataflow-streams
[6]: Stream Pipeline DSL - https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#spring-cloud-dataflow-stream-intro-dsl
[7]: Architectural Patterns: Uncover Essential Patterns in the Most Indispensable Realm of Enterprise Architecture, Pethuru Raj Chelliah